Asymptotically Optimal Encodings of Range Data Structures for Selection and Top-k Queries *

Roberto Grossi[†] John Iacono[‡] Gonzalo Navarro[§] Rajeev Raman[¶]
S. Rao Satti[‖]

Abstract

Given an array $A[1,n]$ of elements with a total order, we consider the problem of building a data structure that solves two queries: (a) selection queries receive a range $[i,j]$ and an integer k and return the position of the kth largest element in $A[i,j]$; (b) top-k queries receive $[i,j]$ and k and return the positions of the k largest elements in $A[i,j]$. These problems can be solved in optimal time, $O(1+\lg k/\lg\lg n)$ and $O(k)$, respectively, using linear-space data structures.

We provide the first study of the *encoding* data structures for the above problems, where A cannot be accessed at query time. Several applications are interested in the relative order of the entries of A, and their positions, rather their actual values, and thus we do not need to keep A at query time. In those cases, encodings save storage space: we first show that any encoding answering such queries requires $n\lg k - O(n+k\lg k)$ bits of space; then, we design encodings using $O(n\lg k)$ bits, that is, asymptotically optimal up to constant factors, while preserving optimal query time.

1 Introduction

A frequent problem in data and log mining applications is to find highest or lowest values in a range of a stream: the coldest days in a time period, peaks in the stock market, most popular terms in Twitter, most frequent queries in Google, and so on. As a less obvious scenario, consider autocompletion search in databases [22, 24]. As the user types in a query, the system presents the k most highly scoring (i.e., the most popular) completions of the text entered so far, chosen from a lexicon of phrases. Viewing the lexicon as a sorted sequence of strings with scores stored in an array A, the system maintains the range $[i,j]$ of the phrases prefixed by text typed in so far, and chooses the strings with the k highest scores in $A[i,j]$. Similarly, in Web search engines, A could contain the sequence of PageRank values of the pages in an inverted list sorted by URL. Then we

*Early partial versions of this article appeared in *Proc. ESA 2013* and *Proc. FSTTCS 2014*. Grossi partially funded by MIUR PRIN 2012C4E3KT national research project; Navarro funded in part by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F; Satti partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant number 2012-0008241).

[†]Department of Informatics, University of Pisa, Italy
[‡]Department of Computer Science and Engineering, Polytechnic Institute of New York University, USA
[§]Department of Computer Science, University of Chile, Chile
[¶]Department of Computer Science, University of Leicester, UK
[‖]School of Computer Science and Engineering, Seoul National University, Korea

could efficiently retrieve the k most highly ranked pages that contain a query term, restricted to a range of page identifiers (which can model a domain of any granularity). The problem is, again, to find the k highest values in a range $A[i, j]$. Directly finding the kth highest value may also be of interest. For example, in interfaces that show the first k results and then, upon user request, the next k, it is useful to obtain the $(k+1)$th to $2k$th results without having to obtain the first k results again.

The research work presented in this paper is motivated by the observation that, in these examples, the actual contents of A are not interesting by themselves (e.g., the scores are not reported). All we need is to find the *positions* in A where the highest values occur in a range. Hence storage of the contents of A could be avoided if we had a way to find those highest values *without accessing* A at query time.

We now formalize the problem of interest. Consider an array $A[1, n]$ of integers, reals, or in general any totally sorted universe. We are interested in the following two queries on A:

1. Selection queries: $\mathsf{sel}(i, j, k)$ returns the position of the kth largest value in range $A[i, j]$, for any given $1 \leq i \leq j \leq n$ and $1 \leq k \leq j - i + 1$.

2. Top-k queries: $\mathsf{top}(i, j, k)$ returns the positions of the k largest values in $A[i, j]$, in sorted order of value, for any given $1 \leq i \leq j \leq n$ and $1 \leq k \leq j - i + 1$.

Since these queries are sensitive only to the relative order between elements of A, and not to the actual values, we can replace the values in A by their *rank* (i.e., their position after sorting A in increasing order, breaking ties arbitrarily), and all the $\mathsf{sel}(\cdot)$ and $\mathsf{top}(\cdot)$ queries will return correct answers. Thus, in the sequel, we will consider that A is already a permutation of $[n]$ without loss of generality.

While optimal-time solutions exist for implementing those two queries, in this article we are interested in a kind of data structures called an *encoding*. An encoding is a data structure that, after preprocessing A, can answer queries on A without accessing A itself. Encodings are interesting when they use less space than that necessary to represent A (let us call it $|A|$). Otherwise, any data structure allowed to use $O(|A|)$ space could be modified to contain a copy of A inside, and then trivially become an encoding. Thus, interesting encodings cannot, by definition, recover all the values of A, but they can still answer the predefined queries for which they have been designed.

In our case, since A stores a permutation of $[n]$ and thus its storage requires $|A| \geq \lg_2 n! = \Theta(n \lg n)$ bits, we will be interested in encodings that use $o(n \lg n)$ bits. Such encodings are useful when the values in A are intrinsically uninteresting and only the indices where the $\mathsf{sel}(\cdot)$ or $\mathsf{top}(\cdot)$ values occur are sufficient, which is the case for the applications mentioned before.

Contributions. Since encodings do not access the data in A, a first question is what is the minimum size an encoding must have in order to answer the desired queries, irrespectively of the query time. In Section 3 we prove with a simple argument that any encoding solving either $\mathsf{sel}(\cdot)$ or $\mathsf{top}(\cdot)$ queries requires $n \lg k - O(n + k \lg k)$ bits of space, even if we restrict the query ranges to *one-sided* queries, of the form $A[1, j]$.

This shows that there are inherent limitations in space saving: we cannot hope to have an interesting encoding that works for any value of k, because values where $\lg k = \Theta(\lg n)$ would require encodings of $\Theta(n \lg n)$ bits, which are not interesting according to our definition. Still the challenge is to find encodings for some given maximum k value, κ, which handle queries for any $1 \leq k \leq \kappa$. Thus we can aim at encodings of size $O(n \lg \kappa) = o(n \lg n)$ when $\lg \kappa = o(\lg n)$.

The core of our research work aims at an encoding that, in $O(n \lg \kappa)$ bits of space, solves queries $\mathsf{sel}(i, j, k)$ in time $O(1+\lg k/\lg\lg n)$, for any $1 \le k \le \kappa$. The space is optimal up to constant factors, whereas the time is optimal for any structure using $O(n \operatorname{polylog} n)$ space [23]. Then we show how the structure for $\mathsf{sel}(\cdot)$ can also be used to solve $\mathsf{top}(i, j, k)$ queries in optimal time, $O(k)$. As a special case, we also show that $\mathsf{sel}(\cdot)$ queries can break the time lower bound for $\mathsf{sel}(1, j, \kappa)$ queries, that is, if they are *one-sided* and work only for $k = \kappa$ fixed at construction time. All our time results hold on a RAM machine with words of $w = \Theta(\lg n)$ bits.

Related work. The $\mathsf{sel}(\cdot)$ and $\mathsf{top}(\cdot)$ query problems are a natural extension of the well-known *range maximum query (RMQ)* problem, which corresponds to both $\mathsf{sel}(\cdot)$ and $\mathsf{top}(\cdot)$ with $k = 1$: namely, query $\mathsf{rmq}(i, j)$ looks for the position of the largest value in $A[i, j]$. The problem of encoding RMQs is well studied [12,14,32]. Fischer and Heun [14] gave an encoding of A that uses $2n+o(n)$ bits and answers RMQs in $O(1)$ time; their space bound is asymptotically optimal to within lower-order terms. The case $k = 2$ was studied more recently by Davoodi et al. [11], obtaining $3.272n + o(n)$ bits of space and $O(1)$ time.

We are not aware of any previous work on $\mathsf{sel}(\cdot)$ or $\mathsf{top}(\cdot)$ encoding for general k. After the conference versions of this article appeared [20, 28], Gawrychowski and Nicholson [18] found the exact main term in the lower bound for these encodings, $n \lg k + n(k + 1) \lg(1 + 1/k)$, which is between $n \lg k + n/\ln 2$ and $n \lg k + \frac{k+1}{k} n/\ln 2$. This bound refines ours in the lower-order term, $O(n)$. They also build an encoding using optimal space up to lower-order terms. This encoding supports the queries, but not efficiently (i.e., it needs $\Omega(n)$ time), thus it is closer to a storage method than to a data structure with optimal query time. Their most recent version [17] contains an encoding using $1.5\, n \lg \kappa - \Theta(n)$ bits, which solves queries $\mathsf{top}(i, j, \kappa)$ and $\mathsf{sel}(i, j, \kappa)$, for κ fixed at construction, in time $O(\kappa^6 \lg^2 n\, \omega(1))$. This time is now sublinear, but still far from optimal.

The non-encoding version of the $\mathsf{sel}(\cdot)$ query problem has recently been studied intensively [7,9,15,16,23], always using linear space (i.e., $O(n \lg n)$ bits). Gagie et al. [15,16] solved the problem in $O(\lg n)$ time for any k, using a wavelet tree representation of A. Brodal and Jørgensen [8] reduced the time to $O(\lg n/\lg\lg n)$, with a structure similar to a multi-ary wavelet tree. Jørgensen and Larsen [23] obtained a query time of $O(\lg k/\lg\lg n + \lg\lg n)$, finally improved to $O(1 + \lg k/\lg\lg n)$ by Chan and Wilkinson [9].[1] These last two solutions build on an idea called *shallow cuttings* [25], which allows one to decompose the general problem into $O(n/k)$ carefully chosen problems of size $O(k)$, and then using Brodal and Jørgensen's structure [8] on those subproblems. We will also use shallow cuttings in our solutions.

Jørgensen and Larsen [23] introduced the *κ-capped* range selection problem, where a parameter κ is provided at preprocessing time, and the data structure only supports selection for ranks $1 \le k \le \kappa$ (as explained, interesting encodings can only solve this *κ-capped* version of the problem). They showed that even the one-sided κ-capped range selection problem requires query time $\Omega(\lg k/\lg\lg n)$ for structures using $O(n \operatorname{polylog} n)$ words; therefore the result of Chan and Wilkinson is the best possible for that space. This also shows that our faster results for one-sided queries are possible only because the structures only solve queries with $k = \kappa$.

It is worth noting that the data structures presented in this article are not merely a succinct implementation of the shallow cutting idea employed by Chan and Wilkinson [9] to obtain their

[1] Chan and Wilkinson claim a bound of $O(1+\lg_w k)$ for the "trans-dichotomous" model with word size $w = \Omega(\lg n)$. This is, however, based on an incorrect application of a result of Grossi et al. [21]; the proof presented in their paper [9] only yields a time bound of $O(1 + \lg k/\lg\lg n)$ (B. T. Wilkinson, personal communication).

optimal time. As their solution requires access to the array A at query time, we must address the simultaneous problems of reducing the space to asymptotically optimal, preserving optimal query time, and avoiding to access A during a query.

In the non-encoding model, the $\mathsf{top}(\cdot)$ query problem could be solved with our optimal-time $\mathsf{sel}(\cdot)$ solution at hand (see, e.g., Muthukrishnan [27]). We first obtain the kth value, v, and then use an RMQ data structure on A: We compute $p = \mathsf{rmq}(i, j)$, report it, and then continue recursively on the intervals $A[i, p-1]$ and $A[p+1, j]$, stopping as soon as we obtain values smaller than v. This takes the optimal $O(k)$ time. Note, however, that this idea cannot be directly used in the encoding model because the value v is not available and thus cannot be exploited as mentioned above. It also does not deliver the results in sorted order. Brodal et al. [6] gave linear-space data structures to retrieve the top-k results in order in time $O(k)$, even in online form where each new result is delivered in $O(1)$ time, without knowing k in advance. However, these data structures are not encodings as they require the explicit values of A.

Problem of independent interest. We single out a problem that could have other applications, and that arises as a subproblem in our encoding (see Section 5.1). Consider an array $Y[1, t]$ of t elements under a total order. Given a construction-time parameter ℓ, the purpose is to design an encoding to solve the following queries having any $1 \le j \le n$ and $1 \le d \le \ell$ as input (recall that we cannot access Y at query time).

1. Next-larger queries: $\mathsf{next\text{-}larger}(j, d)$ returns the position of the dth left-to-right value in $Y[j+1, t]$ that is strictly larger than $Y[j]$.
2. Previous-larger queries: $\mathsf{prev\text{-}larger}(j, d)$ returns the position of the dth right-to-left value in $Y[1, j-1]$ that is strictly larger than $Y[j]$.

The above queries return a special value 0 when the wanted position does not exist. In Section 5.1 we describe an encoding that answers queries in time $O(d)$, using $O(\ell\, t)$ bits of space.[2] This is mostly interesting for low values of ℓ, generalizing the existing structures that solve the case $\ell = 1$ [13]. Previous-smaller and next-smaller queries are obvious variants that can be solved similarly. In a conference version [20, Sec. 3.1] we showed how this encoding can be used to solve $\mathsf{top}(i, j, k)$ queries for any $1 \le k \le \kappa$, using $O(\kappa n)$ bits and $O(k^2)$ time, but this is subsumed in space and time by our better $\mathsf{top}(\cdot)$ solutions in this article.

Paper organization. The paper is organized as follows. In Section 2 we give an overview of the known succinct data strucures that we employ for our encodings. We present the lower bound on the space required by any encondings for our problem in Section 3. After that, we describe our general approach and relate it to the existing solutions based on shallow cuttings in Section 4, and give its succinct implementation in Section 5: in these sections, we pose a number of algorithmic challenges that are solved in Sections 6–8. Finally, we describe an encoding for the special case of one-sided queries in Section 9 and draw our conclusions in Section 10.

[2] Each query to our encoding can actually report all the d (left-to-right or right-to-left, respectively) values in time $O(d)$. The reason is that to answer, say, $\mathsf{next\text{-}larger}(j, d)$, we need to also answer incrementally $\mathsf{next\text{-}larger}(j, 1)$, ..., $\mathsf{next\text{-}larger}(j, d-1)$, taking overall $O(d)$ time.

2 Preliminaries

Our results make use of a number of popular succinct data structures, which we list below for the sake of completeness.

