Online Sorted Range Reporting

Gerth Stølting Brodal[1], Rolf Fagerberg[2], Mark Greve[1], and Alejandro López-Ortiz[3]

[1] MADALGO*, Dept. of Computer Science, Aarhus University, Denmark, {gerth,mgreve}@cs.au.dk

[2] Dept. of Math. and Computer Science, University of Southern Denmark, rolf@imada.sdu.dk

[3] School of Computer Science, University of Waterloo, Canada, alopez-o@uwaterloo.ca

Abstract. We study the following one-dimensional range reporting problem: On an array A of n elements, support queries that given two indices $i \leq j$ and an integer k report the k smallest elements in the subarray $A[i..j]$ in sorted order. We present a data structure in the RAM model supporting such queries in optimal $O(k)$ time. The structure uses $O(n)$ words of space and can be constructed in $O(n \log n)$ time. The data structure can be extended to solve the online version of the problem, where the elements in $A[i..j]$ are reported one-by-one in sorted order, in $O(1)$ worst-case time per element. The problem is motivated by (and is a generalization of) a problem with applications in search engines: On a tree where leaves have associated rank values, report the highest ranked leaves in a given subtree. Finally, the problem studied generalizes the classic range minimum query (RMQ) problem on arrays.

1 Introduction

In information retrieval, the basic query types are exact word matches, and combinations such as intersections of these. Besides exact word matches, search engines may also support more advanced query types like prefix matches on words, general pattern matching on words, and phrase matches. Many efficient solutions for these involve string tree structures such as tries and suffix trees, with query algorithms returning nodes of the tree. The leaves in the subtree of the returned node then represent the answer to the query, e.g. as pointers to documents.

An important part of any search engine is the ranking of the returned documents. Often, a significant element of this ranking is a query-independent precalculated rank of each document, with PageRank [1] being the canonical example. In the further processing of the answer to a tree search, where it is merged with results of other searches, it is beneficial to return the answer set ordered by

* Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

the pre-calculated rank, and even better if it is possible to generate an increasing prefix of this ordered set on demand. In short, we would like a functionality similar to storing at each node in the tree a list of the leaves in its subtree sorted by their pre-calculated rank, but without the prohibitive space cost incurred by this solution.

Motivated by the above example, we consider the following set of problems, listed in order of increasing generality. For each problem, an array $A[0..n-1]$ of n numbers is given, and the task is to preprocess A into a space-efficient data structure that efficiently supports the query stated. A query consists of two indices $i \leq j$, and if applicable also an integer k.

Sorted range reporting:
Report the elements in $A[i..j]$ in sorted order.
Sorted range selection:
Report the k smallest elements in $A[i..j]$ in sorted order.
Online sorted range reporting:
Report the elements in $A[i..j]$ one-by-one in sorted order.

Note that if the leaves of a tree are numbered during a depth-first traversal, the leaves of any subtree form a consecutive segment of the numbering. By placing leaf number i at entry $A[i]$, and annotating each node of the tree by the maximal and minimal leaf number in its subtree, we see that the three problems above generalize our motivating problems on trees. The aim of this paper is to present linear space data structures with optimal query bounds for each of these three problems. We remark that the two last problems above also form a generalization of the well-studied range minimum query (RMQ) problem [2]. The RMQ problem is to preprocess an array A such that given two indices $i \leq j$, the minimum element in $A[i..j]$ can be returned efficiently.

Contributions: We present data structures to support sorted range reporting queries in $O(j-i+1)$ time, sorted range selection queries in $O(k)$ time, and online sorted range reporting queries in worst case $O(1)$ time per element reported. For all problems the solutions take $O(n)$ words of space and can be constructed in $O(n \log n)$ time. We assume a unit-cost RAM whose operations include addition, subtraction, bitwise AND, OR, XOR, and left and right shifting and multiplication. Multiplication is not crucial to our constructions and can be avoided by the use of table lookup. By w we denote the word length in bits, and assume that $w \geq \log n$.

Outline: In the remainder of this section, we give definitions and simple constructions used extensively in the solution of all three problems. In Section 2, we give a simple solution to the sorted range reporting problem which illustrates some of the ideas used in our more involved solution of the sorted range selection problem. In Section 3, we present the main result of the paper which is our solution to the sorted range selection problem. Building on the solution to the previous problem, we give a solution to the online sorted range reporting problem in Section 4.

