On (Dynamic) Range Minimum Queries in External Memory

Lars Arge[1], Johannes Fischer[2], Peter Sanders[2], and Nodari Sitchinava[2]

[1] MADALGO*, Aarhus University, Aarhus, Denmark
large@madalgo.au.dk
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
{johannes.fischer,sanders}@kit.edu, nodari@ira.uka.de

Abstract. We study the one-dimensional range minimum query (RMQ) problem in the external memory model. We provide the first space-optimal solution to the batched static version of the problem. On an instance with N elements and Q queries, our solution takes $\Theta(\text{sort}(N+Q)) = \Theta\left(\frac{N+Q}{B}\log_{M/B}\frac{N+Q}{B}\right)$ I/O complexity and $O(N+Q)$ space, where M is the size of the main memory and B is the block size. This is a factor of $O(\log_{M/B} N)$ improvement in space complexity over the previous solutions. We also show that an instance of the batched *dynamic* RMQ problem with N updates and Q queries can be solved in $O\left(\frac{N+Q}{B}\log^2_{M/B}\frac{N+Q}{B}\right)$ I/O complexity and $O(N+Q)$ space.

1 Introduction

Static one-dimensional range minimum queries (RMQs) are defined as follows: *Given an array $A[1,N]$ of N elements from a totally ordered universe $\mathcal{U}$, and a query in the form* RangeMin(i,j)*, return the index of the smallest element in the subarray $A[i,j]$.*

RMQs have a wide range of applications in many areas of data structures and algorithms, for example in data compression, text indexing, and graph algorithms. For more applications, we refer the interested reader to a recent article [8], which also contains optimal solutions for one-dimensional static RMQs in the RAM model. Research on RMQs remains a hot topic, see for example a recent invited talk by Raman [12].

In this paper we consider the external memory (EM) model [1], where performance is measured by the number of block transfers (I/Os) of B words each, and where the internal memory size is limited to M words. In this setting, in case of the *online* RMQ problem (meaning that each query must be answered immediately as it arrives) all previous $O(1)$ time, linear space *static* internal memory solutions [8] are also optimal in the EM model, since each query must spend at least one I/O to report the output. However, if we consider the *offline* version of the problem (also often referred to as *batched*), where the initial array and all

* MADALGO is a center of the Danish National Research Foundation

queries are specified in advance and queries can be answered in arbitrary order, we can achieve better amortized bounds. In particular, Chiang et al. [7] showed that Q queries in the *static* batched range minima problem can been answered with $O((n+q)\log_m(n+q)) = O(\text{sort}(N+Q))$ I/Os. (Following the common notation in the EM literature, we define $n = N/B$, $q = Q/B$ and $m = M/B$.) Their solution takes $O(Q + N\log_m N)$ space. Batched range minima arise in suffix sorting [11] (if one also wants to find longest common prefixes) and other stringology problems, for example in string mining tasks [9].

We also consider a *dynamic* scenario, where N updates (insertions and deletion of elements) to the underlying array are arbitrarily interspersed with the Q queries. In this setting, it is impossible to achieve constant time per operation [10]: we can sort N items by first inserting them into the array, and then repeatedly querying for and removing the minimum of the entire array. This shows that online dynamic RMQs are at least as hard as priority queues.

1.1 Our Contributions

We formulate the *dynamic batched range minima problem* as follows: *Given a batch of N updates (*Insert*/*Delete *operations) affecting a dynamically changing "structure" over a universe $\mathcal{U}$ and Q* RangeMin *operations, report answers to all* RangeMin *queries. Answers to queries may arrive in arbitrary order, but must be correct based on the state of the "structure" at the time of the query.*