2.1 Bit-vectors

A bit-vector $B[1,n]$ is an array of n bits. We will be interested in solving two queries on it: $\mathsf{rank}_b(B,i)$ tells the number of occurrences of bit b in $B[1,i]$, and $\mathsf{select}_b(B,j)$ gives the position of the jth occurrence of bit b in B. We will use the following result:

Lemma 1 ([10]) *A bit-vector $B[1,n]$ can be stored in $n+o(n)$ bits (that is, $o(n)$ bits on top of B itself) so that queries* rank *and* select *are answered in $O(1)$ time.*

When the number m of 1s in $B[1,n]$ is small, the following result will be of interest as well:

Lemma 2 ([31]) *A bit-vector $B[1,n]$ with m 1s can be stored in $m\lg(n/m)+O(m)+o(n)$ bits, so that queries* rank *and* select *are answered in $O(1)$ time.*

Note that from this compressed representation we can still retrieve any $B[i] = \mathsf{rank}_1(B,i) - \mathsf{rank}_1(B,i-1)$ in constant time. If we aim at answering $\mathsf{rank}_1(B,i)$ only when $B[i]=1$, we can use less space, but now this is insufficient to recover the contents of B. The structure is called a monotone minimum perfect hash function (mmphf), as the rank values can be regarded as mapping elements in a universe $[1,n]$ to the domain $[1,m]$ while respecting the order:

Lemma 3 ([2]) *Given a bit-vector $B[1,n]$ with m 1s we can answer queries $\mathsf{rank}_1(B,i)$, whenever $B[i]=1$, in $O(1)$ time, using $O(m\lg\lg(n/m))$ bits, and without accessing B.*

2.2 Sequences

A sequence $S[1,n]$ over alphabet $[1,\sigma]$ requires $n\lg\sigma$ bits if represented in plain form. Within almost the same space, we can answer not only the basic query $\mathsf{access}(S,i)=S[i]$, but also the queries $\mathsf{rank}_c(S,i)$ and $\mathsf{select}_c(S,j)$ for any $c\in[1,\sigma]$, which are the natural extensions of the operations on bit-vectors:

Lemma 4 ([4, Thm. 6]) *A sequence $S[1,n]$ over alphabet $[1,\sigma]$ can be stored in $n\lg\sigma+o(n\lg\sigma)$ bits, so that* rank_c *queries are solved in time $O(1+\lg\lg_w\sigma)$,* select_c *queries are solved in time $O(1)$, and* access *queries are answered in any time complexity of the form $\omega(1)$.*

When the frequencies n_c of the symbols $c\in[1,\sigma]$ are skewed, it is possible to use space close to the zeroth-order entropy of S, $nH_0(S)=\sum_{1\le c\le\sigma} n_c\lg(n/n_c)\le n\lg\sigma$ bits, and still answer the queries. For this article, the most useful result of this kind is the following:

Lemma 5 ([4, Thm. 7]) *A sequence $S[1,n]$ over alphabet $[1,\sigma]$ can be stored in $nH_0(S)+o(n)$ bits, so that* rank_c, select_c, *and* access *queries are all solved in time $O(1+\lg_w\sigma)$.*

To obtain constant-time access and select simultaneously when $\lg\sigma=\omega(w)$, we can resort to an earlier version of Lemma 4, which uses slightly more space:

Lemma 6 ([19]) *A sequence $S[1,n]$ over alphabet $[1,\sigma]$ can be stored in $(1+\epsilon)n\lg\sigma + o(n\lg\sigma)$ bits, for any constant $\epsilon > 0$, so that rank_c queries are solved in time $O(\lg\lg\sigma)$, select_c queries are solved in time $O(1)$, and access queries are answered in constant time $O(1/\epsilon)$.*

Finally, the following result gives a structure to support a restricted form of rank_c queries in constant time, by resorting to mmphfs.

Lemma 7 ([3, Sec. 3]) *Given a sequence $S[1,n]$ over alphabet $[1,\sigma]$ we can answer queries $\mathsf{rank}_c(S,i)$, where $S[i] = c$, in $O(1)$ time, using $O(n\lg H_0(S)) = O(n\lg\lg\sigma)$ bits, and without accessing S.*

2.3 Parentheses and trees

A sequence $P[1,2n]$ of parentheses '(' (opening) and ')' (closing) is balanced if, read left to right, there are never more closing than opening parentheses, and in total there is the same number of both. There is an opening parenthesis $P[j]$ matching each closing parenthesis $P[i]$ (this is the maximum $j < i$ such that $P[j,i]$ is also balanced). Such j is found with operation $\mathsf{findopen}(P,i)$, which will be used in this article. Concretely, we use the following result:

Lemma 8 ([26]) *A balanced sequence of parentheses $P[1,2n]$ can be stored in $2n + o(n)$ bits (that is, $o(n)$ bits on top of P itself) so that queries $\mathsf{findopen}(\cdot)$ are answered in $O(1)$ time.*

It is also useful to interpret P as a bit-vector and add constant-time rank and select support, using $o(n)$ further bits (Lemma 1). The operations will be called $\mathsf{rank}_)$, $\mathsf{rank}_($, $\mathsf{select}_)$, and $\mathsf{select}_($.

A parenthesis sequence $P[1,2n]$ can be used to represent a general ordinal tree of n nodes, so that a large number of tree operations are supported in constant time. The next lemma lists those that will be used in this article:

Lemma 9 ([29]) *An ordinal tree of n nodes can be represented in $2n + o(n)$ bits, so that the following operations are supported in constant time, among others: compute the parent of a node v, compute the ith child of a node v, find the mth left-to-right leaf, compute the preorder of a node v and the node with preorder r, compute the depth of a node v, determine if a node v is an ancestor of another node u, compute the ancestor at any distance d of a node v, compute the subtree size of a node v, and find the internal node with inorder s (leaves not counted).*

We will use this lemma to represent binary trees where internal nodes always have two children. Then the left child of a node is the first and the right child is the second. Moreover, the inorder of an internal node is uniquely defined.

2.4 Predecessor queries

Given an increasing array $P[1,\kappa]$ of values in $[1,m]$, a predecessor query finds, given x, the maximum i with $P[i] \le x$. One can represent P as κ 1s on a bitvector $B[1,m]$, so that the predecessor of x is $\mathsf{select}_1(\mathsf{rank}_1(B,x))$. Using Lemma 2 to represent B, the space is $O(\kappa\lg(m/\kappa)) + o(m)$ bits and the time is constant. It is not possible, however, to have constant time without the $o(m)$-bits term [30]. In our article we will make heavy use of a structure called the *succinct SB-tree*:

Lemma 10 ([21, Lem. 3.3]) *If we have independent constant-time access to $P[1,\kappa]$, we can solve predecessor queries on P in time $O(1 + \lg\kappa/\lg\lg m)$ using $O(\kappa\lg\lg m)$ bits, plus a precomputed table of size $o(m)$ that depends only on m.*

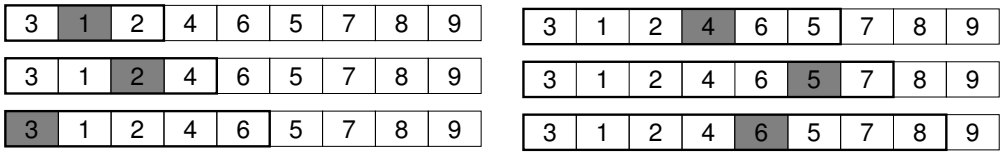

Figure 1: Illustration of our example of how the successive queries $\mathsf{sel}(1, j, 3)$ (in bold rectangles) spot the successive values of the permutations π_i (grayed cells). The six snapshots of the queries are shown in columwinse order.

Note that the $o(m)$ bits are still present, but they do not depend on P, thus we will have many succinct SB-trees and a single $o(m)$-bits table for all. Though better times, like $O(\lg \lg \kappa)$, can be obtained with structures that use $O(\kappa \lg m)$ bits [30], our results are not affected by the slower time of succinct SB-trees, whereas their lower space usage turns out to be fundamental.

3 Lower bounds

In this section we show that, given $A[1, n]$ and k, any encoding answering queries $\mathsf{sel}(1, j, k)$ or $\mathsf{top}(1, j, k)$ needs at least (essentially) $n \lg k$ bits. Note that these queries are weaker as they consider the first j positions of A rather than a range of its positions. The technique is to encode about n/k arbitrary permutations of $[k]$ in A, in a way that they can be retrieved with either of those queries. Thus the encodings cannot use less space that what is necessary to encode those arbitrary permutations, that is, rougly $n/k \times \lg_2 k! = \Omega(n \lg k)$ bits.

Assume for simplicity that $n = \ell k$, for some integer ℓ. Consider an array A of length n, initialized to $A[j] = j$, for $1 \leq j \leq n$, and then re-order its elements as follows: take $\ell - 1$ permutations π_i on $[k]$, $0 \leq i < \ell - 1$, and permute the elements in the subarray $A[ik + 1, (i + 1)k]$ according to permutation π_i, where $A[ik + j] = ik + \pi_i(j)$ for $0 \leq i < \ell - 1$ and $1 \leq j \leq k$. Note that the last k elements of A are not reordered, as they do not encode any π_i. Also, for $0 \leq i_1 < i_2 < \ell - 1$, the elements in the subarray for $i = i_1$ are all smaller than the elements in the subarray for $i = i_2$.

We now show how to reconstruct the $\ell - 1$ permutations by performing $\mathsf{sel}(1, j, k)$ queries on the array A. The main idea is easy to grasp with an example.

Example. Assume we have permutations $\pi_0 = (3\ 1\ 2)$ and $\pi_1 = (1\ 3\ 2)$ where $k = 3$. Figure 1 illustrates the process. Our array is $A[1, 9] = \langle 3, 1, 2, 4, 6, 5, 7, 8, 9 \rangle$, where π_0 is encoded in $A[1, 3]$ and π_1 in $A[4, 6]$ (with values shifted by $ik = 3$). Then, $\mathsf{sel}(1, 3, 3) = 2$ tells us that the minimum among the first 3 elements in π_0 (i.e. the 3rd largest element) is at $\pi_0(2)$, so $\pi_0(2) = 1$. Next, $\mathsf{sel}(1, 4, 3) = 3$ tells us that the second minimum (2nd largest element) in π_0 is at $\pi_0(3)$, so $\pi_0(3) = 2$, and thus $\pi_0(1) = 3$. This is because $A[1, 4]$ contains $A[4]$, which must be larger than all $A[1, 3]$, and thus the 3rd largest element in $A[1, 4]$ must be the 2nd largest element in $A[1, 3]$. With $\mathsf{sel}(1, 6, 3) = 4$ we discover that the 3rd element in π_1 is at $\pi_1(1)$, so $\pi_1(1) = 1$, and so on. □

Now we formalize the process described in the example.

Lemma 11 *The position of the kth largest value in the prefix $A[1, ik + j - 1]$ is the position of value $(i - 1)k + j$, for any $1 \leq i < \ell$ and $1 \leq j \leq k$.*

Proof. Since the values of A were initially increasing and then we locally permuted the blocks of length k, it holds that, for each $1 \leq i < \ell$, $A[x] < A[y]$ for any $x \leq ik$ and $y > ik$. Then the values

in $A[ik+1, ik+j-1]$ are the largest of $A[1, ik+j-1]$, and the values in $A[(i-1)k+1, ik]$ are the largest of $A[1, ik]$. Thus, the kth largest value in $A[1, ik+j-1]$ is the $(k-j+1)$th largest value in $A[1, ik]$. This value is also the $(k-j+1)$th largest value in $A[(i-1)k+1, ik]$, or which is the same, the jth smallest value in $A[(i-1)k+1, ik]$. Thus, by the definition of $A[(i-1)k+1, ik]$, it is the value $(i-1)k+j$. □

Therefore, $\mathsf{sel}(1, ik+j-1, k)$, the position of the kth value in the prefix $A[1, ik+j-1]$, is equal to $(i-1)k + \pi_{i-1}^{-1}(j)$, which is the position of value $(i-1)k+j$. Then, any π_{i-1} can be easily computed with the $k-1$ queries $\mathsf{sel}(1, ik+j-1, k)$ for $1 \leq j \leq k-1$.

Since representing $\ell - 1$ arbitrary permutations on $[k]$ requires $\lg((k!)^{\ell-1}) = (\ell-1)\lg k! = (n/k-1)(k\lg k - O(k)) = (n-k)\lg k - O(n-k) = n\lg k - O(n + k\lg k)$ bits, any encoding able to answer all queries $\mathsf{sel}(1, j, k)$ on A needs also this number of bits.

The proof applies to $\mathsf{top}(1, j, k)$ as well, since we can reconstruct the value $\mathsf{sel}(1, ik+j-1, k)$ from $\mathsf{top}(\cdot)$ queries: $\mathsf{sel}(1, ik+j-1, k)$ is the only element that disappears from the answer set when we move from $\mathsf{top}(1, ik+j-1, k)$ to $\mathsf{top}(1, ik+j, k)$. As we move, the element $A[ik+j]$ enters in the answer and the element that was the smallest (i.e., the kth), which belongs to $A[(i-1)k+1, ik]$, leaves the answer set.

Theorem 1 *Any encoding of an array $A[1, n]$ answering $\mathsf{sel}(\cdot)$ or $\mathsf{top}(\cdot)$ queries, even if restricted to ranges $A[1, j]$ and for a fixed k value, requires at least $n\lg k - O(n + k\lg k)$ bits of space.*

4 General approach

We describe Jørgensen and Larsen's "shallow cuttings" idea [23], and the way Chan and Wilkinson [9] take advantage of it. In general terms, our encoding for $\mathsf{sel}(\cdot)$ queries will implement their solution in an encoding scenario. This poses, however, a number of challenges that will be dealt with in the subsequent sections; the plan is described at the end of this section. Table 1 gives the notation used throughout the article.

4.1 Shallow cuttings

Let $A[1, n]$ be a permutation on $[n]$. Consider each entry $A[i]$ as a point $(x, y) = (i, A[i])$, and set a parameter κ. A horizontal line sweeps the grid space $[1, n] \times [1, n]$ from $y = n$ (top) to $y = 1$ (bottom). The points hit are included in a single *root cell*, which spans a three-sided area called a *slab*, of the form $[1, n] \times [y, n]$, which includes all the points of the cell. Once we reach a point (x^*, y^*) that makes the root cell contain 2κ points, we *close* the cell and leave its slab with its definitive area $[1, n] \times [y^*, n]$.

Let x_{split} be the κth smallest x-coordinate in the above root cell. This is called the *split point*. The sweeping process is repeated recursively on each of the two grid spaces $[1, x_{\text{split}}] \times [1, n]$ and $[x_{\text{split}} + 1, n] \times [1, n]$. This will create two *children cells* as follows. They will contain the topmost points whose x-coordinates are $\leq x_{\text{split}}$ and $> x_{\text{split}}$, respectively. Their slabs will grow downwards as we continue with the sweeping process, independently for each cell. When those cells, in turn, reach size 2κ, we close them, find their split points, and continue the recursion on the resulting grid spaces. The recursive process terminates on a *final cell* when less than 2κ points are left in the current grid space.

Variable	Meaning
A	Array where we perform $\mathsf{sel}(\cdot)$ or $\mathsf{top}(\cdot)$ queries.
n	Number of elements in A.
k	Argument of a particular $\mathsf{sel}(\cdot)$ or $\mathsf{top}(\cdot)$ query.
κ	Maximum k value allowed from construction.
T_C	The binary tree of cells induced by shallow cutting.
t	Number of internal nodes in T_C, it has $t+1$ leaves, and $2t+1$ nodes in total.
x_i	The t final split points induced by shallow cutting.
y_i	The value of A associated with split point x_i by shallow cutting.
A_v	Array of the $O(\kappa)$ y-coordinates of the points (i.e., values of A) in the extent of node v, mapped to $[1, O(\kappa)]$ respecting relative order.
A_i	Array similar to A_v associated with the special extent of split point x_i.
E_v	Array of the $O(\kappa)$ positions where the elements of A_v appear in A.
P_v	Central range of E_v that refers only to the points in the slab of v.
v_-, v_+	Nodes preceding and following v when its extent is defined.
z	Number of levels of marked nodes in the solution to access P_v.
t_ℓ	Used to define the level ℓ of a marked node, $t_\ell^2 \leq \lvert v\rvert < t_{\ell-1}^2$.
M	Bit-vector that indicates which nodes of T_C are marked, in preorder.
L	Sequence giving the levels of the marked nodes (1s in M).
o_v	Bit-vector of 2κ bits storing which points of v are original.
r_v	Array of κ entries storing the ranks of each original point of v at the node v' that leaves the path of unmarked nodes where v belongs and inherits the point.
b_v	Bit-vector that concatenates the distances, in unary, from each original point of v to the node v' described in the previous line.
o, r, b	Arrays created by concatenating o_v, r_v, and b_v in preorder for the unmarked nodes.
π	The path of unmarked/unsampled nodes where v belongs. All nodes have the same level ℓ.
u	Parent of the topmost node in the path π.
u'	The only node of level ℓ leaving π (at the bottom; the others have level $> \ell$).
c_v	Sequence of colors assigned to unmarked node v to represent inherited points.
c_π	Concatenation of sequences c_v along the path π of unmarked nodes.
c'_π, c^g_π	Actual representation of sequence c_π, as a string and a bitvector per color.
B, R	Bit-vectors used to find c_π for any node $v \in \pi$.
p_v	Position in P_v of the first point inherited in $P_{u'}$.
h_v	Bit-vector that indicates which of the points in $P_{u'}$ are inherited from P_v.
o_π	The o_v values, now stored contiguously along path π.

Table 1: Notation.