We now give some definitions and simple results used by our constructions.

Essential for our constructions to achieve overall linear space, is the standard trick of packing multiple equally sized items into words.

Lemma 1. *Let X be a two-dimensional array of size $s \times t$ where each entry $X[i][j]$ consists of $b \leq w$ bits for $0 \leq i < s$ and $0 \leq j < t$. We can store this using $O(stb + w)$ bits, and access entries of X in $O(1)$ time.*

We often need an array of s secondary arrays, where the secondary arrays have variable lengths. For this, we simply pad these to have the length of the longest secondary array. For our applications we also need to be able to determine the length of a secondary array. Since a secondary array has length at most t, we need $\lfloor \log t \rfloor + 1$ bits to represent the length. By packing the length of each secondary array as in Lemma 1, we get a total space usage of $O(stb + s \log t + w) = O(stb + w)$ bits of space, and lookups can still be performed in $O(1)$ time. We summarize this in a lemma.

Lemma 2. *Let X be an array of s secondary arrays, where each secondary array $X[i]$ contains up to t elements, and each entry $X[i][j]$ in a secondary array $X[i]$ takes up $b \leq w$ bits. We can store this using $O(stb + w)$ bits of space and access an entry or length of a secondary array in $O(1)$ time.*

Complete binary trees: Throughout this paper $\mathcal{T}$ will denote a complete binary tree with n leaves where n is a power of two. We number the nodes as in binary heaps: the root has index 1, and an internal node with index x has left child $2x$, right child $2x + 1$ and parent $\lfloor x/2 \rfloor$. Below, we identify a node by its number.

We let $\mathcal{T}_u$ denote the subtree of $\mathcal{T}$ rooted at node u, and $h(\mathcal{T}_u)$ the *height* of the subtree $\mathcal{T}_u$, with heights of leaves defined to be 0. The height of a node $h(u)$ is defined to be $h(T_u)$, and *level* ℓ of $\mathcal{T}$ is defined to be the set of nodes with height ℓ. The height $h(u)$ of a node u can be found in $O(1)$ time as $h(\mathcal{T}) - d(u)$, where $d(u)$ is the *depth* of node u. The depth $d(u)$ of a node u can be found by computing the index of the most significant bit set in u (the root has depth 0). This can be done in $O(1)$ time and space using multiplications [3], or in $O(1)$ time and $O(n)$ space without multiplications, by using a lookup table mapping every integer from $0 \ldots 2n - 1$ to the index of its most significant bit set.

To navigate efficiently in $\mathcal{T}$, we explain a few additional operations that can be performed in $O(1)$ time. First we define $\text{anc}(u, \ell)$ as the ℓ'th ancestor of the node u, where $\text{anc}(u, 0) = u$ and $\text{anc}(u, \ell) = \text{parent}(\text{anc}(u, \ell - 1))$. Note that $\text{anc}(u, \ell)$ can be computed in $O(1)$ time by right shifting u ℓ times. Finally, we note that we can find the leftmost leaf in a subtree $\mathcal{T}_u$ in $O(1)$ time by left shifting u $h(u)$ times. Similarly we can find the rightmost leaf in a subtree $\mathcal{T}_u$ in $O(1)$ time by left shifting u $h(u)$ times, and setting the bits shifted to 1 using bitwise OR.

LCA queries in complete binary trees: An important component in our construction is finding the lowest common ancestor (LCA) of two nodes at the same depth in a complete binary tree in $O(1)$ time. This is just $\text{anc}(u, \ell)$ where ℓ is the index of the most significant bit set in the word u XOR v.

Selection in sorted arrays: The following theorem due to Frederickson and Johnson [4] is essential to our construction in Section 3.4.

Theorem 1. *Given m sorted arrays, we can find the overall k smallest elements in time $O(m + k)$.*

Proof. By Theorem 1 in [4] with $p = \min(m, k)$ we can find the k'th smallest element in time $O(m + p \log(k/p)) = O(m + k)$ time. When we have the k'th smallest element x, we can go through each of the m sorted arrays and select elements that are $\leq x$ until we have collected k elements or exhausted all lists. This takes time $O(m + k)$. $\square$

2 Sorted range reporting

In this section, we give a simple solution to the sorted range reporting problem with query time $O(j - i + 1)$. The solution introduces the concept of *local rank labellings* of elements, and shows how to combine this with radix sorting to answer sorted range reporting queries. These two basic techniques will be used in a similar way in the solution of the more general sorted range selection problem in Section 3.

