Concurrent Range Reporting in Two-Dimensional Space

Peyman Afshani* Cheng Sheng† Yufei Tao† Bryan T. Wilkinson*‡

Abstract

In the *concurrent range reporting* (CRR) problem, the input is L disjoint sets $S_1, ..., S_L$ of points in $\mathbb{R}^d$ with a total of N points. The goal is to preprocess the sets into a structure such that, given a query range r and an arbitrary set $Q \subseteq \{1, ..., L\}$, we can efficiently report all the points in $S_i \cap r$ for each $i \in Q$. The problem was studied as early as 1986 by Chazelle and Guibas [9] and has recently re-emerged when studying higher-dimensional complexity of orthogonal range reporting [2, 3].

We focus on the one- and two-dimensional cases of the problem. We prove that in the pointer-machine model (as well as comparison models such as the real RAM model), answering queries requires $\Omega(|Q| \log(L/|Q|) + \log N + K)$ time in the worst case, where K is the number of output points. In one dimension, we achieve this query time with a linear-space dynamic data structure that requires optimal $O(\log N)$ time to update. We also achieve this query time in the static case for dominance and halfspace queries in the plane. For three-sided ranges, we get close to within an inverse Ackermann ($\alpha(\cdot)$) factor: we answer queries in $O(|Q| \log(L/|Q|)\alpha(L) + \log N + K)$ time, improving the best previously known query times of $O(|Q| \log(N/|Q|) + K)$ and $O(2^L L + \log N + K)$. Finally, we give an optimal data structure for three-sided ranges for the case $L = O(\log N)$.

*MADALGO, Aarhus University, Denmark, {peyman,btw}@madalgo.au.dk. MADALGO is a center of the Danish National Research Foundation.

†Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong, {csheng,taoyf}@cse.cuhk.edu.hk. Supported by grants GRF 4165/11, GRF 4164/12, and GRF 4168/13 from HKRGC.

‡Supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

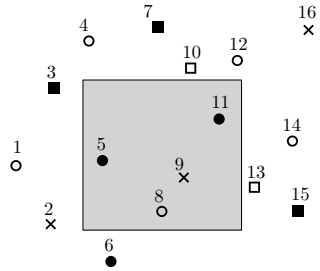


Figure 1: The dataset has $L = 5$ sets of points indicated with different types of markers. Let r be the shaded rectangle and $Q = \{\times, \bullet\}$. Only points 5, 9, and 11 should be reported.

1 Introduction

We consider *concurrent range reporting* (CRR) in two-dimensional space. The dataset consists of L sets $S_1, ..., S_L$ of points in $\mathbb{R}^d$, where L is potentially $\omega(1)$. We want to preprocess the sets into a structure such that, given a query range r and an arbitrary set $Q \subseteq \{1, ..., L\}$, we can efficiently report all the points in $S_i \cap r$ for each $i \in Q$. See Figure 1. This is equivalent to performing traditional range searching on $|Q|$ sets of points *simultaneously*, thus raising the hope of improving query efficiency (compared to searching each set separately) by sharing some of the common computation. We consider axis-parallel rectangular ranges (i.e., orthogonal ranges) as well as halfspace ranges. Special versions of the orthogonal problem are *three-sided CRR* and *two-sided CRR* where r is a rectangle of the form $[x_1, x_2] \times [y, \infty)$ and $[x, \infty) \times [y, \infty)$, respectively.

The CRR problem is useful in several domains. An important example is *geographic information system* (GIS), which organizes a massive volume of spatial entities of different varieties, such as residential buildings, factories, hotels, restaurants, gas stations, schools, etc. One major use of a GIS is to offer a convenient way for a user to explore a map, by rendering the entities in the current navigation window of the user. A user is often given the option of turning on only some *thematic layers*. One such layer, for instance, can be the set of hotels whereas another layer may include all the restaurants, and so on. It is exactly a CRR query to find all and only the entities of the requested layers. An analogous application is a variant of *spatial keyword search*. Imagine that, in Figure 1, the points with an identical marker represent restaurants of the same company (e.g., $\times, \bullet, \circ, ...$ correspond to McDonald's, Pizza Hut, Burger King, ... respectively). A typical query is like "*find all restaurants of McDonald's and Pizza Hut in my neighborhood*". This is exactly a CRR query with $Q = \{\times, \bullet\}$.