A binary tree T_C is created to reflect the cell refinement process (see Figure 2). The root cell is associated with the root node of T_C, the first two children cells to the left ($[1, x_{\text{split}}]$) and right ($[x_{\text{split}} + 1, n]$) children of the root, and so on. The leaves of T_C are associated with the final cells, which have not been split and contain κ to $2\kappa - 1$ points (unless $n < \kappa$, in which case only a root cell exists).

At any moment of the sweeping process, we have a sequence of points $x_1 < x_2 < \ldots$ that have been chosen as split points; new points are inserted anywhere in the sequence as further cells are split. These points delimit the x-coordinate slab ranges of which are the leaves of T_C at the current moment of the sweep. When the next split occurs, say within the slab covering interval $[x_i + 1, x_{i+1}]$, we obtain two new cells, whose slabs cover the x-coordinate intervals $[x_i + 1, x_{\text{split}}]$ and $[x_{\text{split}} + 1, x_{i+1}]$. We associate the *keys* $[x_i + 1, x_{\text{split}}]$ and $[x_{\text{split}} + 1, x_{i+1}]$ and the *extents* $[x_{i-1} + 1, x_{i+1}]$ and $[x_i + 1, x_{i+2}]$, respectively, with the two new cells (assume further split points 0 and n in the extremes).

When the sweep finishes, T_C has t internal nodes and $t + 1$ leaves, and there are $t + 2$ split points $0 = x_0 < x_1 < x_2 < \ldots < x_t < x_{t+1} = n$ (writing 0 and n explicitly), which delimit the slabs of the final leaves of T_C. In the following, we will use x_i to refer to these final split points. In addition to the extents associated with cells, we associate the *special* extents $[x_{i-1} + 1, x_{i+1}]$ with the split points x_i, for $1 \le i \le t$. The root of T_C has key and extent $[1, n]$. Note that, since leaves contain successive positions of A, it holds $\kappa \le x_{i+1} - x_i < 2\kappa$ for all i (if $n \ge \kappa$).

Example. Figure 2 gives an example (values y_i will be defined soon). Note that the child slabs inherit half of the points of their parent slab. □

This construction has a number of key properties [23]:

1. It creates $O(t) = O(n/\kappa)$ cells, each containing κ to 2κ points (if $n \ge \kappa$).
2. If c is the cell of the highest (closest to the root) node $v \in T_C$ whose key is contained in a query range $A[i, j]$, then $[i, j]$ is contained in the extent of c.
3. The top-κ values in $A[i, j]$ belong to the union of the 3 cells comprising the extent of c (these contain at most 6κ points).

4.2 Optimal-time select queries

Using the properties of shallow cuttings, Chan and Wilkinson [9] reduce the $O(\lg n / \lg \lg n)$ time of Brodal and Jørgensen [8] as follows. At each node $v \in T_C$, they store the structure of Brodal and Jørgensen for the array $A_v[1, O(\kappa)]$ of the y-coordinates of the points in the extent of v. Actually, they store in A_v the local permutation in $[O(\kappa)]$ induced by the relative ordering in A, thus A_v requires $O(\kappa \lg \kappa)$ bits in each v and $O(n \lg \kappa)$ bits in total. The structure for range selection also uses $O(\kappa \lg \kappa)$ bits and answers queries in time $O(1 + \lg \kappa / \lg \lg n)$.[3] They also store an array $E_v[1, O(\kappa)]$, so that $E_v[i]$ is the position in $A[1, n]$ of the value stored in $A_v[i]$. For the special extents associated with split points x_i, they also store structures A_i analogous to the structures A_v.[4] The structures A_i add up to $O(n \lg \kappa)$ bits, since they are built on sub-arrays of length up to 4κ whose contents are mapped to the range $[1, O(\kappa)]$.

[3] One could expect time $O(1+\lg \kappa / \lg \lg \kappa)$, but the denominator may stay at $\lg \lg n$ by the use of global precomputed tables of total size $o(n)$.

[4] Arrays E_i are not necessary because the special extents refer to contiguous ranges in A.

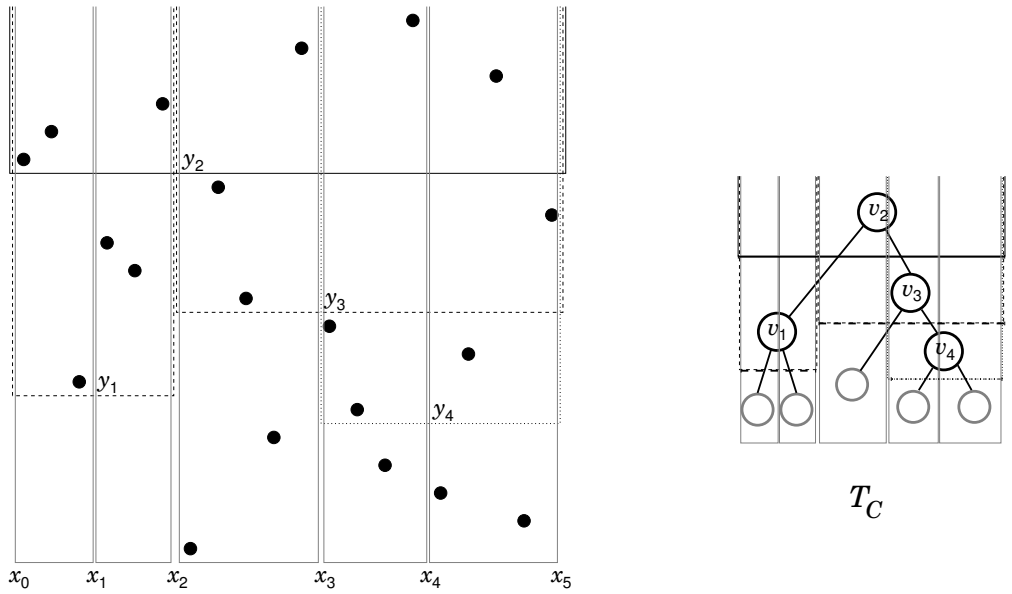

Figure 2: An example of the shallow cutting process with $\kappa = 3$. On the left, the points are swept top to bottom. First, the root cell is closed when x_0 is found (the root slab is shown with a black solid line). The splitting point is x_2. Then its two children (slabs in black dashed lines) are formed. A third slab in black dotted lines is the right child of the right child of the root. The grayed lines show the slabs of the child cells. We show the split points x_i and their associated y^* values, y_i. On the right, the induced binary tree T_C is shown, with leaves in gray. The keys of the internal nodes are the horzontal intervals shown in bold lines, and their identifiers are v_i, where i is their inorder number.

Property 3 of shallow cuttings implies that the kth largest element of $A[i,j]$, for any $k \leq \kappa$, is also the kth largest value in $A_v[l,r]$, where v is the node that corresponds to interval $A[i,j]$ by property 2 and $E_v[l-1] < i \leq j < E_v[r+1]$ are the elements in the extent of node v enclosing $[i,j]$ most tightly. Thus query $\mathsf{sel}(i,j,k)$ on A is mapped to query $p = \mathsf{sel}(l,r,k)$ on A_v. Once the local answer is found in $A_v[o]$, the global answer is $E_v[o]$.

Summing up, the main ingredients are based on the funtionalities of tree T_C, and arrays E_v, A_v and A_i. Chan and Wilkinson [9] manage to store them in $O(n(\lg\kappa + \lg\lg n + (\lg n)/\kappa))$ bits, which gives $O(n \lg n)$ bits when added over a set of suitable κ values (their structure works for every $1 \leq k \leq n$, so several κ-capped structures are built). Also, their solution requires to access A and thus does not immediately translate into our setting.

4.3 Encodings for optimal-time select queries

Our general plan is to derive an encoding from the strategy of Chan and Wilkinson, which retains the optimal time for $\mathsf{sel}(\cdot)$ but reduces the space to $O(n \lg\kappa)$ and does not access A. This requires addressing several challenges.

1. In Section 5 we design a succinct representation of T_C that is able to find the node v given the interval $A[i,j]$, so that from v we gain access to the data associated with node v in constant time. This structure uses $O((n/\kappa)\lg\kappa) + o(n)$ bits. Associated with each node v we will store

Chan and Wilkinson's structures A_v for range selection (whose space is $O(\kappa \lg \kappa)$ bits and thus can be afforded), and a data structure that simulates array E_v (as its direct representation cannot be afforded). We will also store the structures A_i associated with the split points x_i.

2. In Section 6 we provide constant-time access to any E_v using $O(n \lg \kappa)$ bits. Together with the previous result, this already yields an $O(\lg \kappa)$ time algorithm for $\mathsf{sel}(\cdot)$ queries, as we can first find the node v in constant time, then do a binary search for l and r in E_v, then run the range selection query on $A_v[l, r]$ in time $O(1 + \lg \kappa / \lg \lg n)$, and finally return $E_v[o]$ in $O(1)$ time. Our representation of E_v uses a hierarchical marking of nodes plus a color-based encoding of the inheritance of points along cells in paths of unmarked nodes in T_C.

3. In Section 7 we address the bottleneck of the previous solution: we replace the binary search by fast predecessor queries on E_v, so as to obtain $O(1+\lg \kappa / \lg \lg n)$ time. This is obtained by storing succinct SB-trees [21] on some sampled nodes (which include at least all the marked nodes), and searches on the inheritance information along paths of unsampled nodes, using global precomputed tables.

4. In Section 8 we wrap up the results in order to prove Theorem 2. Then we show how to answer top-k queries by using an existing linear-space technique [6] on a reduced universe. This proves Theorem 3.

Theorem 2 *Given an array $A[1, n]$ and a value κ, there is an encoding of A that uses $O(n \lg \kappa)$ bits and supports the query $\mathsf{sel}(i, j, k)$ in time $O(1 + \lg k / \lg \lg n)$ for any $k \leq \kappa$.*

Theorem 3 *Given an array $A[1, n]$ and a value κ, there is an encoding of A that uses $O(n \lg \kappa)$ bits and supports the query $\mathsf{top}(i, j, k)$ in time $O(k)$, for any $k \leq \kappa$.*

5 Shallow cuttings in succinct space

In this section we show how to represent the shallow cutting structure using $O((n/\kappa) \lg \kappa) + o(n)$ bits so that, given the query interval $[i, j]$, we obtain the corresponding node $v \in T_C$ according to property 2 of shallow cuttings, and then give access to the structures associated with node v. We will also need to find, given v, the two "neighbor" nodes v_- and v_+ that define the extent of v, and map between nodes and their keys in both directions.

Finding the maximal range of split points. Our first structure is a bit-vector $S[0, n]$ that marks the split points x_i, that is, $S[x_i] = \mathsf{1}$ for all $0 \leq i \leq t+1$ and $S[j] = \mathsf{0}$ elsewhere. Since S has only $t + 2$ bits set out of n, we can represent it in compressed form (Lemma 2) so that it requires $t \lg(n/t) + O(t) + o(n) = O((n/\kappa) \lg \kappa) + o(n)$ bits of space and supports operations $\mathsf{rank}_b(S, i)$ and $\mathsf{select}_b(S, j)$ in constant time.

With this representation of S we find in constant time the range $[m, M]$ of split points contained in $A[i, j]$. More precisely, we find the largest range $[m, M]$ such that $i \leq x_m < \ldots < x_M \leq j$, in constant time with $m = \mathsf{rank}_{\mathsf{1}}(S, i - 1) + 1$ and $M = \mathsf{rank}_{\mathsf{1}}(S, j)$. Note that the range $[m, M]$ can contain zero split points in some cases. We have the following result:

Lemma 12 *Given the range $[i, j]$, we can find in $O(1)$ time the maximal range $[m, M]$ of split points (if any) contained in $[i, j]$ with a structure that uses $O((n/\kappa) \lg \kappa) + o(n)$ bits of space.*

If the range $[m, M]$ contains zero or one split points (i.e., $m \geq M$), then $[i, j]$ does not contain a complete cell:[5] either $[i, j]$ is fully contained in the range of the mth left-to-right leaf of T_C (and contains no split points) or $[i, j]$ starts in the range of the mth leaf and ends in that of the $(m+1)$th leaf of T_C (and contains one split point). In both situations, the range $[i, j]$ is contained in the special extent of the mth split point of T_C, $[x_{m-1} + 1, x_{m+1}]$, recalling that x_m is found using the bit-vector S. In this simple case, we compute the offset $o = \mathsf{select}_1(S, m-1)$, perform the query $\mathsf{sel}(i-o, j-o, k)$ on the structure A_m associated with split point x_m, and remap the answer to the global position by just adding o.

In the sequel we consider the more complex case of two or more split points, that is, $m < M$.

Finding the key of the node v for a range $A[i, j]$. If $m < M$, the following procedure finds the desired key [23]. Find, within $x_m, \ldots, x_M$, the split point x_r with maximum associated y_r-coordinate (this is the y^* coordinate given to the slab of the cell that was closed when x_r was chosen as a split point). Find the split point x_s with the second maximum. If $s < r$ (i.e., x_s is to the left of x_r), then the key of the desired node v is $[x_s + 1, x_r]$, otherwise it is $[x_r + 1, x_s]$.

To find the first and second maxima, let the array $Y[1, t] = y_1, \ldots, y_t$ contain the y^* values associated with the split points $x_1, \ldots, x_t$. We do not represent Y itself, but rather store a range top-2 encoding of it [11]. This structure requires $O(t) = O(n/\kappa)$ bits and returns the positions of the first and second maxima in $Y[m, M]$, x_r and x_s, in $O(1)$ time.[6] We have the following result:

Lemma 13 *Given split points $x_m < x_M$, we can find in $O(1)$ time the maximal key of a node $v \in T_C$ that is contained in $[x_m + 1, x_M]$, with a structure that uses $O(n/\kappa)$ bits of space.*

Example. See Figure 2 again, and consider a range $A[i, j]$ that contains x_1 to x_4. Then $r = 2$ and $s = 3$, and the key is $[x_2 + 1, x_3]$, because $y_2 = \max\{y_1, y_2, y_3, y_4\}$ and y_3 is the second maximum. Instead, if $A[i, j]$ contains x_3 to x_4, then $r = 3$ and $s = 4$ because $y_3 > y_4$. □

Finding the extent of v. Assume w.l.o.g. that $r < s$ and thus the desired key is $[x_r + 1, x_s]$; the case $[x_s + 1, x_r]$ is symmetric. To compute the extent of this key we need to find the split points that, at the moment when the key $[x_r + 1, x_s]$ was created during the sweep, preceded x_r and followed x_s. Let us call these split points $x_{r'}$ and $x_{s'}$, respectively. Here we use the encoding for prev-larger and next-larger queries described at the end of Section 1.

At the time we created the split point x_s, the split points that existed were precisely those with y^* value larger than that associated with x_s. Thus, since $x_r < x_s$, the split point that followed x_s is $x_{s'}$, with $s' = \mathsf{next\text{-}larger}(s, 1)$, the leftmost value in $Y[s+1, t+1]$ that is larger than $Y[s]$ (assume $Y[t+1] = n+1$ so this is always defined). Similarly, since all the values in $Y[r+1, s-1]$ are smaller than $Y[s]$, and $Y[r] > Y[s]$, the split point that preceded x_r when x_s was created was $x_{r'}$, with $r' = \mathsf{prev\text{-}larger}(s, 2)$, the second rightmost value in $Y[0, s-1]$ that is larger than $Y[s]$ (assume $Y[0] = n+1$ so this is always defined).[7] In Section 5.1 we show how to support prev-larger and next-larger queries in constant time using $O(t) = O(n/\kappa)$ bits of space. Then the extent is $[x_{r'} + 1, x_{s'}]$.

[5] In some border cases it can, but these are still correctly handled as indicated here.

[6] Note that it is not a matter of obtaining $r = \mathsf{rmq}(m, M)$ and then choosing s from $s_1 = \mathsf{rmq}(m, r-1)$ and $s_2 = \mathsf{rmq}(r+1, M)$, since we have no way to compare $Y[s_1]$ with $Y[s_2]$ if we do not store Y.