We construct local rank labellings for each r in $0 \ldots \lceil \log \log n \rceil$ as follows (the rank of an element x in a set X is defined as $|\{y \in X \mid y < x\}|$). For each r, the input array is divided into $\lceil n/2^{2^r} \rceil$ consecutive subarrays each of size 2^{2^r} (except possibly the last subarray), and for each element $A[x]$ the r'th local rank labelling is defined as its rank in the subarray $A[\lfloor x/2^{2^r} \rfloor 2^{2^r} .. (\lfloor x/2^{2^r} \rfloor + 1)2^{2^r} - 1]$. Thus, the r'th local rank for an element $A[x]$ consists of 2^r bits. Using Lemma 1 we can store all local rank labels of length 2^r using space $O(n2^r + w)$ bits. For all $\lceil \log \log n \rceil$ local rank labellings, the total number of bits used is $O(w \log \log n + n \log n) = O(nw)$ bits. All local rank labellings can be built in $O(n \log n)$ time while performing mergesort on A. The r'th structure is built by writing out the sorted lists, when we reach level 2^r. Given a query for $k = j - i + 1$ elements, we find the r for which $2^{2^{r-1}} < k \leq 2^{2^r}$. Since each subarray in the r'th local rank labelling contains 2^{2^r} elements, we know that i and j are either in the same or in two consecutive subarrays. If i and j are in consecutive subarrays, we compute the start index of the subarray where the index j belongs, i.e. $x = \lfloor j/2^{2^r} \rfloor 2^{2^r}$. We then radix sort the elements in $A[i..x-1]$ using the local rank labels of length 2^r. This can be done in $O(k)$ time using two passes by dividing the 2^r bits into two parts of 2^{r-1} bits each, since $2^{2^{r-1}} < k$. Similarly we radix sort the elements from $A[x..j]$ using the labels of length 2^r in $O(k)$ time. Finally, we merge these two sorted sequences in $O(k)$ time, and return the k smallest elements. If i and j are in the same subarray, we just radix sort $A[i..j]$.

3 Sorted range selection

Before presenting our solution to the sorted range selection problem we note that if we do not require the output of a query to be sorted, it is possible to get a conceptually simple solution with $O(k)$ query time using $O(n)$ preprocessing time and space. First build a data structure to support range minimum queries in $O(1)$ time using $O(n)$ preprocessing time and space [2]. Given a query on $A[i..j]$ with parameter k, we lazily build the Cartesian tree [5] for the subarray $A[i..j]$ using range minimum queries. The Cartesian tree is defined recursively by choosing the root to be the minimum element in $A[i..j]$, say $A[x]$, and recursively constructing its left subtree using $A[i..x-1]$ and its right subtree using $A[x+1..j]$. By observing that the Cartesian tree is a heap-ordered binary tree storing the elements of $A[i..j]$, we can use the heap selection algorithm of Frederickson [6] to select the k smallest elements in $O(k)$ time. Thus, we can find the k smallest elements in $A[i..j]$ in *unsorted* order in $O(k)$ time.