Define S as the union of $S_1, ..., S_L$. Set $N = \sum_{i=1}^{L} |S_i|$. In practice, N is often far greater than L. By standard tie-breaking techniques, we consider that no two points in S have the same x- (or y-) coordinate. To facilitate discussion, let us associate every point $p \in S_i$ with a *color* i. Accordingly, Q can be interpreted as a set of colors. Note, however, that CRR should not be confused with *colored range reporting*[1] [4].

1.1 Previous Results

The one-dimensional CRR problem was studied first by Chazelle and Guibas [10] in their seminal 1986 paper on fractional cascading where they obtained a linear-size data structure with $O(|Q| \log \frac{L}{|Q|} + \log N + K)$ query time. Here, K is the number of reported points. They also proved a $\Omega(|Q| \log \frac{L}{|Q|})$ lower bound in a pointer machine model, although a peculiar aspect of the lower bound is that their representation of the query colors is not the same as the one used in their data structure so their lower bound does not in fact apply to their

[1] In colored range reporting, the underlying dataset is the same as in the CRR problem. Given a rectangle r, a query retrieves the distinct *colors* of the points covered by r. In other words, even if a color has multiple points in r, the color should be reported only once. The CRR problem differs in that, in addition to a rectangle r, a query also specifies a color set Q of interest. Accordingly, the output of the query is also substantially different from that in colored range reporting, as can be verified easily.

algorithm (see Section 2 for more details). By applying dynamic fractional cascading [9, 16], this method supports an update in $O(\log N)$ time, but the query cost becomes $O(|Q| \log \frac{L}{|Q|} \log\log N + \log N + K)$.

In two dimensions, a trivial solution is to issue $|Q|$ individual range reporting queries, one on each color in Q. Using a structure of Chazelle [8], this gives a data structure for the two-dimensional CRR problem that uses $O(N \log N / \log\log N)$ space[2] and answers a query in $O(|Q| \log \frac{N}{|Q|} + K)$ time. If all queries are three-sided, the space can be lowered to linear by resorting to the priority search tree [15], while the query cost remains the same. The only non-trivial solutions for CRR in dimensions two and higher were given by Afshani et al. [2] (see also [3]). They considered a special instance of three-sided CRR, which has an additional constraint that Q can be selected from only a number $M \leq 2^L$ of possible choices. They gave a linear-space structure that answers a (three-sided) query in $O(ML + \log N + K)$ time. While this adequately serves the purposes in [2], in our context where Q can be any subset of $[L]$ (notation $[x]$ represents the set of integers $\{1, ..., x\}$), the query time becomes $O(2^L L + \log N + K)$.

1.2 Our Results

Our main contribution is to show that almost optimal results can also be obtained for two-dimensional CRR problems. Our bounds involve some sublogarithmic functions which we now define. Let $\alpha_p(x)$ for $p \geq 1$ be a family of functions that are defined for $x \geq 1$. In the base case $p = 1$, $\alpha_1(x) = x - 2$. For $p > 1$, we let $\alpha_p(x) = \min\{i \mid i \geq 1 \wedge \alpha_{p-1}^{(i-1)}(x) \leq 2\}$. Here, $\alpha_{p-1}^{(i-1)}$ is the identity function composed with $i - 1$ instances of the α_{p-1} function. In this way, $\alpha_2(x) = \lceil n/2 \rceil$, $\alpha_3(x) = \max\{1, \lceil \log x \rceil\}$, $\alpha_4(x) = \max\{1, \lceil \log^* x \rceil\}$ and so on. Let $\alpha(x)$ be $\min\{p \mid \alpha_p(x) \leq 3\}$ so that it is asymptotically equivalent to the inverse Ackermann function [12].