[7] Note that $\mathsf{prev\text{-}larger}(s, 2)$ is not necessarily $\mathsf{prev\text{-}larger}(\mathsf{prev\text{-}larger}(s, 1), 1) = \mathsf{prev\text{-}larger}(r, 1)$ as there might be an element x to the left of $Y[r]$ such that $Y[r] > x > Y[s]$.

Lemma 14 *Given the key of a node $v \in T_C$, and knowing which of its extremes has a lower y value, we can obtain the extent of v in $O(1)$ time with a structure that uses $O(n/\kappa)$ bits of space.*

Example. In Figure 2, for the key $[x_2 + 1, x_3]$, we find the extent $[x_0 + 1, x_5]$, whereas for the key $[x_3 + 1, x_4]$, the extent is $[x_2 + 1, x_5]$. In both cases, the extent contains the range $A[i, j]$. □

Finding the node with a given key. We have obtained the key of v, but not yet v. Similarly, we have obtained its extent, but not its corresponding neighboring nodes v_- and v_+. The structure A_v contains the data corresponding to the extent of v, but we will also need to refer to its neighboring nodes in order to decode the results obtained in A_v.

To reference the nodes, we will represent the topology of T_C, which has $2t + 1$ nodes, with the succinct tree representation of Lemma 9. It uses $4t + 2 = O(n/\kappa)$ bits of space and supports all the operations we need, in constant time.

If the key of node v is $[x_r + 1, x_s]$ and its extent is $[x_{r'} + 1, x_{s'}]$, then the neighbor nodes of v will be those with keys $[x_{r'} + 1, x_r]$ and $[x_s + 1, x_{s'}]$. In general, we will need to find the nodes corresponding to arbitrary keys.

Given a key $[x_r + 1, x_s]$, where $Y[r] > Y[s]$, we can compute the corresponding node $v \in T_C$ as follows. Since this key was created with the split point x_s, the corresponding node of T_C is the left child of the s-th node of T_C in inorder [23]. This node with inorder s is computed in constant time in our representation (Lemma 9), and then we can also compute its left child in constant time. If, instead, the key is $[x_s + 1, x_r]$ (still with $Y[r] > Y[s]$), then v is the right child of the sth node in inorder.

Example. Again in Figure 2, the key $[x_2 + 1, x_3]$ holds $y_2 > y_3$, thus we take the internal node of T_C with inorder 3 (that is, v_3), and the desired node is its left child (that is, the third left-to-right leaf). Consider instead the key $[x_3, x_5]$. Since $y_3 < y_5 = n + 1$, we take the internal node with inorder 3 (v_3 again) and the answer is its right child, that is, v_4. □

If the key is given but we do not know which is smaller between $Y[r]$ and $Y[s]$, we find the rth inorder node u_r, the sth inorder node u_s, and compare their depths in T_C; the deeper one corresponds to the smallest value.[8] This is also useful to compute the extent of the resulting node, since the procedure we have given needs to know which of the two endpoints has a lower y^* value.

Lemma 15 *Given the key of a node $v \in T_C$, we can find the node v itself, its extent, and its neighbor nodes v_- and v_+, in $O(1)$ time with a structure that uses $O(n/\kappa)$ bits of space.*

Finding the key of a given node. Conversely, let v be a node and assume we want to find its key $[x_r + 1, x_s]$. If v is the root, its key is $[x_0 + 1, x_n]$. Otherwise, we compute the parent node u of v in T_C, the inorder rank $i(u)$ of u, and the subtree size of v, $|v|$ (which is always odd since T_C is binary). Then, if v is the left child of u, we have $r = i(u) - (|v| + 1)/2$ and $s = i(u)$. If v is a right child, then $r = i(u)$ and $s = i(u) + (|v| + 1)/2$. Recall that we can obtain the value $x_i = \mathsf{select}_1(S, i)$ of the ith split point, for any i. The following lemma considers the space of the succinct representation of T_C and the bitvector S.

[8] This only works if $[x_r + 1, x_s]$ is a key; it cannot be used for the top-2 problem we had mentioned.

Lemma 16 *Given a node $v \in T_C$, we can find its corresponding key in $O(1)$ time with a structure that uses $O((n/\kappa)\lg\kappa) + o(n)$ bits of space.*

Example. Consider the node v_3 in Figure 2. Its parent is v_2. Since v_3 is a right child, the inorder of v_2 is 2, and $|v_3| = 5$, we have $r = 2$ and $s = 2+(5+1)/2 = 5$. That is, the key of v_3 is $[x_2+1, x_5]$. Now consider the third left-to-right leaf. Its parent is v_3 (with inorder 3), the leaf is a left child and its subtree size is 1. So we compute $r = 3-(1+1)/2 = 2$ and $s = 3$, thus the key is $[x_2+1, x_3]$. □

Associating structures with nodes. Once we have identified a node v, the succinct representation of T_C yields its preorder rank $p(v)$ in constant time (Lemma 9). This is used to associate any desired data structure (such as A_v, for example) with the $p(v)$th entry of an array.

5.1 Computing next-larger and prev-larger queries

We now show how to compute values next-larger(j, d) and prev-larger(j, d) for any $1 \le j \le t$ and $1 \le d \le \ell$, for some parameter ℓ given at construction time (see Section 1 for the definition of these queries). Our data structure will answer those queries in $O(d)$ time, using $O(\ell t)$ bits of space. For our needs, constant $\ell = 2$ is sufficient, so the time is $O(1)$ and the space is $O(n/\kappa)$ bits.

We will describe the structure to support prev-larger queries for an array $Y[1,t]$; the one for next-larger is analogous. We define, for each element $Y[j]$, ℓ *pointers*, $D_1[j] \ldots D_\ell[j]$, to the ℓ rightmost elements larger than $Y[j]$ that are in $Y[1, j-1]$.

Definition 1 *Given an array $Y[1,t]$, we define arrays of pointers $D_0[1,t]$ to $D_\ell[1,t]$ as follows: $D_0[j] = j$, and $D_d[j] = \max(\{i < D_{d-1}[j] : Y[i] > Y[j]\} \cup \{0\})$, for $d > 0$.*

We now prove a result that is essential for the space-efficient representation of all D_d arrays, so that we can compute any prev-larger$(j, d) = D_d[j]$ in time $O(d)$. The following lemma shows that if we draw, for a given d, all the arcs starting at $D_{d-1}[j]$ and ending at $D_d[j]$ for all j, then no arcs "cross".

Lemma 17 *Let $j_1, j_2 \in [1, n]$ and $0 < d \le \ell$, and let us call $i_1 = D_{d-1}[j_1]$ and $i_2 = D_{d-1}[j_2]$. Then, if $i_1 < i_2$ and $D_d[j_2] < i_1$, it holds $D_d[j_1] \ge D_d[j_2]$.*

Proof. It must hold $Y[i_1] < Y[i_2]$, since otherwise $D_d[j_2] \ge i_1$ by Definition 1 (as it would hold $Y[j_2] < Y[i_2] \le Y[i_1]$ and $0 < i_1 < i_2$), contradicting the hypothesis.

Now let us call $r_1 = D_d[j_1]$ and $r_2 = D_d[j_2] < i_1$. Then we have (1) $Y[r_2] > Y[j_2]$, because $r_2 = D_d[j_2]$; (2) $Y[j_2] \ge Y[i_1]$, because otherwise it would be $r_2 = D_d[j_2] \ge i_1$, as implied by Definition 1 (since $i_1 < i_2 = D_{d-1}[j_2]$ and $Y[i_1] > Y[j_2]$, and $r_2 \ge i_1$ contradicts the hypothesis); and (3) $Y[i_1] > Y[j_1]$, because $i_1 = D_{d-1}[j_1]$. Therefore, $Y[j_1] < Y[r_2]$, and then $r_1 = D_d[j_1] \ge r_2$, as implied by Definition 1 since $r_2 = D_d[j_2] < i_1 = D_{d-1}[j_1]$. □

Example. Figure 3 illustrates the lemma. The solid arcs cannot cross in the x coordinate. □

This property enables a space-efficient implementation of the pointers. We set bit-vector $T_0 = 1(10)^t$ to represent $D_0[j] = j$. We represent each "level" $d > 0$ of pointers separately, as a set of arcs

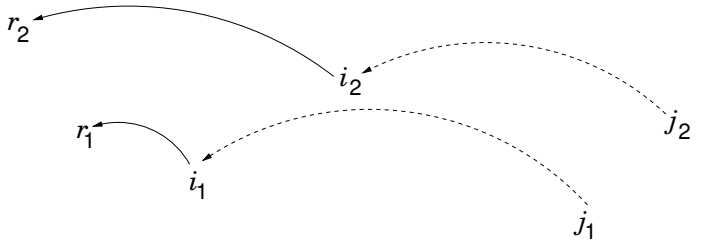

Figure 3: Illustration of Lemma 17, where $r_1 = D_d[j_1]$ and $r_2 = D_d[j_2]$. The heights of the elements represent their y value (higher is larger). The dashed arrows represent D_{d-1} and the solid arrows the step between D_{d-1} and D_d.

leading from $D_{d-1}[j]$ to $D_d[j]$. For a level $d > 0$ and for any $0 \leq i \leq t$, let $p_d[i] = |\{j,\ D_d[j] = i\}|$ be the number of pointers of level d that point to position i. We then store a bit-vector

$$T_d[1, 2t+1] \quad = \quad 1\mathsf{0}^{p_d[0]}\, 1\mathsf{0}^{p_d[1]}\, 1\mathsf{0}^{p_d[2]} \ldots 1\mathsf{0}^{p_d[t]},$$

where we mark the number of times each position is the target of pointers from level d. Each $\mathsf{1}$ corresponds to a new position and each $\mathsf{0}$ to the target of an arc. Note that the sources of those arcs correspond to the $\mathsf{0}$s in bit-vector T_{d-1}, that is, to arcs that go from $D_{d-2}[j]$ to $D_{d-1}[j]$. Arcs that enter the same position i are sorted according to their source position, so that we associate the leftmost $\mathsf{0}$s of $\mathsf{0}^{p_d[i]}$ with the arcs with the rightmost sources. Conversely, we associate the rightmost $\mathsf{0}$s of $\mathsf{0}^{p_{d-1}[i]}$ with the arcs with the leftmost targets. This rule ensures that those arcs entering, or leaving from, the same position do not cross in T_d (as implied by Lemma 17). The matching between sources and targets is represented with a balanced sequence of parentheses (Lemma 8)

$$B_d[1, 2t] \quad = \quad (^{p_{d-1}[0]})^{p_{d-1}[0]} (^{p_d[0]-p_{d-1}[0]})^{p_{d-1}[1]} (^{p_d[1]})^{p_{d-1}[2]} (^{p_d[2]} \ldots)^{p_{d-1}[t]}.$$

This sequence matches arc targets (opening parentheses) and sources (their corresponding closing parentheses). For example, take $T_1 = \mathtt{10100100101101100 1011}$, $T_2 = \mathtt{10001001001011011 0111}$, and $B_2 = ()(())(())(()()())()$ in Figure 4 (right column, on the center). Each $\mathsf{0}$ in T_2 represents a target corresponding to a parenthesis '(' in B_2, and it matches the $\mathsf{0}$ in T_1 that is the corresponding source represented by the companion ')' in B_2: reading B_2 from left to right, the first $\mathsf{0}$ in T_2 is matched with the first $\mathsf{0}$ in T_1; the next two $\mathsf{0}$s in T_2 are matched with the next two $\mathsf{0}$s in T_1 by their nested pairs of parentheses, and so on. Here the enclosing pair of parentheses in $(()()())$ from B_2 matches the sixth $\mathsf{0}$ in T_2 with the ninth $\mathsf{0}$ in T_1 (see the corresponding arc $(7, 2)$ in Figure 4 and observe that there are $7 + 1$ preceding $\mathsf{1}$s in T_1 and $2 + 1$ preceding $\mathsf{1}$s in T_2).

In general we write $p_d[i]$ parentheses '(' before the $p_{d-1}[i+1]$ parentheses ')', so the targets in position i of T_d can match all the corresponding sources that are to the right of position i in T_{d-1}. Since the first $p_{d-1}[0]$ sources in T_{d-1} are special as they have targets at position 0 in T_d (i.e. they induce the only self-loops), we start preceding these sources with $p_{d-1}[0]$ targets from the $p_d[0]$ ones in T_d, and the remaining $p_d[0] - p_{d-1}[0]$ targets are written after these sources in B_d.

The tracking from $d = 0$ to $d = \ell$ will proceed by computing values z_d, so that the value of interest corresponds to the z_dth $\mathsf{0}$ in T_d. The following formula computes z_d from z_{d-1}:

$$z_d \quad = \quad \mathsf{findopen}(B_d, \mathsf{select}_)(B_d, z_{d-1})).$$

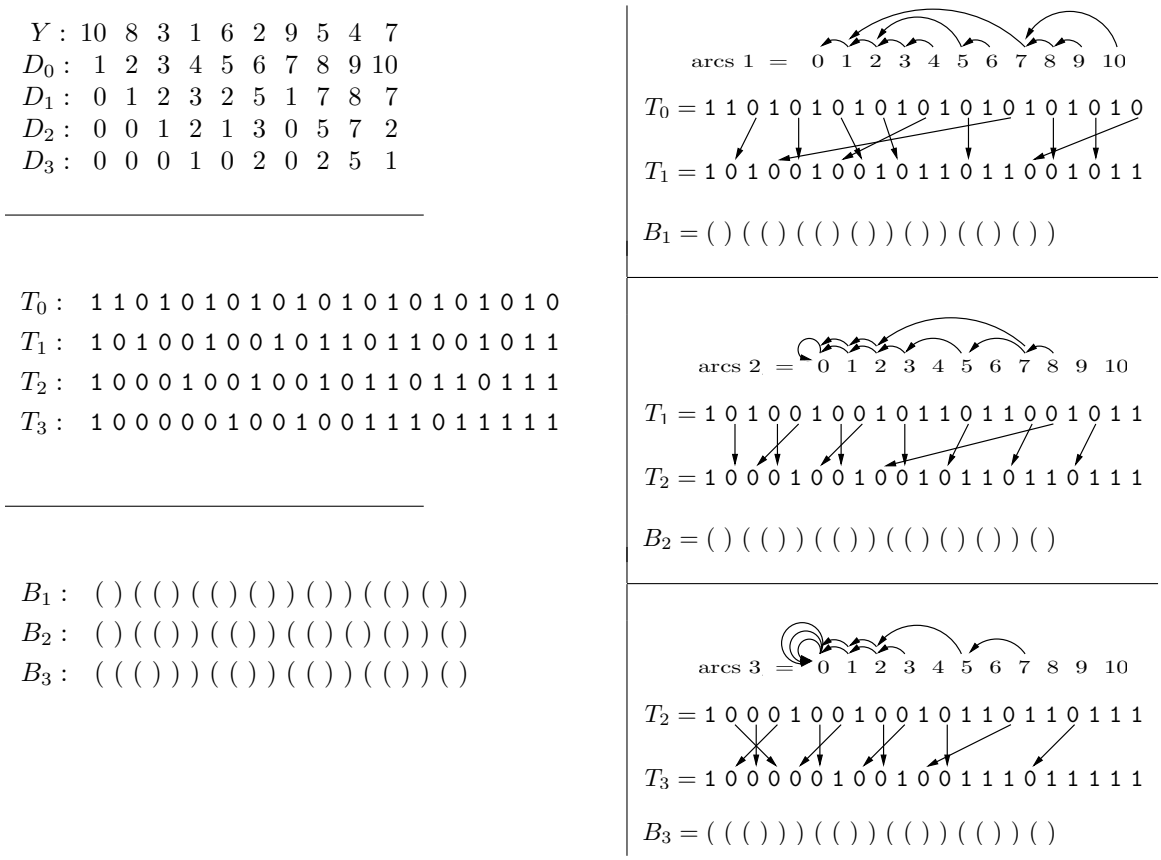

Figure 4: On the left, an example array Y, the data D_d we store on it, and our representation, T_d and B_d, of D_d. On the right, a graphical scheme of how the representation works.