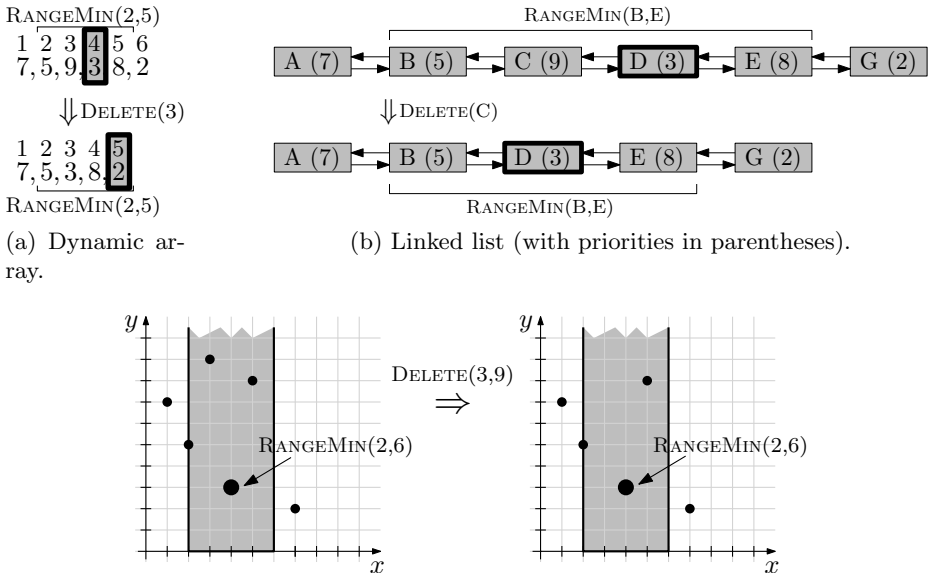
Because there is no definite concept on what the dynamization of a static array should be, in Section 2 we explore three possible versions of the underlying "structure" in the above problem definition: a linked list, a dynamic array and a point set. In section 4.2 we present a solution to the third version, which, at a first glance, seems to be the simplest to reason about in the EM model. However, in Section 5 we show that we can reduce one version to another in $O(\text{sort}(N+Q))$ I/Os. We conjecture that $\Omega(\text{sort}(N+Q))$ I/Os is the lower bound even for the static batched range minima problem, i.e., that our reductions are tight and that the three versions are equivalent.

In addition, in Section 4.1 we improve the solution of Chiang et al. [7] and present the first *linear* space solution to the *static* batched RMQ problem. Our solution also implies a linear space *parallel* I/O-efficient solution to the RMQ problem in the parallel external memory (PEM) model [3].

2 Different Scenarios for Dynamic RMQ

In general, there are different ways to think about how the Insert/Delete operations affect the array, and what the answers to RangeMin queries should be. This section is meant to explain their differences. See also Fig. 1.

Dynamic Array Version. This version most closely resembles the static version: we identify elements by their rank (number of elements, including itself, to the left) in the array at the time of the operation. Hence, although the elements



(a) Dynamic array.

(b) Linked list (with priorities in parentheses).

(c) Point grid.

Fig. 1: Different versions of dynamic range minimum queries. Range Minima are shown in bold.

are stored in a dynamic data structure, we can treat them as if they were stored in an array, and consequently denote them by $A[i]$.

Insert(i, x)**:** insert a new element with value $x \in \mathcal{U}$ right after position i into A; all positions originally after i will be shifted right by one element: $A \leftarrow A[1, i] \cdot [x] \cdot A[i + 1, N]$.

Delete(i)**:** delete the element at position i from A; all positions originally after i will be shifted left by one element: $A \leftarrow A[1, i - 1] \cdot A[i + 1, N]$.

RangeMin(i, j)**:** return (the position of) the minimum in $A[i, j]$, as in the static setting.

Though closely matching the static scenario, this version actually turns out to be most difficult to deal with: the intuitive reason for this is that the indices specify *ranks* in a dynamically changing array, and hence first need to be *unranked* (selected) before knowing which object they affect. This issue will be addressed in Sect. 5.2.

Linked List Version. Elements are part of a doubly linked list, and are identified by handles (pointers to the elements). Each list element v has an associated priority (value) $p_v \in \mathcal{U}$.

Insert(v, u, p_v)**:** Insert a new element v with priority $p_v \in \mathcal{U}$ right after element u: if the list A was originally $A_1 \rightsquigarrow u \rightarrow w \rightsquigarrow A_2$, it now becomes $A_1 \rightsquigarrow u \rightarrow v \rightarrow w \rightsquigarrow A_2$.

DELETE(v)**:** Delete the element v from A: if the list A was originally $A_1 \rightsquigarrow u \to v \to w \rightsquigarrow A_2$, it now becomes $A_1 \rightsquigarrow v \to w \rightsquigarrow A_2$.

RANGEMIN(u, v)**:** return (a pointer to the element with) the minimum priority among all elements that are currently between u and v (both inclusively).