Theorem 1. *Given N points in $\mathbb{R}^2$ divided into L disjoint sets, we can construct a pointer-machine structure that answers three-sided CRR queries and has one of the following pairs of bounds (K is the size of the query result):*

1. *$O(N\alpha_p(N/L))$ space and $O(|Q| \log(L/|Q|) + \log N + K)$ query time, for any constant $p \geq 3$;*
2. *$O(N)$ space and $O(|Q|(\log(L/|Q|) + \log \alpha_p(N/L)) + \log N + K)$ query time, for any constant $p \geq 3$; or*
3. *$O(N)$ space and $O(|Q| \log(L/|Q|)\alpha(N/L) + \log N + K)$ query time.*

Each of these data structures require $O(S(N) \log N)$ preprocessing time, where $S(N)$ is the data structure's space bound.

Our data structure makes extensive use of a space-saving technique for three-sided range searching appearing as early as the work of [17] that involves dividing the point set into slabs and building data structures for the points within each slab and a data structure on representative points from each slab.

Additionally, we obtain a number of other results that enhance our understanding of the CRR problems: We consider two different methods of specifying the query colors as the previous papers had not done this in a satisfactory way (e.g., consider the peculiar aspect of Chazelle and Guibas's lower bound [9]). We prove a significantly more involved lower bound and provide some other reductions that ultimately show the different formulations of the query colors are in fact equivalent. We also observe that the $\Omega(|Q| \log(L/|Q|) + \log N + K)$ query lower bound also holds in the comparison model, which means our results are almost optimal in other fundamental models such as the real RAM model.

The data structure of Afshani et al. [2] achieves linear space and $O(\log N + K)$ query time for the case $L \leq \log\log N - \log\log\log N$. In Section 6, we give a data structure that achieves these same bounds for a broader special case of $L = O(\log N)$. Our data structure uses persistence to transform a dynamic 1d solution based on an interval tree into a static three-sided solution. If we use either of these data structures for small settings of L and the data structure of Theorem 1 for larger L, then we can rewrite the bounds

[2] All logarithms in this paper have base 2. Furthermore, we follow the convention that every logarithm returns a value at least 1, by defining $\log x = \max\{1, \log_2 x\}$.

of Theorem 1 so that, for $p \geq 4$, $\alpha_p(N/L)$ becomes $\alpha_p(L)$ and $\alpha(N/L)$ becomes $\alpha(L)$. This is because, for $p \geq 4$ and $L > \log\log N - \log\log\log N$, $\alpha_p(N/L) = O(\alpha_p(L))$ and $\alpha(N/L) = O(\alpha(L))$.

By standard (range tree) techniques, Theorem 1 implies that a general (four-sided) CRR query can be settled with the same query cost, after increasing space by a factor of $O(\log N)$. When restricted to two-sided ranges, and also for halfspace ranges, we obtain linear space and $O(|Q| \log \frac{L}{|Q|} + \log N + K)$ query time. We also present an optimal dynamic one-dimensional structure.

Theorem 2. *Given N values in $\mathbb{R}$ divided into L disjoint sets, there is a dynamic structure that requires $O(N)$ space, and answers a one-dimensional CRR query in $O(|Q| \log \frac{L}{|Q|} + \log N + K)$ time, where K is the size of the query result. The structure supports insertions and deletions in $O(\log N)$ amortized time.*

We sometimes need to perform a *concurrent lookup* on a BST T indexing L keys. Given a set Q of sorted keys, each of which is indexed by T, this operation finds the nodes in T corresponding to the keys in Q. Chazelle and Guibas [10] show that the minimum spanning tree of the $|Q|$ output nodes in T has size $O(|Q| \log(L/|Q|))$. It is then straightforward to perform a concurrent lookup in $O(|Q| \log(L/|Q|))$ time by performing an Euler tour of this minimum spanning tree: at each step we progress towards the least query key that we have not yet found. Since the query keys are sorted, we always know the next least key to progress towards.

2 Query Representation and Lower Bounds

In this section, we discuss the representation of query colors as there is a potential for ambiguity when studying CRR problems in a pointer machine. In fact this ambiguity has already revealed itself in the previous work of Chazelle and Guibas [10]: when employing the fractional cascading technique, they assume that the query colors are given as pointers to special nodes of the pointer machine but while proving the optimality of said technique, they assume that the query colors are given instead by indices. This means in their data structure, it is assumed that the query begins by having access to $|Q|$ pointers (thus $|Q|$ entry points into the data structure) but when proving the lower bound, they assume the query algorithm begins by having access to one entry point. The latter is a much more restrictive starting setup for the query algorithm which means their data structure does not in fact fit their lower bound framework!