We use the formula as follows. Starting with $z_0 = j$, we use the formula up to ℓ times in order to find, consecutively, $z_1, z_2, \ldots, z_\ell$. At any point we have that $D_d[j] = \mathsf{rank}_1(T_d, \mathsf{select}_0(T_d, z_d)) - 1 = \mathsf{select}_0(T_d, z_d) - z_d - 1$. This gives the desired answer $\mathsf{prev\text{-}larger}(j, d) = D_d[j]$. Note that we only store T_ℓ if we need to compute just $D_\ell[j]$.

Example. Figure 4 exemplifies the data structures on an array of $t = 10$ elements and for $\ell = 3$. On the left we show Y, the arrays D_0 to D_3, and their representation as the bit-vector T_d and the parenthesis sequences B_d. On the right we show how the data structures work. For each d, we first draw the arcs that go from $D_{d-1}[j]$ to $D_d[j]$ (note that, for $d > 1$, their sources may be far away from position j). If we want to compute $D_d[j]$, we start with the arc that leaves from j at "arcs 1", and this gives us $D_1[j]$. Then we go to "arcs 2" and find the arc that leaves from $D_1[j]$, which gives us $D_2[j]$. This way we compute any $D_d[j]$ in time $O(d)$. Note that, when several arcs lead to the same position, we are careful to pick the correct one in the next level. If the arrow arrives at position i left to right, it corresponds in the next level to the ith arrow that leaves, right to left. This is taken care of by the parentheses. □

By adding select support to T_d, and select and $\mathsf{findopen}$ support to B_d (Lemmas 1 and 8), we have that the overall space is $4\ell t + o(\ell t)$ bits.

Theorem 4 *Given an array $Y[1, t]$ and a parameter ℓ, we can build a structure that uses $4\ell t + o(\ell t)$*

bits and answers any query prev-larger(j, d) *on* Y *in time* $O(d)$*, for any* $1 \leq i \leq t$ *and* $1 \leq d \leq \ell$*, without accessing* Y*. If we only want to compute* prev-larger(j, ℓ)*, the structure uses* $2(\ell+1)t+o(\ell t)$ *bits of space. An analogous structure solves queries* next-larger(j, d)*.*

6 Constant-time access to E_v

In this section we describe a data structure that gives constant-time access to the values $E_v[1, O(\kappa)]$ in any node v. We recall the general picture for the sake of presentation. We begin with the input array $A[1, n]$, and create the tree T_C of $2t + 1$ nodes for the shallow cuttings on the set of points $\{(i, A[i])|1 \leq i \leq n\}$: note that each point is a pair given by a position i (the x-coordinate) and its value $A[i]$ (the y-coordinate), and there is a one-to-one mapping between points and positions. Without introducing ambiguity, we thus refer to the points represented in T_C for the properties but we actually store just their positions (not their values).

Specifically, consider a node $v \in T_C$: E_v is the set of positions such that $E_v[i]$ is the position in A of the value stored in $A_v[i]$, where array A_v stores the values belonging to the extent of v (essentially, letting $j = E_v[i]$, it is $A[j] = A_v[i]$, however the values in A_v are mapped to the interval $[1, \mathcal{O}(\kappa)]$ respecting the order between values.) For any node $v \in T_C$, we want to encode E_v so that each $E_v[i]$ can be retrieved in constant time. To this end, recall that v_- and v_+ are the nodes that precede and follow v in its extent: they can be accessed as shown in Lemma 15. Introducing the notation P_x to indicate the subset of $O(\kappa)$ positions from E_v whose corresponding points occur in the slab of node $x \in \{v_-, v, v_+\}$, we have that $E_v = P_{v_-} : P_v : P_{v_+}$. Hence, we will focus only on P_v without loss of generality, as we can easily simulate the concatenation $E_v = P_{v_-} : P_v : P_{v_+}$. Concretely, in this section we prove the following result.

Theorem 5 *Given the structures for constant-time navigation in* T_C *(Lemma 9) and for handling shallow cuttings in* T_C *(Lemmas 13 to 16), for any node* $v \in T_C$*, any position* $P_v[i]$ *can be retrieved in* $O(1)$ *time, with structures that use* $O(n \lg \kappa)$ *bits of space.*

The main idea is that most nodes in T_C cover a small span in A, and thus the x-coordinates of their points can be specified with a small offset. Nodes will be classified by subtree size, so that fewer bits are used for the P_v arrays of lower nodes. Some nodes of each class of subtree sizes will be *marked* and all their points will be stored explicitly using this technique. For the unmarked nodes, we observe that the points in their cells are inherited by their descendants, so we will find a way to describe the (marked) descendant where each point is to be retrieved.

6.1 Marking nodes

We define an exponentially decreasing sequence of sizes as follows: $t_0 = t$ and $t_{\ell+1} = \lceil \lg t_\ell \rceil$, until reaching a step z such that $t_z = 1$. Node v will be of *level* ℓ if $t_\ell^2 \leq |v| < t_{\ell-1}^2$ (recall that $|v|$ is the number of nodes in the subtree of v). For any $\ell \geq 1$, we mark a node $v \in T_C$ if it is of level ℓ and:

C1. it is a leaf or both its children are of level $> \ell$; or

C2. both its children are of level ℓ; or

C3. it is the root or its parent is of level $< \ell$.

Note that in fact there are no nodes of level $\ell = 0$. More generally, we have the following limit.

Lemma 18 *The number of marked nodes of level ℓ is $O(t/t_\ell^2)$.*

Proof. The key property is that the descendants of v are of the same level of v or more. So nodes marked by C1 above cannot descend from each other, thus each such marked node has at least t_ℓ^2 descendants not shared with another. As T_C has $2t+1$ nodes, there cannot be more than $(2t+1)/t_\ell^2$ nodes marked by this condition. By the same key property, nodes marked by C2 form a binary tree whose leaves are those marked by C1, thus there are at most other $(2t+1)/t_\ell^2$ nodes marked by C2. For C3, note that all unmarked nodes of level ℓ are in disjoint paths (otherwise the parent of two nodes of level ℓ would be marked by C2), and the path terminates in a node already marked by C1 or C2 (contrarily, a node of level ℓ marked by C3 must be a child of a node of level $< \ell$, and thus cannot descend from nodes of level ℓ, by the key property). Therefore, C3 marks the highest node of each such isolated path leading to a node marked by C1 or C2, and thus the number of nodes marked this way is limited by those marked by C1 or C2. □

6.2 Handling marked nodes

Marked nodes, across all the levels, are few enough to admit an essentially naive storage of their array P_v. If a marked node v represents a slab with left boundary $x_l + 1$, we store all its $P_v[o]$ values as the integers $P_v[o] - x_l$. As shown in Lemmas 13 and 15, we know both v and x_l, and thus we obtain $P_v[o]$ in constant time. Since a node of level ℓ contains less than $t_{\ell-1}^2$ descendants (about half of which are leaves), its slab spans less than $(t_{\ell-1}^2 + 1)/2$ consecutive split points x_i, and thus less than $\kappa(t_{\ell-1}^2 + 1)$ positions in A. Thus, each such integer $P_v[o] - x_l$ can be represented using $\lg(\kappa(t_{\ell-1}^2 + 1)) = O(t_\ell + \lg \kappa)$ bits.

We need a few further structures to give constant-time access to structures P_v, since their size depend on the level of the node. Our succinct representation of T_C gives the preorder rank $p(v)$ of node v in constant time (Lemma 9). We store a bit-vector $M[1, 2t + 1]$ where $M[p(v)] = 1$ iff node v is marked. Further, we store a string $L[1, O(t)]$ where we write down the level of each marked node, that is, $L[\mathsf{rank}_1(M, p(v))] = \ell$ iff v is marked and of level ℓ. Since every $\ell \leq \lg^* t$, the alphabet of L is $[0, \lg^* t]$. Then we can represent L using $|L|H_0(L) + o(t)$ bits so that operations access, rank, and select on L take $O(1 + \lg_w \lg^* t) = O(1)$ time (Lemma 5).

With M and L we can create separate storage areas per level for the explicit arrays P_v of marked nodes, each of which uses the same space, $\kappa \lg(\kappa(t_{\ell-1}^2 + 1))$ bits, for nodes of the same level ℓ. If a node v is marked (i.e., $M[p(v)] = 1$) and is of level $\ell = L[\mathsf{rank}_1(M, p(v))]$, then we store its array P_v as the rth one in a separate sequence for level ℓ, where $r = \mathsf{rank}_\ell(L, \mathsf{rank}_1(M, p(v)))$.

Lemma 19 *Constant-time access to any entry in P_v for any marked node v can be provided within $O(n \lg \kappa)$ total bits of space.*

Proof. We have explained how to store the arrays classified by level so as to provide constant-time access to any P_v. Let us now consider the space.

The arrays P_v themselves use $O(\kappa(t_\ell + \lg \kappa))$ bits each. The second term, $O(\kappa \lg \kappa)$ bits per marked node, adds up to $O(n \lg \kappa)$ bits overall. Since, by Lemma 18, there are $O(t/t_\ell^2)$ marked nodes of level ℓ, the first term, $O(\kappa\, t_\ell)$, adds up to $O((t/t_\ell^2) \cdot (\kappa\, t_\ell)) = O(n/t_\ell)$ bits over all the marked nodes of level ℓ. Adding over all the levels ℓ we have $O(n) \sum_{\ell=0}^{z} 1/t_\ell$. Since $t_z = 1$ and $t_{\ell-1} > 2^{t_\ell - 1}$, it holds that $t_{z-s} > 2^s$ for $s \geq 4$, and thus $\sum_{\ell=0}^{z} 1/t_\ell \leq O(1) + \sum_{s \geq 4} 1/2^s = O(1)$, therefore the terms $O(\kappa\, t_\ell)$ add up to $O(n)$ bits.

Bit-vector M uses $2t + 1 = O(n/\kappa)$ bits, whereas the storage of L uses $|L|H_0(L) + o(t)$ bits. Letting n_ℓ be the number of occurrences of ℓ in L, we have $|L|H_0(L) = \sum_\ell n_\ell \lg(|L|/n_\ell)$. Since $n_\ell \lg(|L|/n_\ell)$ is increasing[9] with n_ℓ and $n_\ell = O(t/t_\ell^2)$ by Lemma 18, we have $|L|H_0(L) \leq O(t)\sum_\ell \lg(t_\ell^2)/t_\ell^2 = O(t)\sum_\ell \lg(t_\ell)/t_\ell^2 \leq O(t)\sum_\ell 1/t_\ell = O(t)$ (we showed in the previous paragraph that the sum is $O(1)$). □

6.3 Handling unmarked nodes

The problem of supporting constant-time access to P_v is solved for marked nodes, but T_C may have $\Theta(t)$ unmarked nodes. To deal with unmarked nodes, we first observe that an unmarked node v at level ℓ has exactly one level ℓ child and one child x at level $> \ell$ (otherwise v would be marked by C1 or C2). Furthermore, x is marked by C3. Finally, the marked parent of an unmarked level ℓ node must be the root or at level ℓ itself. Thus, as already observed in the proof of Lemma 18, level-ℓ unmarked nodes form disjoint paths in T_C, and all the nodes adjacent to such paths are marked.

Now consider the points in slabs corresponding to unmarked nodes. When a cell is closed and split into two, the leftmost (rightmost) κ points in its slab become part of its left (right) child cell. Thus, each child cell starts out with κ *inherited* points, which are in common with its parent slab, and (at most) κ further *original* points will be added to it before it is itself closed (becoming a child slab) and split.

For each point of node v, in x-coordinate order, we use a bit to specify if the point is inherited ($\mathsf{0}$) or original ($\mathsf{1}$). Let $o_v[1, 2\kappa]$ be this bit-vector, which will be stored for all the unmarked nodes $v \in T_C$, at a total cost of $O(n)$ bits. We now describe how to recover the position (contained in P_v) of an original and an inherited point, with different mechanisms.

6.3.1 Retrieving the positions of original points

Let π be a path of unmarked nodes of level ℓ, and let v be an unmarked node in π. Each original point p of v must become an inherited point of some marked descendant v' that is adjacent to π (recall that v' represents all the positions of its points explicitly). Thus the coordinate of each such original point p can be specified by recording which marked descendant v' contains it, and the rank of p among the points of v'.

The ranks are stored in an array $r_v[1, \kappa]$, with one entry per original point in v. The distances require a more sophisticated mechanism. Suppose that the jth original point in v is in v's marked descendant v' at distance d_j along π. Note that the point is inherited by the d_j intermediate descendants of v as well. Then we write the bit-vector $b_v = \mathsf{1}^{d_1-1}\mathsf{0}\mathsf{1}^{d_2-1}\mathsf{0}\ldots\mathsf{1}^{d_\kappa-1}\mathsf{0}$.

The vectors o_v, r_v and b_v are concatenated in the same preorder as the nodes. While vectors o_v and r_v are of fixed size, vectors b_v are not. So we can concatenate all the bit-vectors $o_v[1, 2\kappa]$ and vectors $r_v[1, \kappa]$ in preorder into a global bit-vector $o[1, O(\kappa t)] = o[1, O(n)]$ and a global array $r[1, O(\kappa t)] = r[1, O(n)]$. Then, if v is unmarked (i.e., $M[p(v)] = \mathsf{0}$), $o_v[i]$ is at $o[2\kappa(m-1)+i]$, where $m = \mathsf{rank}_\mathsf{0}(M, p(v))$, and moreover $\mathsf{rank}_\mathsf{1}(o_v, i) = \mathsf{rank}_\mathsf{1}(o, 2\kappa(m-1)+i) - \mathsf{rank}_\mathsf{1}(o, 2\kappa(m-1))$. Given any original point $o_v[i] = 1$, it is the jth original point for $j = \mathsf{rank}_\mathsf{1}(o_v, i)$, and thus its corresponding entry is $r_v[j]$, which is found at $r[\mathsf{rank}_\mathsf{1}(o, i)]$. Finally, we concatenate all the bit-vectors b_v for the unmarked nodes v in preorder creating a bit-vector b. If $o_v[i] = 1$ and $j = \mathsf{rank}_\mathsf{1}(o_v, i)$, then we

[9] At least for $n_\ell \leq |L|/e$. When n_ℓ is larger we can simply bound $n_\ell \lg(|L|/n_\ell) = O(n_\ell)$, thus we can remove all those large n_ℓ terms from the sum and add an extra term $O(t)$ to absorb them all.

recover $d_j = \mathsf{select}_0(b_v, j) - \mathsf{select}_0(b_v, j-1)$. On the concatenated bit-vectors, for the original point $o[i]$ we compute $j = \mathsf{rank}_1(o, i)$ and then $d_j = \mathsf{select}_0(b, j) - \mathsf{select}_0(b, j-1)$.

Thus, to obtain the position $P_v[i]$ of an original point (i.e., with $o_v[i] = 1$) in an unmarked node v, we obtain the distance d_j to the marked descendant v' where $P_v[i]$ is stored. Since v' is marked, the answer obtained in constant time (Lemma 19) from $P_{v'}[r[\mathsf{rank}_1(o, i)]]$ (to which we add the starting position of the slab of v').

The remaining problem is then to find the marked node v' leaving π at distance d_j from v. The strategy is to find the node u' that is "at the end" of π. More precisely, u' is a child of the lowest node of π and is the only node leaving π that is of the same level ℓ of v (thus u' is marked). Since we can compute node depths and ancestors at any distance in constant time on T_C (Lemma 9), we can compute the ancestor a of u' that is at depth $depth(v) + d_j - 1$, and find v' as the child of a that is not in π, that is, is not an ancestor of u'.

There is a slight ambiguity to describe v' using d_j: both u' and its sibling leave π, and they are at the same distance to their ancestors. To distinguish them, we encode $d_j + 1$ instead of d_j in b_v to denote the node u', whereas its sibling is denoted with d_j as usual. Therefore, when we compute a and it holds $a = u'$, we know that $v' = u'$.