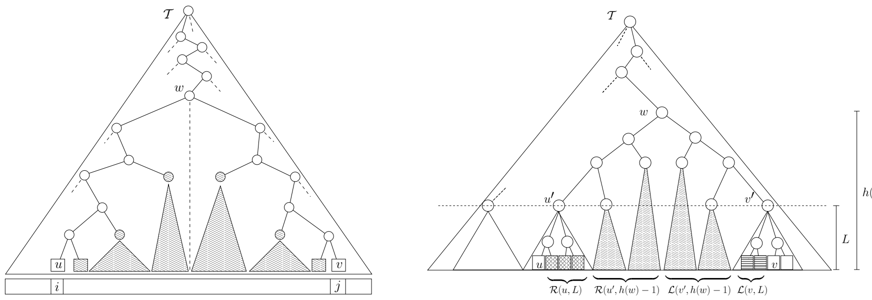
In the remainder of this section, we present our data structure for the sorted range selection problem. The data structure supports queries in $O(k)$ time, uses $O(n)$ words of space and can be constructed in $O(n \log n)$ time. When answering a query we choose to have our algorithm return the indices of the elements of the output, and not the actual elements. Our solution consists of two separate data structures for the cases where $k \leq \lfloor \log n/(2\log\log n)^2 \rfloor$ and $k > \lfloor \log n/(2\log\log n)^2 \rfloor$. The data structures are described in Sections 3.3 and 3.4 respectively. In Sections 3.1 and 3.2, we present simple techniques used by both data structures. In Section 3.1, we show how to decompose sorted range selection queries into a constant number of smaller ranges, and in Section 3.2 we show how to answer a subset of the queries from the decomposition by precomputing the answers. In Section 3.5, we describe how to build our data structures.

3.1 Decomposition of queries

For both data structures described in Sections 3.3 and 3.4, we consider a complete binary tree $\mathcal{T}$ with the leaves storing the input array A. We assume without loss of generality that the size n of A is a power of two. Given an index i for $0 \leq i < n$ into A we denote the corresponding leaf in $\mathcal{T}$ as $\text{leaf}[i] = n + i$. For a node x in $\mathcal{T}$, we define the canonical subset C_x as the leaves in $\mathcal{T}_x$. For a query range $A[i..j]$, we let $u = \text{leaf}[i]$, $v = \text{leaf}[j]$, and $w = \text{LCA}(u, v)$. On the two paths from the two leaves to their LCA w we get at most $2\log n$ disjoint canonical subsets, whose union represents all elements in $A[i..j]$, see Figure 1(a).

For a node x we define the sets $\mathcal{R}(x, \ell)$ (and $\mathcal{L}(x, \ell)$) as the union of the canonical subsets of nodes rooted at the right (left for $\mathcal{L}$) children of the nodes on the path from x to the ancestor of x at level ℓ, but excluding the canonical subsets of nodes that are on this path, see Figure 1(a). Using the definition of the sets $\mathcal{R}$ and $\mathcal{L}$, we see that the set of leaves strictly between leaves u and v is equal to $\mathcal{R}(u, h(w)-1) \cup \mathcal{L}(v, h(w)-1)$. In particular, we will decompose queries as shown in Figure 1(b). Assume L is a fixed level in $\mathcal{T}$, and that the LCA w is at a level $> L$. Define the ancestors $u' = \text{anc}(u, L)$ and $v' = \text{anc}(v, L)$ of u and v at level L. We observe that the query range, i.e. the set of leaves strictly between leaves u and v can be represented as $\mathcal{R}(u, L) \cup \mathcal{R}(u', h(w)-1) \cup \mathcal{L}(v', h(w)-1) \cup \mathcal{L}(v, L)$. In the case that the LCA w is below or at level L, the set of leaves strictly between u and v is equal to $\mathcal{R}(u, h(w)-1) \cup \mathcal{L}(v, h(w)-1)$.

Hence to answer a sorted range selection query on k elements, we need only find the k smallest elements in sorted order of each of these at most four sets,



(a) The shaded parts and the leaves u and v cover the query range.

(b) Decomposition of a query range into four smaller ranges by cutting the tree at level L.

Fig. 1. Query decomposition.

and then select the k overall smallest elements in sorted order (including the leaves u and v). Assuming we have a sorted list over the k smallest elements for each set, this can be done in $O(k)$ time by merging the sorted lists (including u and v), and extracting the k smallest of the merged list. Thus, assuming we have a procedure for finding the k smallest elements in each set in $O(k)$ time, we obtain a general procedure for sorted range queries in $O(k)$ time.

The above decomposition motivates the definition of *bottom* and *top* queries relative to a fixed level L. A *bottom* query on k elements is the computation of the k smallest elements in sorted order in $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where x is a leaf and $\ell \leq L$. A *top* query on k elements is the computation of the k smallest elements in sorted order in $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where x is a node at level L. From now on we only state the level L where we cut $\mathcal{T}$, and then discuss how to answer bottom and top queries in $O(k)$ time, i.e. implicitly assuming that we use the procedure described in this section to decompose the original query, and obtain the final result from the answers to the smaller queries.