We explicitly define these two different (but as it turns out equivalent) query representations. In the first representation, the set of query colors is a sorted set of indices. We consider this the *index query representation*. In an alternate representation, each color i is associated with some specific node c_i in the pointer-machine structure called a *color node*. Let $\mathcal{C} = \{c_1, c_2, \ldots, c_L\}$ be the set of all the color nodes. A set of query colors is specified by a set $Q \subseteq \mathcal{C}$ (more precisely, Q is a set of pointers to the color nodes) that is not necessarily sorted in any way. We call this the *pointer query representation*.

We first give $O(L)$-space and $O(|Q| \log(L/|Q|))$-time reductions between the index and pointer query representations. We then proceed to show that there is a lower bound of $\Omega(|Q| \log(L/|Q|))$ for both query representations. Thus, these representations are equivalent in the pointer-machine model, and any data structure for CRR with $O(|Q| \log(L/|Q|) + \log N + K)$ query time is optimal in the pointer-machine model.

2.1 Query Representation Reductions

Lemma 1. *Given a data structure D for CRR with the pointer query representation that requires T_D query time and S_D space, there exists a data structure for CRR with the index query representation that requires $O(S_D + L)$ space and $O(T_D + |Q| \log(L/|Q|))$ query time.*

Proof. We build a BST T on the L color nodes of D based on the sorted order of the colors. Then, given a sorted set of query colors Q, we find their associated color nodes in $O(|Q| \log(L/|Q|))$ time via a concurrent lookup in T. We then query D with the color nodes. □ □

Lemma 2. *Given a data structure D for CRR with the index query representation that requires T_D query time and S_D space, there exists a data structure for CRR with the pointer query representation that requires $O(S_D + L)$ space and $O(T_D + |Q| \log(L/|Q|))$ query time.*

Proof. We construct a color node for each color and store them in a BST T based on the sorted order of the colors. As shown by Chazelle and Guibas [10], if we augment T with parent pointers, then we can construct the minimum spanning tree of a set of query color nodes in $O(|Q| \log(L/|Q|))$ time. In the same amount of time, we perform an in-order traversal of the minimum spanning tree to recover the query colors in sorted order. We then query D with the sorted set of query colors. □
□

2.2 Lower Bounds

For the intex query representation, obtaining a comparison-based lower bound is straightforward: consider a dataset where each S_i has only one point and a query range that covers all of these points. For any specific $|Q|$, there are $\binom{L}{|Q|}$ size-$|Q|$ subsets of $[L]$, each of which defines a unique result. Since any query algorithm must be able to distinguish at least as many results, it follows that $\Omega(\log \binom{L}{|Q|}) = \Omega(|Q| \log(L/|Q|))$ comparisons are needed. This is essentially a lower bound for a problem we call the *concurrent lookup problem*: given a set T of keys, store them in a data structure such that given a set Q of sorted query keys, each of which is indexed by T, find the keys in T that correspond to the keys in Q.

We now consider the pointer-machine model. In this model, the memory is a directed graph with out degree of two (the model generalizes easily to any constant out degree as well) and each node of the graph can store one input element. To output an element, the query algorithm must navigate the pointers to reach a cell that contains the element. While Chazelle and Guibas proved a pointer-machine lower bound [10], their lower bound assumes the colors are given as indices rather than pointers. In this way, the CRR problem inherits the difficulty of the concurrent lookup problem, which makes proving a lower bound easier. Instead, we consider the pointer query representation, which means the query algorithm begins with access to $|Q|$ entry points to the data structure; this is in contrast to the traditional pointer machine data structures in which the query algorithm begins from the root of the data structure. In fact, because of this, we cannot take advantage of the difficulty of the concurrent lookup problem, and we must instead consider the geometry of the problem. Essentially, we space out many instances of the concurrent lookup problem and show that the color nodes cannot adequately help all instances of the problem. While simple counting arguments work to establish a lower bound when starting from one pointer, they seem to be ineffective when starting from $|Q|$ pointers. When the query algorithm begins from one pointer, it can only access $2^{O(k)}$ different subsets of the memory cells by making k pointer jumps so to be able to access $\binom{L}{|Q|}$ sets we must have $2^{O(k)} \geq \binom{L}{|Q|}$ or $k = \Omega(|Q| \log(L/|Q|))$. On the other hand, if we give the query algorithm access to $|Q|$ pointers, there are $\binom{L}{|Q|}$ ways just to pick our entry points and so the counting argument is rendered completely ineffective. To establish a lower bound for the pointer query representation, we have to work harder.