Geometric Version. Elements are pairs of the form (x, y), and we think of them as being points on the plane. We denote the set of all points currently stored as S.

INSERT(x, y)**:** Insert a new point (x, y) into S.

DELETE(x, y)**:** Delete the point (x, y) from S.

RANGEMIN(x_1, x_2)**:** return the point (x, y) such that $x_1 \leq x \leq x_2$, and y is minimum among all those points (or return only the y-value of that point).

3 Simple Dynamic Internal Memory Algorithms

As a warmup, in this section we present a fully dynamic data structure for RMQs in the internal memory (RAM/comparison) models. We show it primarily because it forms the basis of all our EM data structures. Note that distinguishing between offline and online queries is irrelevant in internal memory.

Linked List Version. We maintain a balanced binary tree over the linked list. The leaves of the tree store the elements of the linked list, and internal nodes correspond to ranges of consecutive elements of the list. Each internal node v stores a pointer to a leaf μ in the subtree rooted at v with minimum priority; we denote that pointer by μ_v. At any time, the size of the data structure is linear with the number of nodes present in the list. See also Fig. 2.

To perform INSERT or DELETE operations, we go directly to the node specified by the operation, and traverse the path up to the root, updating the values μ_x for each node x on the path bottom up. To perform a query RANGEMIN(u, v), we traverse the two paths from u (resp. v) to the root until they meet. From there, we go back down to u (resp. v) and report the leaf μ_x with the smallest priority among those nodes x hanging off to the right (resp. left) of those paths.

The time for each operation is $O(\log N)$, where N is the number of elements in the linked list at the time of the operation.

Dynamic Array Version. The difference to the list version is that the queries do not specify pointers to the leaves of the tree, but rather ranks in the linked list. So the elements need to be first unranked in the list, which can be achieved by a top-down search in the tree. To support this search, we augment each node v with the size of the subtree rooted at v. Then a top-down traversal easily locates an element in $O(\log N)$ time given its rank. Having identified the elements, queries are processed exactly as in the linked list version. If the RANGEMIN query asks for the *position* of the minimum, we finally have to rank the minimum element by a bottom-up traversal. The subtree sizes are also easily updated during insertions/deletions and when doing rebalancing operations.

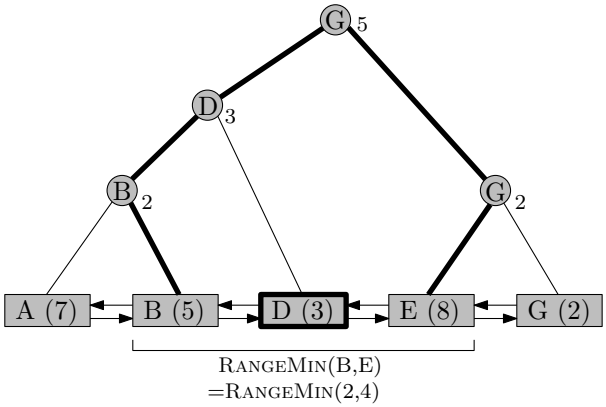


Fig. 2: Data structure to support dynamic RMQs in internal memory. Internal nodes store pointers (depicted within the nodes) to an element below them with minimum priority. The path tracked by the query RangeMin(B, E) is shown in bold. To the right of each internal node v is the number of leaves in the subtree rooted at v, needed for the dynamic array version (e.g., query RangeMin$(2, 4)$).

Geometric Version. We maintain a balanced search tree over the x coordinates of the elements in S. In addition to storing minimum y-values, each internal node stores the largest x-coordinate of an element in its left subtree. Then we locate the elements as in the dynamic array version (by finding predecessors of the x-coordinates), and answer queries as in the linked list version.

4 External Memory Solutions to the Geometric Version

In this section we present a solution to the geometric version of the RMQ problem. We start with a data structure for the static case, as it will form the basis of our dynamic solution.

4.1 Batched Static RMQ

In this section we present a solution to the static RMQ problem that uses only linear space but still incurs only O(sort(N)) I/O complexity.