We still need to find u'. The key property is that u' is the highest marked node of level ℓ in the subtree of v. We calculate the subtree size of v in constant time (Lemma 9) and hence its level ℓ.[10] If the nodes are arranged in preorder, u' is the first node appearing after $p(v)$, $p(u') > p(v)$, which is marked ($M[p(u')] = 1$) and whose level is $L[\mathsf{rank}_1(M, p(u'))] = \ell$. This corresponds to the first occurrence of ℓ in L after position $\mathsf{rank}_1(M, p(v))$, and is found in constant time with $p = \mathsf{select}_\ell(L, \mathsf{rank}_\ell(L, \mathsf{rank}_1(M, p(v))) + 1)$. Then $p(u')$ is $\mathsf{select}_1(M, p)$. Finally, the tree representation gives us u' from its preorder rank $p(u')$ in constant time as well (Lemma 9).

Lemma 20 *Constant-time access to the position $P_v[i]$ of any original point in the unmarked nodes v can be provided within $O(n \lg \kappa)$ bits of space.*

Proof. We have already explained how constant-time access is provided. Let us analyze the space. The arrays o and r require $O(n)$ and $O(n \lg \kappa)$ bits, respectively. To bound the space of array b we claim that, summed across all the nodes v in the path π, the arrays b_v add up to $2|\pi|\kappa$ bits: each b_v has κ 0-bits, and each 1-bit in b_v represents the same point when it is inherited in a descendant of v along π. Since π contains in total $|\pi|\kappa$ inherited points, the 1s in all the bit-vectors b_v of π also add up to $|\pi|\kappa$. Thus, $|b| = \sum_{v \in T_C} |b_v| = O(t\kappa) = O(n)$ bits. Arrays M and L were already considered in Lemma 19. □

6.3.2 Retrieving the positions of inherited points

We cannot use bit-vectors analogous to b_v for the inherited points in v, as we cannot bound their size (because the same points are inherited over and over along π). For each inherited point p in v, we instead specify which ancestor of v on π has p as an original point, and then retrieve the position of the point as that of an original point in the ancestor using Lemma 20. If the ancestor is outside π, we specify the marked parent u of the topmost unmarked node in π, and retrieve the

[10] To find the level in constant time from the subtree size, we can check directly for the case $\ell = 0$, and store the other answers in a small table of $\lg^2 t$ cells.

position from P_u using Lemma 19 (as u is marked). In the rest of this subsection, we assume that the ancestor is inside π.

To specify the ancestors, we code the points using 4κ *colors*. Of these colors, 2κ are said to be *original* colors and 2κ are said to be *inherited* colors. For each original color g there is a corresponding inherited color g'. All the points in u are given arbitrary distinct original colors. Then we traverse the nodes v in π top to bottom. If point p in v is inherited (from its parent v'), we look at the color of p in v'. If p has an original color g in v', we give p color g' in v. Otherwise, if p is also inherited in v', having color g', it will also have color g' in v. On the other hand, if point p is original in v, we give it one of the currently unused original colors: any color g such that g is not already an original color in v and g' is not among the κ inherited colors of v can be used as the original color for p. Note that no colors g and g' can be present simultaneously in any v', thus writing g' in v unambiguously determines which color is inherited from v'. The colors of node v are represented in a string $c_v[1, 2\kappa]$, adding up to $O(n \lg \kappa)$ bits.

This scheme gives sufficient information to track the inheritance of points across π: conceptually when a new, original, point p appears in v, it is given an original color g. Then the point is inherited along the descendants of v as long as color g' exists below v. Thus, to find the appropriate ancestor of v that contains, as an original point, a given inherited point p of color g', we concatenate all the color strings c_v on π into a string c_π, top to bottom, and ask for the nearest preceding occurrence of color g. Inside c_π, the subarray c_v starts at position $2\kappa(\mathit{depth}(v) - \mathit{depth}(u)) + 1$. Thus, we seek to find the rightmost $c_\pi[j] = g$ preceding some $c_\pi[i] = g'$. With j, we have that v' is the ancestor of v at depth $\mathit{depth}(u) + \lceil j/(2\kappa) \rceil - 1$, and the position of the desired (original) point is $P_{v'}[j \bmod 2\kappa]$.

The sequence of colors c_π will be associated with the last node u' of π, and all of them will be concatenated in preorder of those nodes u'. A bit-vector $B[1, O(t)]$ will mark the starting position of each sequence c_π in the concatenation (by chunks of 2κ entries), and another bit-vector $R[1, 2t + 1]$ contains all 0s except $R[p(u')] = 1$ for all the nodes u' of all the paths π. Thus we have access to any individual sequence c_π: for any $v \in \pi$ terminated in u' (Section 6.3.1 explains how to compute u'), c_π starts at position $1 + 2\kappa(\mathsf{select}_1(B, \mathsf{rank}_1(R, p(u'))) - 1)$ of the concatenated sequence.

To find j, we will not represent c_π directly, but rather c'_π, where both the original colors g and the inherited colors g' are written as g. To distinguish them, we store 2κ bit-vectors c^g_π, so that $c^g_\pi[\mathsf{rank}_g(c'_\pi, i)] = 1$ iff $c_\pi[i] = g$ (and 0 iff $c_\pi[i] = g'$). We use a representation for c'_π that requires $O(|\pi|\kappa \lg \kappa)$ bits and gives constant select time (Lemma 4). We also add the structure of Lemma 7 to c'_π. This adds $O(|\pi|\kappa \lg \lg \kappa)$ bits and allows us to compute $r = \mathsf{rank}_g(c'_\pi, i)$ in constant time, given that $c'_\pi[i] = g$. Then we find the latest 1 in $c^g_\pi[1, r]$, $o = \mathsf{select}_1(c^g_\pi, \mathsf{rank}_1(c^g_\pi, r))$. This corresponds to the last occurrence of g preceding $c_\pi[i] = g'$. The position is mapped back from $c^g_\pi[o]$ to c_π with $j = \mathsf{select}_g(c'_\pi, o)$.

Lemma 21 *Constant-time access to the position $P_v[i]$ of any inherited point in the unmarked nodes v can be provided within $O(n \lg \kappa)$ bits of space.*

Proof. We have already explained how to obtain the position in constant time. The space is dominated by the sequences c_v, represented as c_π and these in turn as c'_π and c^g_π, which add up to $O(n \lg \kappa)$ bits. Bit-vectors B and R use just $O(n/\kappa)$ further bits. □

Lemmas 19, 20, and 21 prove Theorem 5.

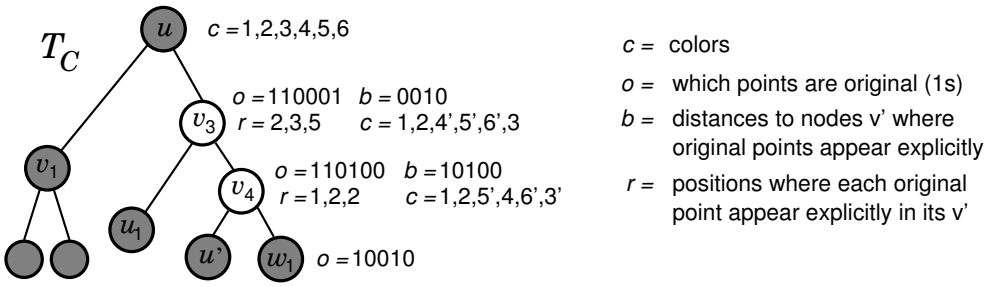

Figure 5: Illustration of the structures for constant-time access to P_v on T_C, assuming that nodes v_3, v_4, and u' are all of the same level. Marked nodes are filled with gray.

Example. In the tree T_C in Figure 2, nodes v_2 and v_3 are of level 1 and the rest are of level 2, and all turn out to be marked. To show a more interesting example, Figure 5 assumes that the grid has more points towards the bottom, so that the leaves that descended from v_3 and v_4 are now internal nodes (and have new labels u_1, u', and w_1, whose reason will be clear later), so that nodes v_3, v_4, and u' are all of the same level ℓ, whereas u_1 and w_1 are of level $\ell + 1$. Then the path is $\pi = \langle v_3, v_4 \rangle$, node u (which was v_2 in Figure 2) is the upper limit of π, and node u' acts as its lower limit. For example, $o_{v_3} = 110001$ because the first, second, and sixth points in the slab of v_3, read left to right, are original, whereas the others are inherited from u. Also, $b_{v_3} = 0010$, indicating $d_1 = 1$, $d_2 = 1$, and $d_3 = 2$, because the first and second original points are inherited by u_1, which is the node at distance 1 that leaves π. Instead, the third original point of v_3 is inherited by w_1, which is the node at distance 2 from v_3 that leaves π (u' is also at distance 2; to avoid ambiguities we assume it is at distance 3, as explained soon). The positions where those original points are represented in the marked nodes that leave the path are $r_{v_3} = 2, 3, 5$, since the first and second are the second and third points in u_1, and the third original point of v_3 is the fifth point in w_1. Finally, $c_{v_3} = 1, 2, 4', 5', 6', 3$ because (as shown in o_{v_3}), the third, fourth, and fifth points in v_3 are inherited, and they correspond to the original points marked 4, 5, and 6, in the parent u. The three new original colors of v_3 receive arbitrary free colors 1, 2, and 3. In v_4, three points (with colors $5'$, $6'$, and $3'$) are inherited, corresponding to those with colors $5'$ and $6'$ in v_3 (which are in turn inherited from 5 and 6 in u), and to the one with color 3 in v_3, which is original in that node. The other three colors in v_4 are original and receive free original colors 1, 2, and 4. We also show the array o_{w_1}, since later in the article marked nodes will also store these bit-vectors. □

7 Predecessor queries on E_v

Having constant-time access to E_v enables searching for the desired limits where the queries are to be run. Recall that our queries involve a range $A[i, j]$ and, for a suitable node $v \in T_C$, this translates into finding the largest l and the smallest r such that $E_v[l-1] < i \leq j < E_v[r+1]$ (see Section 4.2). This is a form of predecessor query on E_v that we can perform by a binary search. However the resulting $O(\lg \kappa)$ search time is larger than the promised time complexity. In this section we obtain faster predecessor searches that replace the binary search. Once again, we will focus on providing predecessor searches on P_v, the positions of the points in the slab of v. Predecessors on $E_v = P_{v_-} : P_v : P_{v_+}$ are then obtained by finding the neighbor nodes v_- and v_+,

as shown in Lemma 15, and then determining with a couple of comparisons whether to run the query on P_{v-}, P_v, or P_{v+}. Concretely, in this section we prove the following theorem:

Theorem 6 *Given the structures for constant-time navigation in T_C (Lemma 9), for handling shallow cuttings in T_C (Lemmas 13 to 16), and for constant-time access to all arrays P_v in T_C (Theorem 5), predecessor queries on the array P_v of any node $v \in T_C$ can be carried out in time $O(1 + \lg\kappa/\lg\lg n)$ using $O(n \lg\kappa)$ bits of space.*

A classical predecessor structure [30] on $P_v[1, 2\kappa]$ uses $O(\kappa \lg n)$ bits, as the universe is $[1, n]$, the set of positions in A. These spaces would add up to $O(n \lg n)$ bits (note that this structure is needed in all the $O(t)$ nodes of T_C, not only the marked ones). Instead, since we have independent constant-time access to P_v, we use succinct SB-trees (Lemma 10).

On a node v of level ℓ, the universe of positions is of size $O(\kappa\,|v|) = O(\kappa\, t_{\ell-1}^2)$, thus the succinct SB-tree would use $O(\kappa \lg\lg(\kappa\, t_{\ell-1}^2)) = O(\kappa \lg t_\ell + \kappa \lg\lg\kappa)$ bits. While the second term adds up to $O(n \lg\lg\kappa)$, the first term is still too large: just considering the nodes with $\ell = 1$, it adds up to $O(n \lg\lg n)$ bits if we store this structure on every node of T_C.

To reduce space, we will store this structure only on *sampled* nodes, and will handle the unsampled ones with other techniques. We will sample all the nodes marked in Section 6, and in addition we will will further sample every $(t_\ell/\lg^2 t_\ell)$th node in the paths π of unmarked nodes of level ℓ. To associate information with sampled nodes of each level, we use the analogous of bitvector M and sequence L of Section 6.2.

Lemma 22 *Predecessor queries on the array P_v of any sampled node v can be carried out in time $O(1 + \lg\kappa/\lg\lg n)$ using $O(n \lg\lg\kappa)$ bits of space.*

Proof. According to Lemma 10, the predecessor time with the succinct SB-tree stored at the node is $O(1+\lg\kappa/\lg\lg(\kappa\, t_{\ell-1}^2))$. This can be improved to $O(1+\lg\kappa/\lg\lg n)$ by using the same precomputed table over a universe of size n for all the nodes; this table requires $o(n)$ further bits.

Let us consider space. The number of sampled nodes of level ℓ is $O(t \lg^2 t_\ell/t_\ell)$, which added over all the levels is $\sum_\ell t \lg^2 t_\ell/t_\ell \le t\,(O(1) + \sum_{s\ge 4} s^2/2^s) = O(t)$ (as in the proof of Lemma 19). Therefore, the term $O(\kappa \lg\lg\kappa)$ in the bit space of succinct SB-trees adds up to $O(n \lg\lg\kappa)$. The other component of the space, $O(\kappa \lg t_\ell)$ bits, adds up to $O(n \lg^3 t_\ell/t_\ell)$ bits for level ℓ. Adding up over all the levels ℓ we have $O(n)\sum_\ell \lg^3 t_\ell/t_\ell \le O(n)(O(1) + \sum_{s\ge 4} s^3/2^s) = O(n)$ bits. Finally, the analogous of bit-vector M uses $O(t)$ bits and the analogous of L uses $O(n_\ell \lg(|L|/n_\ell))$ bits (recall the proof of Lemma 19). Since now $n_\ell = O(t \lg^2 t_\ell/t_\ell)$, this space is $O(t)\sum_\ell \lg^3 t_\ell/t_\ell = O(t)$. □

The paths of unsampled nodes of level ℓ have length $O(t_\ell/\lg^2 t_\ell)$. To provide predecessor searches on unsampled nodes, let us consider one such path π and let v be a node in π. The nodes leaving the path are of level $> \ell$, except the node u' leaving π at the bottom, which is of level ℓ. Therefore, we can divide the range of split points covered by π into three *areas*:

1. The area covered by the subtrees that leave π to the left.
2. The area covered by the subtrees that leave π to the right.
3. The area covered by u'.

Each of those areas is contiguous, (1) preceding (3) preceding (2). Since there are $O(t_\ell)$ subtrees of type (1) and each has nodes of level at least $\ell+1$, the total area covered by those subtrees is of size $O(t_\ell \cdot \kappa\, t_\ell^2) = O(\kappa\, t_\ell^3)$. The case of (2) is analogous. Area (3), instead, can be significantly larger since u' can be of level ℓ. Our solutions will use these areas in different ways depending on whether $\kappa = \Omega(\lg\lg n)$ or $\kappa = O(\lg\lg n)$. We describe each case separately.

7.1 Handling large κ values

When $\kappa = \Omega(\lg\lg n)$, we can afford to store, for each (unsampled) node $v \in \pi$, a succinct SB-tree for the values of P_v falling in area (1) and another for the values in area (2), both using $O(\kappa \lg\lg(\kappa\, t_\ell^3)) = O(\kappa \lg\lg(\kappa\, t_\ell))$ bits. Given a predecessor request on v, we first find the node u' below π as in Section 6.3.1, and determine in constant time whether the query falls in the area (1), (2), or (3) (by obtaining the limits $[x_l+1, x_r]$ of u', Lemma 16). If the query falls in areas (1) or (2) we use the corresponding succinct SB-tree of v, otherwise we use the succinct SB-tree of u' (which is sampled and hence stores a regular succinct SB-tree).