Theorem 3. *Any pointer-machine data structure for CRR with the pointer query representation that uses $(N/L)^{O(1)}$ space, requires $\Omega(|Q| \log(L/|Q|) + \log N + K)$ query time.*

Proof. Our bad input instance is a one-dimensional point set composed of N/L sets of size L. Each set only contains one point of each color and the points in the i-th set are laid consecutively in an *interval* I_i; the intervals are disjoint and they are also used as geometric ranges for queries. Thus, for a set $Q \subseteq \mathcal{C}$ of color nodes, the output size for the query range I_i is exactly $|Q|$. In our proof, the number of colors for all queries will be fixed but to avoid introducing extra variables, we will continue to use $|Q|$ to refer to this fixed value.

If $\log N \geq |Q| \log(L/|Q|)$ then there is nothing left to prove since there is a trivial $\Omega(\log N)$ lower bound in the pointer-machine model. In the rest of this proof, we assume otherwise. Let G be the directed graph that corresponds to the memory layout of the data structure: the vertices of G are memory cells and edges correspond to the pointers between the memory cells. G has L cells that correspond to the color nodes; the query algorithm begins the search from these cells. We say a memory cell (in G) is *shallow*, if it lies at

a depth less than $\log(N/L) - 2$ of a color node. We say an interval I is *shallow*, it there are at least $L/2$ shallow cells that store points in I. Since the number of shallow cells is less than $L2^{\log(N/L)-2} = N/4$, there are less than $N/(2L)$ shallow intervals. We will only use non-shallow intervals for our queries so one can safely ignore the shallow intervals in the rest of this proof.

Given a non-shallow interval I, consider the query algorithm: it starts with access to $|Q|$ pointers and then continues to make pointer jumps in graph G until all the cells containing the output have been accessed. We ignore the part of the output that is stored in shallow cells so for the rest of this proof, by output we mean only the points that are not stored in shallow cells (in other words, we only require the query algorithm to output elements stored in non-shallow cells).

Consider the subgraph H of G composed of memory cells that are explored at the query time and the edges that are used to visit these cells for the first time. Except for the color nodes, each vertex of H has in-degree of exactly one (each color node has in-degree zero). So, H is a forest with $|Q|$ arborescences[3] such that each color node in Q is the root of one arborescence. H contains at most $|Q|$ cells that store the output points of the query. We call these the *output cells*. We denote the number of output cells in an arborescence T with $k(T)$. Let $\alpha|Q|$ be the size of H which is also a lower bound on the query time. We call a subgraph of G a β-heavy hub of size r if it is an arborescence of size r with β output cells. Intuitively, our proof idea is to combine two things: one, that for every query, the subgraph H contains at least one β-heavy hub of size $O(\alpha\beta)$ and two, that the number of β-heavy hubs of size $O(\alpha\beta)$ that the graph G can use at the query time depends on and grows as a function of α.

We set $\beta = C\log(N/L)/\log(L/|Q|)$, where C is a constant to be determined later. Consider an arborescence T in H with $k(T) > 0$. Observe that T must have at least $\log(N/L) - 2$ vertices (since its output cells are not shallow). If a fraction (say a quarter) of all the output elements lie in arborescences T such that $k(T) \leq 2\beta$, then we are good: each such arborescence uses at least an average of $\log(L/|Q|)/(2C)$ pointer jumps to output one element and thus the query time is already $\Omega(|Q|\log(L/|Q|))$. Thus, assume otherwise which means we can afford to remove any arborescences T with $k(T) \leq 2\beta$ and still have three quarters of the output elements left. After this, since the total size of all the arborescences were $\alpha|Q|$, it easily follows that there exists at least one arborescence T such that $|T| \leq 2\alpha k(T)$ with $k(T) \geq 2\beta$. In this case, by using the same technique as in [1], we can find at least one β-heavy hub of size $O(\alpha\beta)$.