We sort the set S of N points by the x-coordinates and build a fully balanced k-ary search tree T with $\Theta(N/M)$ leaves on top of them. Each leaf of T is associated with a contiguous x-range of $\Theta(M)$ elements. We set $k = \Theta(m)$, thus, the height of the tree is $\Theta(\log_k(N/M)) = \Theta(\log_m(N/M))$. Each internal node of the tree is associated with the range that is the union of the ranges of its children. Each node v of the tree stores $\mathcal{M}_v$ – a set of k minima of the ranges associated with the k children of v. (In contrast, the solution of Chiang et al. [7] stores prefix minima for every element of the child subtrees.) We assume that $\mathcal{M}_v$ also stores the relevant output information, e.g. the index of the corresponding

element within the array. There are a total of $O(\frac{N}{kM})$ internal nodes each storing $\Theta(k)$ elements. Thus, the size of the tree is $O(N)$.

We populate the sets $\mathcal{M}_v$ bottom up. Each entry $\mathcal{M}_v[i]$ at the parent v of the leaf nodes is set to the minimum of the elements stored at each leaf w_i. The entries at each remaining internal node v are constructed by scanning the entries $\mathcal{M}_w$ of the children nodes w and computing their minima. Thus, the tree T can be built in $O(\log_m(N/M))$ rounds by scanning each level of $O(N)$ elements for a total of $O(\text{sort}(N))$ I/O complexity.

We process the queries in rounds, in each round processing all queries on a single level (starting with the root level) and propagating them down to the next level. For each query at node v, if the range of the query fully falls within the range of some child w of v, we associate the query with w for further processing and continue with the next query. If the endpoints of the query fall within ranges of two different children w_l and w_r, we can compute the value of the range minima query for the portion of the query that spans the children $w_{l+1}, \ldots, w_{r-1}$ by computing the minimum among the subset of values $\mathcal{M}_v$ and any partial answer stored with the query up to this point. We save this value with the query as a tentative answer and associate the query with the two children w_l and w_r containing the endpoints of the query for further processing. At the leaf nodes, we load the leaf of size $\Theta(M)$ into internal memory and answer the query in internal memory. Once all nodes of the tree are processed, for each query we might have up to two potential answers, each stored at the leaves containing the two endpoints of the query. The final answer is the minimum value of the two copies and can be found I/O-efficiently as follows. We sort the answers by the query identifiers (e.g., lexicographically by the left and right indices of the ranges). This process places the two copies of each query in contiguous memory and we can compute the minimum among all pairs of potential answers by scanning the sorted list.

Let Q be the total number of queries. Each query is associated with at most two nodes of the tree. And since we process all queries through each level, we use only $O(Q)$ space for the queries. Together with $O(N)$ space for the data structure, we use a total of $O(N + Q)$ space. The I/O complexity is $O((n + q) \log_m n)$: there are $O(\log_m(N/M))$ internal levels of the tree and at each level of the tree we scan the set of queries (and the internal structures of the tree, which are at most $O(N/M)$). At the leaves we load each leaf into internal memory once, thus, spending $O(n)$ I/Os and scan the set of queries. Finally, the sorting step takes $O(\text{sort}(Q))$ I/Os. Thus, the total I/O complexity is bounded by $O(\text{sort}(Q + N))$.

Parallel Extensions in the PEM Model. The algorithm of Chiang et al. [7] has been used by Arge et al. [4] to develop a parallel solution to the static batched RMQ problem in the parallel external memory (PEM) model [3]. Our new linear space EM solution immediately leads to a linear space PEM solution.

4.2 Batched Dynamic RMQ

Arge et al. [5] present a framework for solving dynamic problems in the external memory model for the so-called *external decomposable* batched problems. Let us review a few definitions from [5]. (Since the RMQ problem has constant output size per query, we omit the parts relevant for the output-sensitive problems).

Definition 1. *[5] Let $\mathcal{P}$ be a searching problem and let $\mathcal{P}(x, V)$ denote the answer to $\mathcal{P}$ with respect to a set of objects V and a query object x. $\mathcal{P}$ is called* external decomposable*, if for any partition $A \cup B$ of the set V and for every query x, $\mathcal{P}(x, V)$ can be computed in $\mathrm{O}(1)$ additional I/Os given $\mathcal{P}(x, A)$ and $\mathcal{P}(x, B)$ in appropriate form.*

Definition 2. *[5] Let $\mathcal{P}$ be an external decomposable batched searching problem. Consider the problem $\mathcal{P}_C$ where a color chosen from a set C is associated with each query x, and where a set of colors C_v is associated with each object $v \in V$. Only objects where $color(x) \in C_v$ are considered when answering x. Problem $\mathcal{P}$ is called $(I(N, K), S(N, K))$ $m^{1/c}$-colorable if the following two conditions hold:*