3.2 Precomputing answers to queries

We now describe a simple solution that can be used to answer a subset of possible queries, where a query is the computation of the k smallest elements in sorted order of $\mathcal{R}(x, \ell)$ or $\mathcal{L}(x, \ell)$ for some node x and a level ℓ, where $\ell \geq h(x)$. The solution works by precomputing answers to queries. We apply this solution later on to solve some of the cases that we split a sorted range selection query into.

Let x be a fixed node, and let y and K be fixed integer thresholds. We now describe how to support queries for the k smallest elements in sorted order of $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where $h(x) \leq \ell \leq y$ and $k \leq K$. We precompute the answer to all queries that satisfy the constraints set forth by K and y by storing two arrays R_x and L_x for the node x. In $R_x[\ell]$, we store the indices of the K smallest leaves in sorted order of $\mathcal{R}(x, \ell)$. The array L_x is defined symmetrically. We summarize this solution in a lemma, where we also discuss the space usage and how to represent indices of leaves.

Lemma 3. *For a fixed node x and fixed parameters y and K, where $y \geq h(x)$, we can store R_x and L_x using $O(Ky^2 + w)$ bits of space. Queries for the k smallest elements in sorted order in $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) can be supported in time $O(k)$ provided $k \leq K$ and $h(x) \leq \ell \leq y$.*

Proof. By storing indices relative to the index of the rightmost leaf in $\mathcal{T}_x$, we only need to store y bits per element in R_x and L_x. We can store the two arrays R_x and L_x with a space usage of $O(Ky^2+w)$ bits using Lemma 2. When reading an entry, we can add the index of the rightmost leaf in $\mathcal{T}_x$ in $O(1)$ time. The k smallest elements in $\mathcal{R}(x, \ell)$ can be reported by returning the k first entries in $R_x[\ell]$ (and similarly for $L_x[\ell]$). □

3.3 Solution for $k \leq \lfloor \log n/(2 \log \log n)^2 \rfloor$

In this section, we show how to answer queries for $k \leq \lfloor \log n/(2 \log \log n)^2 \rfloor$. Having discussed how to decompose a query into bottom and top queries in Section 3.1, and how to answer queries by storing precomputed answers in Section 3.2, this case is now simple to explain.

Theorem 2. *For $k \leq \lfloor \log n/(2 \log \log n)^2 \rfloor$, we can answer sorted range selection queries in $O(k)$ time using $O(n)$ words of space.*

Proof. We cut $\mathcal{T}$ at level $2\lfloor \log \log n \rfloor$. A bottom query is solved using the construction in Lemma 3 with $K = \lfloor \log n/(2 \log \log n)^2 \rfloor$ and $y = 2\lfloor \log \log n \rfloor$. The choice of parameters is justified by the fact that we cut $\mathcal{T}$ at level $2\lfloor \log \log n \rfloor$, and by assumption $k \leq \lfloor \log n/(2 \log \log n)^2 \rfloor$. As a bottom query can be on any of the n leaves, we must store arrays L_x and R_x for each leaf as described in Lemma 3. All R_x structures are stored in one single array which is indexed by a leaf x. Using Lemma 3 the space usage for all R_x becomes $O(n(w + \lfloor \log n/(2 \log \log n)^2 \rfloor (2\lfloor \log \log n \rfloor)^2) = O(n(w + \log n)) = O(nw)$ bits (and similarly for L_x). For the top query, we for all nodes x at level $2\lfloor \log \log n \rfloor$ use the same construction with $K = \lfloor \log n/(2 \log \log n)^2 \rfloor$ and $y = \log n$. As we only have $n/2^{2\lfloor \log \log n \rfloor} = \Theta(n/(\log n)^2)$ nodes at level $2\lfloor \log \log n \rfloor$, the space usage becomes $O(\frac{n}{(\log n)^2}(w + \lfloor \log n/(2 \log \log n)^2 \rfloor (\log n)^2)) = O(n(w + \log n)) = O(nw)$ bits (as before we store all the R_x structures in one single array, which is indexed by a node x, and similarly for L_x). For both query types the $O(k)$ time bound follows from Lemma 3. □

3.4 Solution for $k > \lfloor \log n/(2 \log \log n)^2 \rfloor$

In this case, we build $O(\log \log n)$ different structures each handling some range of the query parameter k. The r'th structure is used to answer queries for $2^{2^r} < k \leq 2^{2^{r+1}}$. Note that no structure is required for r satisfying $2^{2^{r+1}} \leq \lfloor \log n/(2 \log \log n)^2 \rfloor$ since this is handled by the case $k \leq \lfloor \log n/(2 \log \log n)^2 \rfloor$.