While the succinct SB-trees for areas (1) and (2) are built for v and store the positions of the points of P_v, this is not the case of the regular succinct SB-tree of u', since not all the points in u' are points in v. In this case, given the predecessor $P_{u'}[q]$ of a position p, we must still find the predecessor of $P_{u'}[q]$ in P_v. The points inherited in $P_{u'}$ form a central band in P_v, starting at position p_v. Thus we store, for each node v, a bit-vector $h_v[1, 2\kappa]$ indicating which of the points in its corresponding node u' are inherited from v, as well as p_v. Then the final answer is $p_v + \mathsf{select}_1(h_v, \mathsf{rank}_1(h_v, q)) - 1$, which is computed in constant time. These arrays add $O(n)$ bits of space.

Example. Figure 6 (left) shows a schematic example of this arrangement. A path $\pi = \langle v_1, v_2, v_3, v_4\rangle$ of level ℓ is limited by u and u'. Nodes u_1 and u_2, of level $> \ell$, leave π from the left and w_1 and w_2, also of level $> \ell$, leave from the right. Node u' is of level ℓ and is sampled, so it has its own SB-tree. The other nodes leaving π cover a smaller area, so we can afford two SB-trees for each v, storing the positions of the split points of P_v inside the u_i nodes and inside the w_j nodes. For example, if we build the SB-trees for v_2, we include in the left succinct SB-tree the positions P_{v_2} of the points that are inherited in $\langle u_2\rangle$, and in the the right succinct SB-tree the positions P_{v_2} of the points that are inherited in $\langle w_1, w_2\rangle$ (u_1 cannot have points of P_{v_2} because it does not descend from v_2). □

Lemma 23 *If $\kappa = \Omega(\lg\lg n)$, then predecessor queries in the array P_v of any unsampled node v can be carried out in time $O(1 + \lg\kappa/\lg\lg n)$ using $O(n\lg\kappa)$ bits of space.*

Proof. The time is dominated by the succinct SB-trees, which was explained in Lemma 22. The space of the two additional succinct SB-trees for a node of level ℓ is $O(\kappa \lg\lg(\kappa\, t_\ell))$ bits. This adds up to $O(n(\lg\lg\kappa + \lg\lg\lg n))$ bits, the second term being dominated by the (unsampled) nodes of level $\ell = 1$. Since $\lg\kappa = \Omega(\lg\lg\lg n)$, the space is bounded by $O(n\lg\kappa)$ bits. □

7.2 Handling small κ values

When $\kappa = O(\lg\lg n)$ we will not store succinct SB-trees for areas (1) and (2) for each unsampled node as before, but will use a different mechanism. Let π be a path of unsampled nodes of level

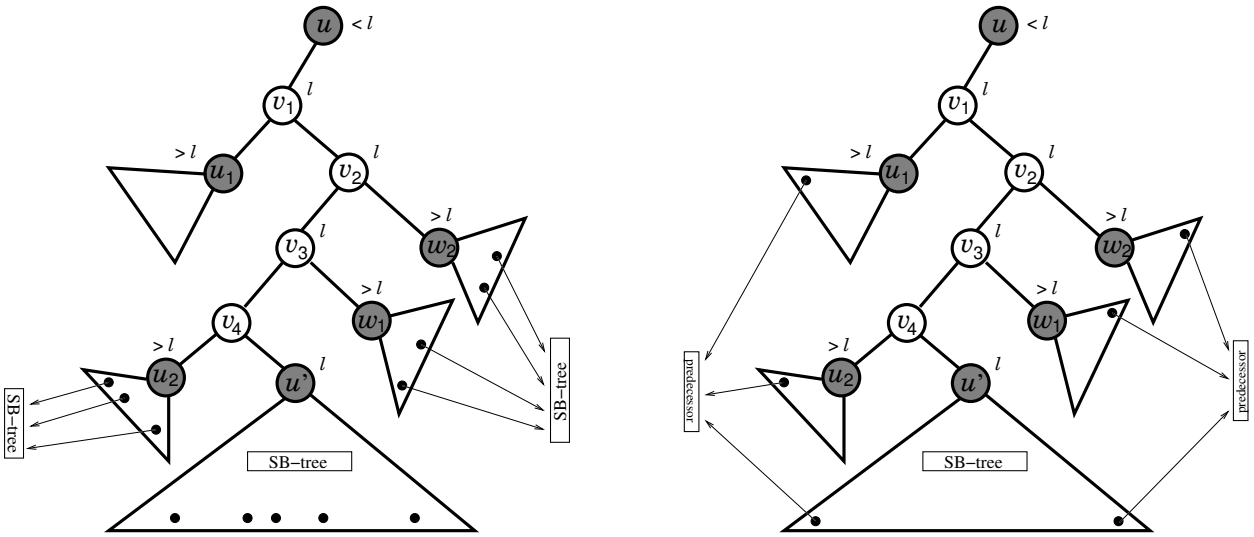

Figure 6: Illustration of the scheme to compute predecessors on paths of unsampled nodes. On the left, the structure for v_2 when $\kappa = \Omega(\lg\lg n)$. The black dots indicate the points inherited from v_2. On the right, the structure for the whole path when $\kappa = O(\lg\lg n)$. The black dots indicate the first/last points of the subtree areas.

ℓ. Let $u_1, u_2, \ldots$ be the nodes that leave π from the left, reading their areas in left-to-right order (i.e., top-down in π) until reaching u', and $w_1, w_2, \ldots$ be the nodes that leave π from the right, also reading them in left-to-right order (i.e., bottom-up in π) from u'. Then the area of A covered by π can be partitioned into the $|\pi|+1$ consecutive areas covered by $u_1, u_2, \ldots, u', w_1, w_2, \ldots$. All those nodes are sampled and thus store their own succinct SB-trees.

We will use a single predecessor structure, associated with π (not with any particular node $v \in \pi$), to determine in which of those $|\pi|+1$ areas the query p belongs (if the query is done for a node $v \in \pi$, then the node containing that area will descend from v).

Let ℓ_i be the level of node u_i. Then the area covered by u_i is of length $O(\kappa\, t^2_{\ell_i - 1})$. Thus we can encode those lengths with, say, γ-codes [5], within $2\sum_i \lg(\kappa\, t^2_{\ell_i-1}) = O(|\pi|\lg\kappa + \sum_i t_{\ell_i})$ bits. To facilitate decoding this description, we will insert areas of length zero every time π goes left (when encoding the areas u_i) or every time π goes right (when encoding the areas w_j). This does not change the asymptotic length of the description.

From a space accounting point of view, this space can be afforded because we can charge $O(\lg\kappa + t_{\ell_i})$ bits to the storage of u_i. As u_i's level is larger than ℓ, it is a marked node (see Section 6). Thus there are $O(t/t^2_{\ell_i})$ such nodes overall, each of which will be charged $O(t_{\ell_i})$ bits only once, from the path π it leaves, for a total of $O(t/t_{\ell_i})$ bits, and this adds up to $O(t)$ bits overall (see the proof of Lemma 19). As for the term $O(\lg\kappa)$, it adds up to $O(t\lg\kappa)$ bits overall.

On the other hand, note that, since $\ell_i > \ell$, it holds that $O(|\pi|\lg\kappa + \sum_i t_{\ell_i}) = O(|\pi|\lg\kappa + |\pi|\lg t_\ell)$. Since $|\pi| = O(t_\ell/\lg^2 t_\ell)$, $t_\ell = O(\lg n)$ even for $\ell = 1$, and $\kappa = O(\lg\lg n)$, the length is $O(\lg n/\lg\lg n) = o(\lg n)$ bits, and thus the whole description of the u_i area lengths fits in $O(1)$ computer words.[11] Moreover, there are $2^{O(\lg n/\lg\lg n)}$ possible descriptions of area lengths for $u_1, u_2, \ldots$, and $O(|\pi|\kappa\, t^2_{\ell_i-1}) = o(\lg^3 n)$ possible queries. Thus we can build a global table of

[11] This is why we need the $\lg^2 t_\ell$ term dividing t_ℓ in the definition of our sampling.

$2^{O(\lg n/\lg\lg n)} \times o(\lg^3 n) \times \lg n = o(n)$ bits storing the answer to every possible query on every possible path. Thus the queries take constant time. We proceed analogously with the areas of $w_1, w_2, \ldots$.

Now, a predecessor query for the areas $u_1, u_2, \ldots, u', w_1, w_2, \ldots$ can be answered as follows: As in Section 7.1, we first determine whether the answer is in u' with a constant number of comparisons, and if so, we obtain the answer with the succinct SB-tree of u'. Otherwise, the answer is in the areas to the left (called u_i) or to the right (called w_j) nodes of u'. In either case, we use the precomputed tables to determine in constant time the index i (left) or j (right) of the area where the predecessor lies. If the answer is on the left area, we compute $v' = u_i$ from the index i as in Section 6.3.1: we find the ancestor a of u' at depth $\mathit{depth}(u) + i$, and then v' is the child of a that is not in π (i.e., is not an ancestor of u'). If the answer is on the right area, we compute $v' = w_j$ similarly, but now a is the ancestor of u' at depth $\mathit{depth}(u') - j$. Note that this works because we have inserted the empty areas in the γ-encoded descriptions.

Example. Figure 6 (right) illustrates the structure for small κ values. Now the predecessor structures associated with π (i.e., the γ-encoded descriptions) store only one extreme split point from each node leaving π. We must insert two empty areas between u_1 and u_2, so the index of u_2 in the γ-encoded description is actually 4, and it is indeed the child not in π of the ancestor of u' at depth $\mathit{depth}(u) + 4$. Similarly, we insert an empty area before w_1 and one after w_2. Then the index of w_1, for example, is 2, and it is the child not in π of the ancestor of u' at depth $\mathit{depth}(u') - 2$. □

Now we use the succinct SB-tree of v' (which is sampled) to find the position of the predecessor of p in its $P_{v'}$ array, $P_{v'}[q]$. The final challenge is to map that position in v' to the corresponding position in v. Note that P_v contains only some of the positions of $P_{v'}$ in the area covered by v' (where p lies), so we seek the predecessor of $P_{v'}[q]$ in P_v. To compute this efficiently, we will reuse the point inheritance information encoded in the bit-arrays o_v of Section 6. With the sequence of $|\pi|$ consecutive arrays o_v, and knowing whether each node in π is a left or a right child, we have sufficient information to track any position $P_{v'}[q]$ upwards and determine its predecessor in P_v.

Let $q' = \mathsf{rank}_0(o_{v'}, q)$ be the number of inherited points having positions in $P_{v'}[1, q]$, and v'' be the parent of v'. If v' is the left child of v'', then the first half of the points of v'' are inherited by v', and therefore the position $P_{v'}[q]$, or its predecessor, in v'' is $P_{v''}[q']$ (note that q' can be zero). If, instead, v' is the right child of v'', then the position is $P_{v''}[\kappa + q']$ since v' inherits the second half of the points. Now we repeat the process from v'' until reaching v, where we obtain the final predecessor position in P_v.

Example. Consider Figure 5 and let v_3 have no SB-tree of its own. Assume that a predecessor search in v_3 is found to fall inside the node w_1. Since w_1 is marked (and thus sampled), it has its own SB-tree, which is searched to find the predecessor, $P_{w_1}[2]$ (this is the 17th left-to-right point in Figure 2). Since $o_{w_1}[2] = 0$, the point is inherited. It is the first inherited point because $\mathsf{rank}_0(o_{w_1}, 2) = 1$. Since w_1 is a right child and $\kappa = 3$, the point is the fourth $(3 + 1)$ in v_4. The point is original in v_4, since $o_{v_4}[4] = 1$. The number of inherited points in v_4 preceding this original point is $\mathsf{rank}_0(o_{v_4}, 4) = 1$, so the first inherited point is the predecessor in v_3, the parent of v_4. Since v_4 is a right child, the predecessor is the fourth left-to-right point $(3 + 1)$ in the slab of v_3. □

To use the bit-vectors o_v in this way, we cannot use the same array $o[1, O(n)]$ where they were stored in preorder in Section 6.3.1. Rather, we must store another copy of bit-vectors o_v in the

path-wise form used to store the sequences c_v in Section 6.3.2, so that all the bit-vectors o_v for unsampled nodes $v \in \pi$ are stored contiguously in a sequence o_π. In addition, we need the bit-vectors $o_{v'}$ for sampled nodes v'. Sampled nodes can be handled as belonging to an empty path where the sampled node acts as u', and we also store $o_{u'}$ in o_π. The space for this new copy of the o_v bit-vectors is $O(n)$ bits. We similarly store the information on left/right directions along each path π, contiguously and adding up to $O(t)$ bits.

Now the bit-vectors o_v and the path directions along π are stored contiguously and add up to length $2|\pi|\kappa$ and $|\pi|$, respectively. Thus, once again, we can prepare a global table that takes every possible concatenation of bit-vectors o_π, a bit-vector $o_{v'}$, the $|\pi|$ left/right directions along the path π, the depths of v and v' in π, and the value q, and it returns the corresponding predecessor in P_v in constant time. The table uses $2^{(|\pi|+1)(2\kappa+1)}|\pi|^2\kappa \cdot \lg n$ bits. Since $\kappa = O(\lg\lg n)$, this is $2^{O(\lg n/\lg\lg n)}o(\lg^3 n) = o(n)$ bits.

Lemma 24 *If $\kappa = O(\lg\lg n)$, then predecessor queries in the P_v array of any unsampled node v can be carried out in time $O(1 + \lg\kappa/\lg\lg n)$ using $O((n/\kappa)\lg\kappa) + o(n)$ bits of space.*

Proof. The time is again dominated by the succinct SB-tree of u', which was explained in Lemma 22. The space is that of the γ-encoded descriptions and global tables. □

Lemmas 22, 24, and 23 complete the proof of Theorem 6.

8 Wrapping up

From the previous elements, we can now assemble a structure that, given a value κ, uses $O(n\lg\kappa)$ bits and answers a query $\mathsf{sel}(i,j,k)$ for any $1 \le k \le \kappa$ in time $O(1 + \lg\kappa/\lg\lg n)$:

1. As described in Section 5 (Lemma 12), we find the maximal interval $[m, M]$ such that $i \le x_m \le x_M \le j$.
2. If the interval contains zero or one split point, then $A[i,j]$ can be directly solved with the range selection structure [8] associated with the special extent $[x_{m-1}+1, x_{m+1}]$ of the split point x_m, which covers at most 4κ consecutive entries of A.
3. Otherwise, we find the highest node $v \in T_C$ containing $[x_m+1, x_M]$, as well as the other two neighbor nodes that span the extent of v, namely, v_- to the left and v_+ to the right, all in constant time, as described in Section 5 (Lemmas 13, 14, and 15).
4. Using the structures of Section 7 (Theorem 6), we find the predecessor $l-1$ of $i-1$, and the predecessor r of j, within the positions of $E_v = P_{v_-} : P_v : P_{v_+}$, in time $O(1 + \lg\kappa/\lg\lg n)$. These structures need access to entries in P_{v_-}, P_v, and P_{v_+}, which is provided in constant time in Section 6 (Theorem 5).
5. We use the range selection structure [8,9] associated with the extent of node v (which has at most 6κ entries) to run the query $o = \mathsf{sel}(l, r, k)$. The time is $O(1 + \lg\kappa/\lg\lg n)$.
6. We use the structures of Section 6 (Theorem 5) to compute the final answer $E_v[o]$ in constant time, which is again provided via direct access to arrays P_{v_-}, P_v, or P_{v_+}.