1. *For all colorings where $|C| = \Theta(\sqrt{m^{1/c}})$ and where the number of different color sets C_v is $\mathrm{O}(m^{1/c})$, for some constant $c \geq 1$, P_C can be solved in $\mathrm{O}(I(N, K)))$ I/O operations and $\mathrm{O}(S(N, K))$ space after an initial sorting step, and*
2. *If (V_1, Q_1) and (V_2, Q_2) are two valid instances of $\mathcal{P}$ then $(V_1 \cup V_2, Q_1 \cup Q_2)$ is also a valid instance.*

Lemma 1. *The static range minima problem is $((n + q) \log_m(n + q), N + Q)$ colorable.*

Proof. Obviously, the RMQ problem is external decomposable by Definition 1. The solution to static RMQ is similar to the solution presented in Section 4.1. We build a full balanced $k = \Theta(\sqrt{m})$-ary tree T on the elements ordered by the x-coordinate and each leaf node containing $\Theta(M)$ elements. Thus, the height of the tree is still $\mathrm{O}(\log_k(N/M)) = \mathrm{O}(\log_m(N/M))$. At each internal node v, instead of storing k minima (one for each subtree rooted at the k children of v), we store $|C| = \Theta(\sqrt{m})$ sets of such k minima: one set for each color.

We populate these sets bottom up. To construct the sets of minima at the parent node v of the leaves, we load $\Theta(M)$ elements of each leaf into internal memory and construct the set of the $|C|$ minima for that leaf, one leaf at a time. We construct the sets of minima for the remaining internal nodes by computing the minima for each color from the ones stored at the children nodes bottom up.

We process the queries as before, top-down level by level, except for each query we use the minima among the ones with the same color as the query.

There are a total of $\mathrm{O}(N/M)$ leaves in the tree, and, consequently, $\mathrm{O}(\frac{N}{kM})$ internal nodes, each storing $k|C|$ minima. Thus, the total space used by the minima is $\mathrm{O}(N|C|/M)$. Each node can be at most $\mathrm{O}(m)$ in size to perform the construction of the tree and processing of the queries I/O-efficiently, i.e., $k|C| = \mathrm{O}(m)$. Thus, if we set $k = |C| = \sqrt{m}$, the tree uses linear space and each node stores $\mathrm{O}(m)$ items. □

Theorem 1. *The batched dynamic range minima problem can be solved in* $\mathrm{O}((n+q)\log_m^2(n+q))$ *I/Os and* $\mathrm{O}(N+Q)$ *space.*

Proof. Arge et al. [5] proved that the batched dynamic version of a static problem $\mathcal{P}$ that is $(I(N,Q), S(N,Q))$ colorable, can be solved in $\mathrm{O}(I(N,Q)\cdot\log_m(n+q))$ I/Os using $\mathrm{O}(S(N,Q))$ space. Combined with Lemma 1, the proof follows. □

5 Reductions between Dynamic Array, Linked List, and Geometric Versions in the EM Model

In this section we reduce the linked list version of the problem to the geometric version, and the dynamic array version to the linked list version. All the reductions take sorting complexity in the EM model, which is less than the complexity of our solution to the geometric version (Section 4). This implies that Theorem 1 also applies to the linked list and dynamic array versions of the problem.

In addition, we show that the geometric version can in turn easily be reduced to the dynamic array version. This would show the equivalence of the three versions if there were a sorting lower bound to any of the versions. However, the sorting lower bound mentioned in the introduction only applies to the online dynamic case, but *not* to the batched case.