The r'th structure uses $O(w + n(2^r + w/2^r))$ bits of space, and supports sorted range selection queries in $O(2^{2^r} + k)$ time for $k \leq 2^{2^{r+1}}$. The total space usage of the $O(\log\log n)$ structures becomes $O(w \log\log n + n \log n + nw)$ bits, i.e. $O(n)$ words, since $r \le \lceil \log\log n \rceil$. Given a sorted range selection query, we find the right structure. This can be done in done in $o(k)$ time. Finally, we query the r'th structure in $O(2^{2^r} + k) = O(k)$ time, since $2^{2^r} \le k$.

In the r'th structure, we cut $\mathcal{T}$ at level 2^r and again at level $7 \cdot 2^r$. By generalizing the idea of decomposing queries as explained in Section 3.1, we split the original sorted range selection query into three types of queries, namely *bottom*, *middle* and *top* queries. We define u' as the ancestor of u at level 2^r and u'' as the ancestor of u at level $7 \cdot 2^r$. We define v' and v'' in the same way for v. When the level of $w = \mathrm{LCA}(u, v)$ is at a level $> 7 \cdot 2^r$, we see that the query range (i.e. all the leaves strictly between the leaves u and v) is equal to $\mathcal{R}(u, 2^r) \cup \mathcal{R}(u', 7 \cdot 2^r) \cup \mathcal{R}(u'', h(w) - 1) \cup \mathcal{L}(v'', h(w) - 1) \cup \mathcal{L}(v', 7 \cdot 2^r) \cup \mathcal{L}(v, 2^r)$. In the case that w is below or at level $7 \cdot 2^r$, we can use the decomposition as in Section 3.1. In the following we focus on describing how to support each type of query in $O(2^{2^r} + k)$ time.

Bottom query: A bottom query is a query on a leaf u for $\mathcal{R}(u, \ell)$ (or $\mathcal{L}(u, \ell)$) where $\ell \le 2^r$. For all nodes x at level 2^r, we store an array S_x containing the canonical subset C_x in sorted order. Using Lemma 1 we can store the S_x arrays for all x using $O(n2^r + w)$ bits as each leaf can be indexed with 2^r bits (relative to the leftmost leaf in $\mathcal{T}_x$). Now, to answer a bottom query we make a linear pass through the array $S_{\mathrm{anc}(u,2^r)}$ discarding elements that are not within the query range. We stop once we have k elements, or we have no more elements left in the array. This takes $O(2^{2^r} + k)$ time.

Top query: A top query is a query on a node x at level $7 \cdot 2^r$ for $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where $7 \cdot 2^r < \ell \le \log n$. We use the construction in Lemma 3 with $K = 2^{2^{r+1}}$ and $y = \log n$. We have $n/(2^{7 \cdot 2^r})$ nodes at level $7 \cdot 2^r$, so to store all structures at this level the total number of bits of space used is

$$O\left(\frac{n}{2^{7\cdot 2^r}}(w + 2^{2^{r+1}}(\log n)^2)\right) = O\left(n\frac{w}{2^r} + \frac{n}{2^{5\cdot 2^r}}(\log n)^2\right)$$
$$= O\left(n\frac{w}{2^r} + \frac{n}{\left\lfloor \log n/(2\log\log n)^2 \right\rfloor^{5/2}}(\log n)^2\right) = O\left(n\frac{w}{2^r}\right) ,$$

where we used that $\lfloor \log n/(2\log\log n)^2 \rfloor < k \le 2^{2^{r+1}}$. By Lemma 3 a top query takes $O(k)$ time.