In order to reduce the time from $O(1+\lg\kappa/\lg\lg n)$ to $O(1+\lg k/\lg\lg n)$, we build our data structures for values $\kappa_s = 2^{2^s}$, for $s = 0, 1, \ldots, \tau$, where τ is such that $2^{2^{\tau-1}} < \kappa \leq 2^{2^\tau}$. The space for those structures is $O(n)\sum_{s=0}^{\tau} \lg \kappa_s = O(n)\sum_{s=0}^{\tau} 2^s < O(n)\, 2^{\tau+1} < O(n)\, 4\lg\kappa = O(n\lg\kappa)$. A query $\mathsf{sel}(i,j,k)$ is run on the structure for κ_s such that $\kappa_{s-1} < k \leq \kappa_s$, that is, $2^{s-1} < \lg k \leq 2^s$,[12] and thus its query time is $O(1+\lg\kappa_s/\lg\lg n) = O(1+2^s/\lg\lg n) = O(1+\lg k/\lg\lg n)$. This proves Theorem 2.

8.1 Answering the query $\mathsf{top}(i,j,k)$

To answer a query $\mathsf{top}(i,j,k)$ we can proceed as for query $\mathsf{sel}(i,j,k)$, until the point where we find the kth largest element in $A_v[l,r]$, let it be $A_v[o]$. Now we find all the elements $A_v[m]$ in $A_v[l,r]$ where $A_v[m] \geq A_v[o]$. With an rmq structure over A_v we can do this using Muthukrishnan's algorithm [27]: find the maximum in $A_v[l,r]$, let it be $A_v[m_1]$, then continue recursively with $A_v[l,m_1-1]$ and $A_v[m_1+1,r]$ stopping the recursion when the maximum, found at $A_v[m]$, satisfies $A_v[m] < A_v[o]$. Recall that A_v is a permutation on $O(\kappa)$ symbols and thus we can afford storing it directly (actually, it is generally part of the selection structures we use [8]). Finally, when we have the positions $m_1, \ldots, m_k$ of the top-k elements, we return $E_v[m_1], \ldots, E_v[m_k]$. The overall time is $O(\lg k/\lg\lg n + k) = O(k)$.

Note that this process delivers the top-k elements in arbitrary order. On the other hand, the set is obtained in online form: after $O(1+\lg k/\lg\lg n)$ time, each new result is delivered in $O(1)$ time. To obtain the result in sorted order and in online form, we build the structure of Brodal et al. [6] on the sets A_v, which amounts to $O(n\lg\kappa)$ further bits. With this structure, we retrieve the k highest values of $A_v[l,r]$ in time $O(k)$ and in online form, analogously as what is done with the structure of Brodal and Jørgensen [8] for the query $\mathsf{sel}(\cdot)$. This proves Theorem 3.

9 One-sided queries

We finish by showing that, at least in some restricted cases that might be of interest, the time lower bound for $\mathsf{sel}(\cdot)$ queries can be circumvented. We will design an encoding that is built for a fixed κ value and answers queries $\mathsf{sel}(1,j,\kappa)$ and $\mathsf{top}(1,j,\kappa)$. We start with the following result for $\mathsf{sel}(\cdot)$ queries, and then use the same encoding to solve $\mathsf{top}(\cdot)$ queries.

Theorem 7 *Given an array $A[1,n]$ and a value κ, there are encodings of A and κ that (1) use $n\lg\kappa + o(n\lg\kappa) + n$ bits and support $\mathsf{sel}(1,j,\kappa)$ queries in any $\omega(1)$ time, or (2) use $(1+\epsilon)n\lg\kappa$ bits and support $\mathsf{sel}(1,j,\kappa)$ queries in $O(1/\epsilon)$ time, for any constant $0 < \epsilon < 1$.*

To build this encoding, we scan the array from left to right, and keep track of the top-κ values in the prefix seen so far. At any position $j > \kappa$, if $A[j]$ is inserted into the top-κ list, then we have to remove the κth largest value in the prefix $A[1,j-1]$. The idea to solve these queries is to record the position of that leaving κth largest value, so that to solve $\mathsf{sel}(1,j,\kappa)$ we find the next $j' > j$ where the top-κ list changes, and then find the value leaving the list when $A[j']$ enters it. This one was the κth largest value in $A[1,j]$.

We wish, however, to store this information using only $O(n\lg\kappa)$ bits. The key idea is to store *colors* in $[1,\kappa]$ associated with the positions $A[j']$ where the top-κ list changes. Each element that

[12] The search for the right s can be done in constant time by checking the cases $s = \tau$ and $s = \tau - 1$, and then consulting a small precomputed table of $2^{2^{\tau-2}} = O(\sqrt{\kappa})$ entries.

A	12	18	17	20	14	19	22	11	25	21	28	16	23	13	15	24	29	27
X	1	2	3	1	-	3	2	-	3	1	1	-	2	-	-	2	2	3
P	1	1	1	1	0	1	1	0	1	1	1	0	1	0	0	1	1	1

Figure 7: Encoding of an array A as P and X, to support $\mathsf{sel}(\cdot)$ and $\mathsf{top}(\cdot)$ queries, for $\kappa = 3$.

enters the list takes the color of the element leaving it. Then, for every prefix $A[1, j]$, the rightmost positions of the κ different colors in $[1, j]$ form the top-κ list for $A[1, j]$. In particular, if $A[j']$ is of color c, then the rightmost occurrence of c in $A[1, j' - 1]$ is the position of the κth element in $A[1, j' - 1]$, that is, $\mathsf{sel}(1, j' - 1, \kappa)$ (and also $\mathsf{sel}(1, j, \kappa)$, since no changes occur in $A[j + 1, j' - 1]$).

We store a bit-vector $P[1, n]$, where $P[j] = 1$ iff a new element is inserted into the top-κ list at position j (or equivalently, the κth largest value of $A[1, j - 1]$ is deleted at position j). The first κ bits of P are 1. We encode P in $n + o(n)$ bits supporting constant-time rank and select (Lemma 1).

Let n' be the number of 1s in P. Our string of colors, $X[1, n']$, holds $X[j] = j$ for $1 \le j \le \kappa$, and $X[j] = X[\mathsf{rank}_1(P, \mathsf{sel}(1, \mathsf{select}_1(P, j) - 1), \kappa)]$ for $\kappa < j \le n'$. Basically, if $A[j]$ becomes part of the top-κ list in $A[1, j]$, and this displaces the previous top-κ element $A[i]$ of $A[1, j - 1]$, then we assign $X[j] = X[i]$. The rest of the formula accounts for the fact that X is defined only on the cells of A where the top-κ list changes, that is, where $P[j] = 1$.

Example. Figure 7 shows an example for an array $A[1, 18]$ and $\kappa = 3$. The top-κ list changes $n' = 13$ times, so we store $X[1, 13]$. The dashes in X are for illustration purposes and are not actually stored; its actual values are associated with the 1s in P. □

We encode X in $(1+o(1))n' \lg \kappa$ bits, so that select on X is supported in $O(1)$ time and access to any $X[j]$ takes any $\omega(1)$ time (Lemma 4). On top of this we add the structure of Lemma 7, which uses $O(n' \lg\lg \kappa) = o(n' \lg \kappa)$ bits[13] and supports in constant time the restricted queries $\mathsf{rank}_{X[j]}(X, j)$.

Therefore, we compute $i = \mathsf{rank}_1(P, j) + 1$, and $c = X[i]$ is the color associated with $A[j']$. Then it holds that $\mathsf{sel}(1, j, \kappa) = \mathsf{select}_1(P, \mathsf{select}_c(X, \mathsf{rank}_c(X, i) - 1))$. Thus, this operation can be supported in any $\omega(1)$ time, dominated by the time to access $X[i]$. By using a slightly larger representation for X (Lemma 6), $(1 + \epsilon)n' \lg \kappa$ bits, we obtain time $O(1/\epsilon)$ for any constant $\epsilon > 0$. Theorem 7 follows.

9.1 Solving top-κ queries

We now use the same encoding to support $\mathsf{top}(1, j, \kappa)$ queries.

Theorem 8 *Given an array $A[1, n]$ and a value κ, there is an encoding of A and κ that uses $n \lg \kappa + o(n \lg \kappa) + n$ bits and supports $\mathsf{top}(1, j, \kappa)$ queries in $O(\kappa)$ time, giving the results in unsorted order. The result can be sorted by value in $O(\kappa \lg\lg \kappa)$ time. The encoding is the same as in Theorem 7.*

[13] This is $o(n' \lg \kappa)$ only if κ is not constant, but if $\kappa = O(1)$ we can directly use the general rank operation of Lemma 4, which in this case takes constant time.

For supporting $\mathsf{top}(\cdot)$ queries we need to find, given a position $X[i]$, the rightmost occurrence preceding i of every color in $[1,\kappa]$. This can be done in $O(\kappa)$ time using the representation of Lemma 4 for X: The string is cut into chunks of size κ. Each chunk stores an *inverted list* of its contents, that is, for each color it stores an increasing list of the positions where it appears in the chunk. Constant-time access is given to any position of any list. Further, one bit-vector B_c per color c is stored, $B_c = \mathsf{0}1^{n_1^c}\mathsf{0}1^{n_2^c}\ldots\mathsf{0}1^{n_\kappa^c}$, where c appears n_j^c times in the jth chunk. Bit-vectors B_c are provided with constant-time rank and select (Lemma 1) and add up to $O(n')$ bits.

In the chunk $l = \lceil i/\kappa \rceil$ where position i belongs, we traverse all the lists of all the colors, so as to record the last occurrence of each color preceding position i. This takes time $O(\kappa)$ because there are κ positions in the chunk, thus the total length of the lists is also κ. Some colors may not occur in the chunk before position i, however. For each such color c, we find its last position in the last chunk before the current one: the starting point of the chunk l in B_c is $s = \mathsf{select}_\mathsf{0}(B_c, l)$, the number of 1s up to s is $o = s - l$, and the chunk where the oth 1 appears is $\mathsf{select}_\mathsf{1}(B_c, o) - o$. Once we find the chunk for c in constant time, we return the last position of the list of c in the chunk, which as said can be accessed in constant time as well.

By the definition of X, it is clear that the rightmost occurrences, up to position $i = \mathsf{rank}_\mathsf{1}(P, j)$, of the distinct colors, form precisely the answer to $\mathsf{top}(1, j, \kappa)$. Thus we find all those positions p in time $O(\kappa)$ and remap them to the original array with $\mathsf{select}_\mathsf{1}(P, p)$.

Note that the top-κ positions do not come sorted by value. By the same properties of X, if the first occurrence of c after $X[i]$ precedes the first occurrence of c' after $X[i]$, then the value associated with c in our answer is smaller than that associated with c', as it is replaced earlier. Thus we find the first occurrence, after i, of each color c in $[1,\kappa]$. The number r_c of times c appears up to position i is $\mathsf{select}_\mathsf{0}(B_c, l) - l$ plus the number of its occurrences up to i inside chunk l, which we have already counted. Then the position of its next occurrence in X is $p_c = \mathsf{select}_c(X, r_c + 1)$, which is computed in constant time in our representation of X (Lemma 4). Once we have the positions p_c, which are integers in $[1, n']$, we can sort them in time $O(\kappa \lg\lg \kappa)$ [1].

Actually, the space $(1 + o(1))n' \lg \kappa$ given in Lemma 4 is obtained using the chunks structure only when $\kappa = \omega(1)$. When $\kappa = O(1)$ ones uses instead Lemma 5, where operations access, rank_c, and select_c on X take constant time. In this case we simply obtain the last position of c before $X[i]$ with $\mathsf{select}_c(X, \mathsf{rank}_c(X, i-1))$, and the position following $X[i]$ with $\mathsf{select}_c(X, \mathsf{rank}_c(X, i) + 1)$, all in constant time per color. Theorem 8 follows.

10 Conclusions

We have studied for the first time the problem of encoding data structures for array range queries $\mathsf{sel}(\cdot)$ and $\mathsf{top}(\cdot)$, which return the kth largest element or all the top-k elements, respectively, of any interval $A[i, j]$. An encoding data structure cannot access the array A. We have shown that at least $n \lg k - O(n + k \lg k)$ bits are necessary for any such encoding. Further, we have given $O(n \lg \kappa)$-bit encodings that answer both queries, for any $1 \le k \le \kappa$, in optimal times $O(1 + \lg k / \lg\lg n)$ and $O(k)$, respectively.

A recent followup work [18] refines our lower bound to $(n \lg k + (k+1)n \lg(1+1/k))(1 - o(1))$ bits for $k = o(n)$, and proves it is tight up to lower-order terms by building an encoding of $n \lg \kappa + O(n)$ bits for queries with a fixed κ value. The encoding does not, however, support efficient queries; it requires $\Omega(n)$ time. In the most recent version [17], they give a slightly larger encoding using $1.5\, n \lg \kappa - \Theta(n)$ bits, which solves queries $\mathsf{top}(i, j, \kappa)$ and $\mathsf{sel}(i, j, \kappa)$ in time $O(\kappa^6 \lg^2 n\, \omega(1))$. While

still far from optimal, the time is polynomial in $\kappa \lg n$ and raises the question of what the space/time tradeoffs are when we consider the constant accompanying the $O(n \lg \kappa)$ space complexity of the encodings. Our encoding obtains optimal times, but the constant is large: $44n \lg \kappa + O(n \lg \lg \kappa)$ bits plus 32 times the space used by the extra structures [6, 8].

Acknowledgements. We thank Yakov Nekrich, who pointed us the results of Brodal et al. [6], and the anonymous referees for their suggestions.

References

[1] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

[2] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In *Proc. 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 785–794, 2009.

[3] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms (TALG)*, 10(4):article 23, 2014.

[4] D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):article 31, 2015.

[5] T. Bell, J. Cleary, and I. Witten. *Text compression.* Prentice Hall, 1990.

[6] G. S. Brodal, R. Fagerberg, M. Greve, and A. Lopez-Ortiz. Online sorted range reporting. In *Proc. 20th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 5878, pages 173–182, 2009.

[7] G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. Towards optimal range medians. *Theoretical Computer Science*, 412(24):2588–2601, 2011.

[8] G. S. Brodal and A. G. Jørgensen. Data structures for range median queries. In *Proc. 20th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 5878, pages 822–831, 2009.

[9] T. Chan and B. T. Wilkinson. Adaptive and approximate orthogonal range counting. In *Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 241–251, 2013.

[10] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[11] P. Davoodi, G. Navarro, R. Raman, and S. Srinivasa Rao. Encoding range minima and range top-2 queries. *Philosophical Transactions of the Royal Society A*, 372(20130131), 2014.

[12] J. Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 158–169, 2010.

[13] J. Fischer. Combined data structure for previous- and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.

[14] J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal of Computing*, 40(2):465–492, 2011.

[15] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.

[16] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: another virtue of wavelet trees. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 1–6, 2009.

[17] P. Gawrychowski and P. K. Nicholson. Optimal encodings for range min-max and top-k. *CoRR*, 1411.6581v2, 2015. `http://arxiv.org/abs/1411.6581v2`.

[18] P. Gawrychowski and P. K. Nicholson. Optimal encodings for range top-k, selection, and min-max. In *Proc. 42nd International Colloquium on Automata, Languages, and Programming (ICALP), Part I*, LNCS 9134, pages 593–604, 2015.

[19] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[20] R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. Srinivasa Rao. Encodings for range selection and top-k queries. In *Proc. 21st Annual European Symposium on Algorithms (ESA)*, LNCS 8125, pages 553–564, 2013.

[21] R. Grossi, A. Orlandi, R. Raman, and S. S. Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proc. 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, LIPIcs 3, pages 517–528, 2009.

[22] P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *World Wide Web Conference (WWW 2013)*, pages 583–594, 2013.

[23] A. G. Jørgensen and K. G. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 805–813, 2011.

[24] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *SIGMOD Conference*, pages 695–706. ACM, 2009.

[25] Jirí Matousek. Cutting hyperplane arrangements. *Discrete and Computational Geometry*, 6:385–406, 1991.

[26] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[27] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.

[28] G. Navarro, R. Raman, and S. Srinivasa Rao. Asymptotically optimal encodings for range selection. In *Proc. 34th Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 291–302, 2014.

[29] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):article 16, 2014.

[30] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[31] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 2(4):article 43, 2007.

[32] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.