5.1 Linked List to Geometric Reduction

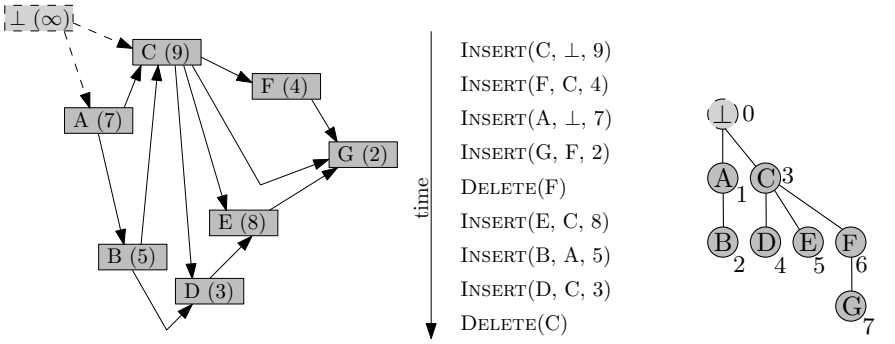
We show how to transform a sequence of $N+Q$ $\textsc{Insert}(v,u,p_v)$, $\textsc{Delete}(v)$, and $\textsc{RangeMin}(u,v)$ linked list operations into a sequence of $N+Q$ $\textsc{Insert}(x,y)$, $\textsc{Delete}(x,y)$, $\textsc{RangeMin}(x_1,x_2)$ geometric operations, by mapping each linked list node v to a point (x_v,y_v) on a plane.

For each linked list node v, we set $y_v = p_v$ by scanning the $\textsc{Insert}(v,u,p_v)$ operations. Finding the x_v's is harder, as described next.

The Insert and Delete operations in the linked list define the following partially persistent linked list (see Fig. 3a for an illustration). The structure is a DAG $\mathcal{G}$, initially consisting of a dummy node $\perp$, representing an artificial list head. $\textsc{Insert}(v,u)$ adds the edges $u \to v$ and $v \to w$, where w is the most recent successor of v in the structure. Likewise, $\textsc{Delete}(v)$ creates the edge $u \to w$, where u and w are the predecessor and successor of v, respectively, at the time of the operation.

To compute the values x_v, we embed $\mathcal{G}$ in one-dimensional space (a line) according to a topological ordering of $\mathcal{G}$. That is, we set x_v equal to the position of the node v in a topological ordering of $\mathcal{G}$. Note that in such an embedding any range $[x_u, x_v]$ includes the nodes of all versions of the linked lists $u \rightsquigarrow v$, and that every node of a linked list $u \rightsquigarrow v$ is contained within the range $[x_u, x_v]$. Therefore, x_v is the correct assignment of the x coordinate to node v.

As $\mathcal{G}$ is planar, we could compute its topological order in $\mathrm{O}(\mathrm{sort}(N))$ I/O complexity [6]. However, we do not know how to construct $\mathcal{G}$ I/O-efficiently. Instead, we consider the following subgraph T of $\mathcal{G}$, which will be enough for our purposes.



(a) Operations and corresponding DAG $\mathcal{G}$. (b) Ordered Tree.

Fig. 3: Reduction from linked list to geometry version.

For each $\textsc{Insert}(v, u, p_v)$ operation we create an edge (u, v) annotated with the timestamp of the operation. Graph T is defined by the adjacency list representation with an ordered set of neighbors for each node u by sorting the generated edges (u, v) lexicographically, primarily by the first node u and secondarily by the edges' timestamps in decreasing order. The following lemma shows that we can use T to compute a topological order of $\mathcal{G}$.

Lemma 2. *Graph T defines a tree rooted at $\perp$. Moreover, T contains all nodes of $\mathcal{G}$, and the preorder traversal of T defines a valid topological order on $\mathcal{G}$.*

Proof. Imagine the DAG $\mathcal{G}$ embedded on a plane, such that the y-coordinate of a node's embedding equals to the timestamp of the insertion operation. Since an $\textsc{Insert}(v, u)$ operation creates a node v below all other nodes, $\textsc{Insert}(v, u)$ creates a downward pointing edge (u, v) and an upward pointing edge (v, w). We draw each edge (u, w) inserted due to $\textsc{Delete}(v)$ operation as a zig-zag: first pointing downward then upward, with the kink at the y-coordinate equal to the timestamp of the Delete operation. We draw $\mathcal{G}$ such that node v corresponding to $\textsc{Insert}(v, u)$ and the kinks in edges corresponding to $\textsc{Delete}(v)$ at time t are drawn to the left of all siblings of v up to time t. Then all edges (u, v_i) for node u (in left to right order) point to nodes v_i with decreasing timestamps. The version of the list at time t is then defined by the traversal of $\mathcal{G}$ starting at node $\perp$ and at each node following the leftmost edge that does not cross the horizontal line $y = t$.