Middle query: A middle query is a query on a node z at level 2^r for $\mathcal{R}(z, \ell)$ (or $\mathcal{L}(z, \ell)$) with $2^r < \ell \le 7 \cdot 2^r$. For all nodes x at level 2^r, let $\min_x = \min C_x$. The idea in answering middle queries is as follows. Suppose we could find the nodes at level 2^r corresponding to the up to k smallest $\min_x$ values within the query range. To answer a middle query, we would only need to extract the k overall smallest elements from the up to k corresponding sorted S_x arrays of the nodes, we just found. The insight is that both subproblems mentioned can be solved using Theorem 1 as the key part. Once we have the k smallest elements in the middle query range, all that remains is to sort them.

We describe a solution in line with the above idea. For each node x at levels 2^r to $7 \cdot 2^r$, we have a sorted array $\mathcal{M}_x^r$ of all nodes x' at level 2^r in $\mathcal{T}_x$ sorted with respect to the $\min_{x'}$ values. To store the $\mathcal{M}_x^r$ arrays for all x, the space required is $O(\frac{n}{2^{2^r}} \cdot 6 \cdot 2^r) = O(\frac{n}{2^r})$ words (i.e. $O(n\frac{w}{2^r}$ bits), since we have $\frac{n}{2^{2^r}}$ nodes at level 2^r, and each such node will appear $7 \cdot 2^r - 2^r = 6 \cdot 2^r$ times in an $\mathcal{M}_x^r$ array (and to store the index of a node we use a word).

To answer a middle query for the k smallest elements in $\mathcal{R}(z, \ell)$, we walk $\ell - 2^r$ levels up from z while collecting the $\mathcal{M}_x^r$ arrays for the nodes x whose canonical subset is a part of the query range (at most $6 \cdot 2^r$ arrays since we collect at most one per level). Using Theorem 1 we select the k smallest elements from the $O(2^r)$ sorted arrays in $O(2^r + k) = O(k)$ time (note that there may not be k elements to select, so in reality we select up to k elements). This gives us the k smallest $\min_{x'}$ values of the nodes $x'_1, x'_2, \ldots, x'_k$ at level 2^r that are within the query range. Finally, we select the k overall smallest elements of the sorted arrays $S_{x'_1}, S_{x'_2}, \ldots, S_{x'_k}$ in $O(k)$ time using Theorem 1. This gives us the k smallest elements of $\mathcal{R}(z, \ell)$, but not in sorted order. We now show how to sort these elements in $O(k)$ time. For every leaf u, we store its local rank relative to $C_{u''}$, where u'' is the the ancestor of u at level $7 \cdot 2^r$. Since each subtree $\mathcal{T}_{u''}$ contains $2^{7 \cdot 2^r}$ leaves, we need $7 \cdot 2^r$ bits to index a leaf (relative to the leftmost leaf in $\mathcal{T}_{u''}$). We store all local rank labels of length $7 \cdot 2^r$ in a single array, and using Lemma 1 the space usage becomes $O(n2^r + w)$ bits. Given $O(k)$ leaves from C_x for a node x at level $7 \cdot 2^r$, we can use the local rank labellings of the leaves of length $7 \cdot 2^r$ bits to radix sort them in $O(k)$ time (for the analysis we use that $2^{2^r} < k$). This completes how to support queries.

3.5 Construction

In this section, we show how to build the data structures in Sections 3.3 and 3.4 in $O(n \log n)$ time using $O(n)$ extra words of space. The structures to be created for node x are a subset of the possible structures S_x, $\mathcal{M}_x^r$, $R_x[\ell]$, $L_x[\ell]$ (where ℓ is a level above x), and the local rank labellings. In total, the structures to be created store $O(n\frac{\log n}{\log\log n})$ elements which is dominated by the number of elements stored in the R_x and L_x structures for all leaves in Section 3.3. The general idea in the construction is to perform mergesort bottom up on $\mathcal{T}$ (level-by-level) starting at the leaves. The time spent on mergesort is $O(n \log n)$, and we use $O(n)$ words of space for the mergesort as we only store the sorted lists for the current and previous level. Note that when visiting a node x during mergesort the set C_x has been sorted, i.e. we have computed the array S_x. The structures S_x and $\mathcal{M}_x^r$ will be constructed while visiting x during the traversal of $\mathcal{T}$, while $R_x[\ell]$ and $L_x[\ell]$ will be constructed at the ancestor of x at level ℓ. As soon as a set has been computed, we store it in the data structure, possibly in a packed manner. For the structures in Section 3.3, when visiting a node x at level $\ell \leq 2\lfloor \log\log n \rfloor$ we compute for each leaf z in the right subtree of x the structure $R_z[\ell] = R_z[\ell - 1]$ (where $R_z[0] = \emptyset$), and the structure $L_z[\ell]$ containing the (up to) $\lfloor \log n/(2\log\log n)^2 \rfloor$ smallest elements in sorted order of $L_z[\ell-1] \cup S_{2x}$. Both structures can be computed in time $O(\lfloor \log n/(2\log\log n)^2 \rfloor)$. Symmetrically, we compute the same structures for all leaves z in the left subtree of x. In the case that x is at level $\ell > 2\lfloor \log\log n \rfloor$, we compute for each node z at level $2\lfloor \log\log n \rfloor$