We now look at the overall structure of graph G. While the definition of a β-heavy hub only makes sense with respect to a given query, the structure of the hub must still be embedded in graph G. This means that G has only a limited number of different β-heavy hubs of size $O(\alpha\beta)$. To bound this number, we can pick a cell v in graph G, consider $O(\alpha\beta)$ pointer jumps that visit $O(\alpha\beta)$ other memory cells in G and then pick β cells out of all the visited cells to be output cells. After considering all the different possible pointer jumps, and all the different ways to pick β cells out of $O(\alpha\beta)$ cells, it is a simple exercise (for more details see [1]) to show that the number of different possible β-heavy hubs of size $O(\alpha\beta)$ in G is $S(N)2^{O(\alpha\beta)}$ where $S(N)$ is the total space used by the data structure ($S(N)$ is in fact the number vertices in graph G). Now, remember that we have at least $N/(2L)$ non-shallow intervals so there exists a non-shallow interval I such that there are at most $S(N)2^{O(\alpha\beta)}L/N$ possible β-heavy hubs of size $O(\alpha\beta)$ that involve cells that store points of I.

We fix I as the query range and it remains to define the color set. Since I has only one point of each color, output points and query colors are interchangeable (i.e., we can determine Q by a query's set of output points). To pick the set of output points, we consider the points of I that are not stored in shallow cells; by the non-shallowness of I, there are at least $L/2$ such points. Q is picked by randomly sampling each such point with probability $2|Q|/L$ (remember that $|Q|$ is a fixed parameter). The number of points that we consider is at least $L/2$, so we expect to sample $|Q|$ colors. Now, consider a possible β-heavy hub h. If the point stored at one of the output cells of h is not sampled, then h is not useful since it cannot become a β-heavy hub for our query. Thus, the probability that h is useful for the our query is at most $(2|Q|/L)^\beta$. This implies that the expected number of β-heavy hubs that can be useful for the query is at most

$$\left(\frac{2|Q|}{L}\right)^\beta S(N)2^{O(\alpha\beta)}\frac{L}{N}.$$

[3]An arborescence is a directed tree in which there is a directed path from the root to every other node, or informally, a directed tree in which the edges are directed away from the root.

Since $\beta = C\log(N/L)/\log(L/|Q|)$, we have $(|Q|/L)^\beta = 2^{-C\log(N/L)} = (N/L)^{-C}$. To be able to answer the query, the query should have at least one such β-heavy hub, implying

$$\left(\frac{L}{N}\right)^{C-1} S(N) \left(\frac{N}{L}\right)^{O\left(\frac{C\alpha}{\log(L/|Q|)}\right)} \geq 1.$$

Since $S(N) = (N/L)^{O(1)}$, by setting C a large enough constant, it follows that we must have $\alpha = \Omega(\log(L/|Q|))$. □ □

Corollary 1. *For $L \leq \sqrt{N}$, any pointer-machine data structure for CRR with the pointer query representation that uses polynomial space, requires $\Omega(|Q|\log(L/|Q|) + \log N + K)$ query time.*

3 Two-Sided and Halfspace CRR

Before looking at 2d CRR, consider the concurrent version of predecessor search. In this problem we must preprocess L sets $S_1, \ldots, S_L$ of elements so that we can efficiently find the predecessor of a query element q in S_i for each i in a set of query colors $Q \subseteq [L]$. The solution to 1d CRR of Chazelle and Guibas [10] also solves concurrent predecessor search in linear space and optimal $O(|Q|\log(L/|Q|) + \log N)$ time.

Assume we have a data structure for traditional range reporting that, in order to answer a query, performs a single 1d predecessor search and then uses $O(K)$ additional time to report all necessary points. Supposed further that the query element used in the 1d predecessor search depends only on the original query range (i.e., it does not depend on the points stored in the data structure). Observe that we can then build an efficient data structure for CRR by building this traditional data structure for each color, but performing the predecessor searches for each query color concurrently. This concurrent predecessor search requires $O(|Q|\log(L/|Q|) + \log N)$ time, and all subsequent work is bounded by $O(K)$.