Graph T consists of all nodes of $\mathcal{G}$ and only the downward pointing edges due to insertions, i.e., it is $\mathcal{G}$ with the zig-zag and "up-edges" removed. Thus, it is a tree. The preorder traversal of T implies a valid topological order of $\mathcal{G}$ for the following two reasons: (1) The zig-zag edges introduced by deletions do not affect the topological ordering of $\mathcal{G}$. This is simply because adding an edge $u \to w$ upon a deletion already implies the existence of the path $u \to v \to w$, and because there cannot occur a subsequent $\textsc{Insert}(\cdot, v)$ operation after v was deleted. (2) The "up-edges" created by $\textsc{Insert}(v, u, p_v)$ always point to the elements inserted

earlier, i.e. to the nodes in the subtrees rooted at siblings of v that appeared before v. Since v appears to the left of those siblings, the preorder traversal of the tree places v before any node it might point to. □

The preorder traversal of T can be accomplished I/O-efficiently by building an Euler tour on T and performing list ranking with the appropriate weights in $O(\text{sort}(N))$ I/Os [7].

Thus, we successfully mapped the linked list nodes to points on the plane. In other words, we computed the mapping $v \to (x, y)$ for every node in the linked list. We use these mappings to replace $\textsc{Insert}(v, u, p_v)$, $\textsc{Delete}(v)$ and $\textsc{RangeMin}(u, v)$ with the corresponding $\textsc{Insert}(x_v, y_v)$, $\textsc{Delete}(x_v, y_v)$, and $\textsc{RangeMin}(x_1, x_2)$. The replacement can be performed I/O-efficiently, by sorting the operations and the mapping by the node names and simultaneously scanning the operations and the mapping, generating the geometric operations.

Given the answers to the geometric queries (namely x-coordinates of the answers), we can map them back to the linked list nodes similarly by sorting the answers and the mapping by the x-coordinates.

5.2 Dynamic Array to Linked List Reduction

In this section we show how to map each index and the corresponding timestamp to a node identifier in the linked list.

We will use the *buffer tree* technique of Arge [2] to perform this conversion. The buffer tree is an (a, b)-tree with branching parameters $a, b \in \Theta(m)$ and each leaf of size $\Theta(B)$. I.e., buffer tree is a balanced m-ary search tree of depth $O(\log_m(n+q))$. In addition, each internal node is augmented with a *buffer* of size $\Theta(M)$. All updates and queries are simply inserted into the buffer of the root, annotated with the timestamp of the operation. When a node's buffer is full, the buffer is emptied by loading its elements into the internal memory, processing the elements and pushing them to the buffers of the children nodes. If this causes some child node's buffer to become full, it is processed and emptied recursively. Rebalancing of the tree is performed in a bottom-up way, as in an ordinary (a, b)-tree.

In our case, we use the buffer tree to represent the elements of the underlying dynamic array as a linked list. Each leaf stores $\Theta(B)$ consecutive alive elements of the linked list, i.e., elements that haven't been deleted yet. Each index (in the dynamic array) is given a *name* that serves as an identifier in the linked list. Those names are stored at the leaves of the buffer tree until the element is deleted: when an $\textsc{Insert}$ operation reaches a leaf, we take the next free name from a pool of names and associate it with the element.

The goal of the conversion is that for each of the operations $\textsc{Insert}(i, x)$, $\textsc{Delete}(i)$, and $\textsc{RangeMin}(i, j)$ we identify the name of the element i (and also j in the case of RMQs) at the current time. To this end, every internal node v with children $w_1, \ldots, w_m$ stores a list $x_1, \ldots, x_m$ of m numbers such that x_i is the number of *alive* elements in the subtree rooted at w_i, i.e., the difference between the number of $\textsc{Insert}$ and $\textsc{Delete}$ operations that passed w_i. Note that

x_i is *not* the number of elements currently stored in the leaves of the subtree rooted at w_i, because there could still be some inserts and/or deletes waiting in the buffers between w_i and the leaves of the subtree rooted at w_i.[3]

Hence, during the buffer emptying process, we can route the operations to the correct children when emptying a buffer: at the root v with children $w_1, \ldots, w_m$ and routing elements $x_1, \ldots, x_m$, we process the operations in the buffer in the order of increasing timestamps, and route an operation referring to index i to the child w_j such that $x_{j-1} < i < x_j$. Before passing the operation to the buffer of w_j, we update it to refer to the array index $i - \sum_{k<j} x_k$; by maintaining prefix sums we can keep track of these subtree ranks easily. Note that we have to update the routing elements x_i immediately when processing an update operation (i.e., increasing/decreasing x_i by one when handling an insertion/deletion, respectively), such that future operations get routed to the correct element.