in the right subtree of x the structure $R_z[\ell] = R_z[\ell-1]$ (where $R_z[2\lfloor\log\log n\rfloor] = \emptyset$), and the structure $L_z[\ell]$ containing the $\lfloor \log n/(2\log\log n)^2 \rfloor$ smallest elements in sorted order of $L_z[\ell-1] \cup S_{2x}$. Both structures can be computed in time $O(\lfloor \log n/(2\log\log n)^2 \rfloor)$. Symmetrically, we compute the same structures for all nodes z at level $2\lfloor\log\log n\rfloor$ in the left subtree of x. For the structures in Section 3.4, when visiting a node x we first decide in $O(\log\log n)$ time if we need to compute any structures at x for any r. In the case that x is a node at level 2^r, we store $S_x = C_x$ and $\mathcal{M}_x^r = \min C_x$. For x at level $2^r < \ell \leq 7\cdot 2^r$ we store $\mathcal{M}_x^r = \mathcal{M}_{2x}^r \cup \mathcal{M}_{2x+1}^r$. This can be computed in time linear in the size of $\mathcal{M}_x^r$. In the case that x is a node at level $7\cdot 2^r$, we store the local rank labelling for each leaf in $\mathcal{T}_x$ using the sorted C_x list. For x at level $\ell > 7\cdot 2^r$, we compute for each z at level $7\cdot 2^r$ in the right subtree of x the structure $R_z[\ell] = R_z[\ell-1]$ (where $R_z[7\cdot 2^r] = \emptyset$), and the structure $L_z[\ell]$ containing the $2^{2^{r+1}}$ smallest elements in sorted order of $L_z[\ell-1] \cup S_{2x}$. Both structures can be computed in time $O(2^{2^{r+1}})$. Symmetrically, we compute the same structures for all nodes z at level $7\cdot 2^r$ in the left subtree of x. Since all structures can be computed in time linear in the size and that we have $O(n\frac{\log n}{\log\log n})$ elements in total, the overall construction time becomes $O(n\log n)$.

4 Online sorted range reporting

We now describe how to extend the solution for the sorted range selection problem from Section 3 to a solution for the online sorted range reporting problem. We solve the problem by performing a sequence of sorted range selection queries Q_y with indices i and j and $k = 2^y$ for $y = 0, 1, 2, \ldots$. The initial query to the range $A[i..j]$ is Q_0. Each time we report an element from the current query Q_y, we spend $O(1)$ time building part of the next query Q_{y+1} so that when we have exhausted Q_y, we will have finished building Q_{y+1}. Since we report the 2^{y-1} largest elements in Q_y (the 2^{y-1} smallest are reported for $Q_0, Q_1, \ldots, Q_{y-1}$), we can distribute the $O(2^{y+1})$ computation time of Q_{y+1} over the 2^{y-1} reportings from Q_y. Hence the query time becomes $O(1)$ worst-case per element reported.

References

1. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (1999)
2. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing **13**(2) (1984) 338–355
3. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. **47**(3) (1993) 424–436
4. Frederickson, G.N., Johnson, D.B.: The complexity of selection and ranking in $X+Y$ and matrices with sorted columns. J. Comput. Syst. Sci. **24**(2) (1982) 197–208
5. Vuillemin, J.: A unifying look at data structures. CACM **23**(4) (1980) 229–239
6. Frederickson, G.N.: An optimal algorithm for selection in a min-heap. Inf. Comput. **104**(2) (1993) 197–214