Lemma 3. *There exists a linear-space data structure for two-sided range reporting that requires, for any query, a single predecessor search that depends on only the query and $O(K)$ additional time. The data structure requires $O(N \log N)$ preprocessing time.*

Proof. Given a point set S, let $\ell_1, \ldots, \ell_m$ be all m non-empty layers of maxima of S, such that ℓ_i contains the maxima of $S \setminus \bigcup_{j=1}^{i-1} \ell_j$. We store the points of each layer in a linked list in order of increasing x-coordinate (or equivalently, decreasing y-coordinate). For each layer ℓ_i, we build a catalog C_i containing the points of ℓ_i keyed by their x-coordinates. We connect the catalogs into a path ordered by layer index to create a catalog graph that supports fractional cascading. Proprocessing time is dominated by the construction of the layers of maxima in $O(N \log N)$ time by a simple sweep-line algorithm.

Given an query with range $r = [x_0, \infty) \times [y_0, \infty)$, we report points layer by layer, starting at ℓ_1, until we reach a layer ℓ_i such that $\ell_i \cap r = \emptyset$. Since the points of ℓ_i dominate the points of all further layers, for any further layer ℓ_j, it must also be that $\ell_j \cap r = \emptyset$. When we consider some layer ℓ_i, we must report $\ell_i \cap r$. In order to do so, we search for the point p associated with the successor of x_0 in C_i. If there is no such point, then $\ell_i \cap r = \emptyset$ and we are done. Otherwise, p is the highest point of ℓ_i that is to the right of x_0. We report points of the linked list for ℓ_i, starting from p, until we reach a point with y-coordinate less than y_0.

We need to search for the successor of x_0 in a path of at most $O(K)$ catalogs in our catalog graph. By fractional cascading, this requires a single predecessor search for x_0 in the augmented catalog for C_1, followed by $O(K)$ additional work. □ □

Lemma 3 implies an optimal two-sided CRR pointer-machine structure with linear space and $O(|Q|\log(L/|Q|) + \log N + K)$ query time that can be built in $O(N \log N)$ time. There is also a linear-space data structure for 2d halfspace range reporting that performs a single predecessor search followed by $O(K)$ additional work. It is similar to the data structure of Lemma 3, but uses convex layers instead of layers of maxima. Convex layers can also be constructed in $O(N \log N)$ time [7]. Finding a point on a convex layer in a given halfspace reduces to predecessor search over the slopes of the edges of the convex layer, using the slope of the halfspace boundary line as the query element. Therefore, there is a linear-space data structure for 2d halfspace CRR with $O(|Q|\log(L/|Q|) + \log N + K)$ query time that can be built in $O(N \log N)$ time.

4 Three-Sided CRR

This section concerns the proof of Theorem 1. Recall that the dataset S has N points in $\mathbb{R}^2$, each of which is associated with a color $i \in [L]$. A three-sided CRR query specifies a rectangle $r = [x_1, x_2] \times [y, \infty)$ and a color set $Q \subseteq [L]$, and retrieves $S_i \cap r$ for each $i \in Q$. As mentioned in Section 1, there is a simple linear-space data structure for three-sided CRR based on priority search trees that requires suboptimal $O(|Q| \log(N/|Q|) + K)$ query time. Priority search trees can be built in $O(N \log N)$ time, which is thus the data structure's preprocessing time. Note, that in the special case where $N = O(L)$, this naive data structure is actually optimal. We will use this data structure as the base case for a few recursive structures. We will also use the two-sided solution of Section 3 as a black box. For each new data structure in this section, the analysis of the data structure's preprocessing time is identical to that of its space consumption, except that the preprocessing bound has an extra multiplicative $O(\log N)$ factor. This extra factor is simply propagated from the naive three-sided data structures and optimal two-sided data structures that we use as black boxes.

Lemma 4. *There exists a data structure for three-sided CRR that requires $O(N \log(N/L))$ space and $O(|Q| \log(L/|Q|) + \log N + K)$ query time.*

Proof. As mentioned in Section 3, using a range tree and our optimal two-sided solution, we can create an $O(N \log N)$-space data structure for three-sided queries with optimal query time. By modifying the tree so that it has fat leaves of size $O(L)$, and building the naive data structure for three-sided CRR in these leaves, we reduce space to $O(N \log(N/L))$ and retain optimal query time. □ □

Note that the data structure of Lemma 4 requires $O(N\alpha_3(N/L))$ space. We now give a technique that reduces space from $O(N\alpha_p(N/L))$ to $O(N\alpha_{p+1}(N/L))$.