This way, all INSERT and DELETE operation reach the leaf where the element they are referring to is stored. There, they are scanned by increasing timestamp, thereby identifying the name of the element they are referring to in the linked list. Corresponding operations in the linked list version are subsequently generated (together with their original timestamps), and the insertion or deletion is actually applied on the linked list stored at the leaf.

The RANGEMIN queries are handled similarly, with only one difference: for a query RANGEMIN(i, j) at time t, we create two tuples (i, t) and (j, t) and insert these tuples at the root. As before, those tuples will be routed to a leaf (using the first components of the tuples for the search), where they will be associated with a name. A final sort by the second component of the tuples (time) will bring the query endpoints back to each other; they are then translated into corresponding queries in the linked list.

A standard analysis on buffer trees concludes that in $\mathrm{O}(\mathrm{sort}(N))$ I/Os we create linked list operations whose answers correspond to the dynamic array queries. It remains to map the answers (elements in a linked list) back to the dynamic array version (indices in the array). We accomplish this by processing all insertions, deletions and the *answers* to the queries using a buffer tree again. However, this time we use the buffer tree in its "native" setting: from the solution to the geometric version we know the x-coordinates of all elements; therefore, we can use x-coordinates as the routing elements. As before, each node stores the number of alive elements in each of its subtrees, and these numbers are updated immediately upon handling an operation waiting in the node's buffer. If a query answer (a linked list element) is routed to child w_j of v, we need to count the sum of alive elements in v's children to the left of w_j and propagate this information down with the query. Finally, when a query answer reaches a leaf of the buffer tree, we count its relative rank within the $\Theta(B)$ elements currently stored at the leaf; this results in the final rank of the query answer.

[3] However, the tree *balancing* is still performed based on the actual number of elements stored in the leaves.

5.3 Geometric to Dynamic Array Reduction

To convert a batch of operation from the geometric setting to the dynamic array setting, we use the ranking mechanism explained at the end of Section 5.2 (using buffer trees) to determine each point's rank. We also store the original x-coordinates (of the geometric setting) along with the new batch of operations.

6 Conclusions and Open Questions

We addressed the problem of static and dynamic batched range minimum queries in external memory and presented several algorithms for solving the problem. Since the sorting lower bound only applies to the online dynamic case, and not to batched queries, the following open questions for future research immediately come to mind: (i) what is the optimal I/O complexity of static batched range minima? We conjecture that it is the sorting bound. (ii) Likewise, what is the optimal I/O complexity of the dynamic batched version of the problem?

Other open questions concern dynamic solutions in more advanced models, like the cache oblivious model or the parallel external memory (PEM) model. The difficulty in addressing the dynamic setting in parallel lies in the seemingly sequential relationship between past updates and future queries.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Arge, L.: The buffer tree: A technique for designing batched external data structures. Algorithmica 37(1), 1–24 (2003)
3. Arge, L., Goodrich, M.T., Nelson, M.J., Sitchinava, N.: Fundamental parallel algorithms for private-cache chip multiprocessors. In: SPAA. pp. 197–206 (2008)
4. Arge, L., Goodrich, M.T., Sitchinava, N.: Parallel external memory graph algorithms. In: IPDPS. pp. 1–11 (2010)
5. Arge, L., Procopiuc, O., Ramaswamy, S., Suel, T., Vitter, J.S.: Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In: SODA. pp. 685–694 (1998)
6. Arge, L., Toma, L., Zeh, N.: I/O-efficient topological sorting of planar DAGs. In: SPAA. pp. 85–93. ACM Press (2003)
7. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: SODA. pp. 139–149 (1995)
8. Fischer, J., Heun, V.: Space efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40(2), 465–492 (2011)
9. Fischer, J., Mäkinen, V., Välimäki, N.: Space efficient string mining under frequency constraints. In: Proc. ICDM. pp. 193–202. IEEE Computer Society (2008)
10. Franceschini, G., Grossi, R.: A general technique for managing strings in comparison-driven data structures. In: ICALP. pp. 606–617 (2004)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 1–19 (2006)
12. Raman, R.: Range extremum queries. In: IWOCA. pp. 280–287 (2012)