Theorem 4. *There exists a data structure for three-sided CRR that requires $O(Np\alpha_p(N/L))$ space and $O(|Q|p \log(L/|Q|) + \log N + K)$ query time, for any $p \geq 3$.*

The data structure of Theorem 4 for parameter $p = q$ recursively uses the data structure for $p = q - 1$. Let $S_p(N)$ and $T_p(N)$ be the space and query time, respectively, required by our data structure for parameter p. In the base case $p = 3$, we simply use the data structure of Lemma 4, so we have $S_3(N) = O(N\alpha_3(N/L))$ and $T_3(N) = O(|Q| \log(L/|Q|) + \log N + K)$, as required. We now wish to prove the theorem for $p = q > 3$.

We construct a range tree $\mathcal{T}$ on the x-coordinates of points so that each node u is associated with an x-range $I(u)$ and the x-ranges of the children of u partition $I(u)$ such that an equal number of points of S lie in each child x-range. The children of u are ordered based on the natural ordering of their x-ranges. Let S_u be the points of S which lie in $I(u)$. Our range tree $\mathcal{T}$ has a different fanout parameter at each level. If the root of $\mathcal{T}$ is at level 1, then a node u at level i is associated with an x-range $I(u)$ that contains $L\alpha_{q-1}^{(i-1)}(N/L)$ points. Our range tree $\mathcal{T}$ has fat leaves of size at most $2L$ at level $\alpha_q(N/L)$.

In each fat leaf u, we store the naive three-sided CRR data structure on S_u, which is optimal for $|S_u| = O(L)$ points. In each non-root node u we build a two-sided CRR data structure D_u^+ on S_u for queries of the form $[x, \infty) \times [y, \infty)$ and (by mirroring S_u) another data structure D_u^- for queries of the form $(-\infty, x] \times [y, \infty)$. Now, consider an internal node u. We build a BST $\mathcal{B}_u$ on the set of left boundaries of the x-ranges of the children of u. Finally, we build a data structure $D_u^{||}$ on S_u for three-sided ranges of the form $[x_1, x_2) \times [y, \infty)$ where $x_1, x_2 \in \mathcal{B}_u$. We call these *aligned* queries.

Lemma 5. *Given a data structure D that uses $S_D(N)$ space and answers three-sided CRR queries in $T_D(N)$ time, there is a data structure that supports aligned queries in N/x columns of x points and uses at most $S_D(NL/x) + O(N)$ space and answers queries in $T_D(NL/x) + O(K)$ time.*

Proof. In each column i, we store all points of the same color j in a linked list $H_i(j)$ so that the points are in descending order of their y-coordinates. Since we do not store lists for colors which are not represented in a column, this requires $O(N)$ space. We call the first point (i.e., the one with the largest y-coordinate) of

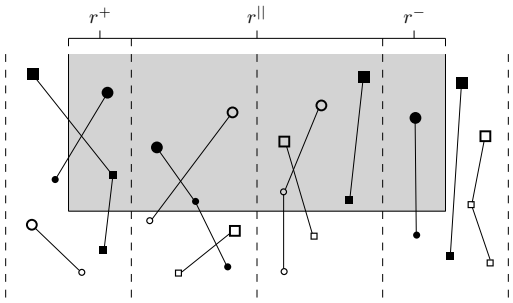


Figure 2: The points of S_u are partitioned into columns. Head points (depicted large) are the highest points of each linked list. The query range is decomposed into two-sided queries in the outer columns and a single aligned query.

each list $H_i(j)$ a head point. See Figure 2. We store all head points in data structure D, which requires at most $S_D(NL/x)$ space.

Given an aligned query with range $r = [x_1, x_2] \times [y, \infty)$ and color set Q, we forward the query to the data structure containing only head points, so that in $T_D(NL/x)$ time we have reported all appropriate head points. Consider any non-head point that should be reported. Then, every prior point in its linked list $H_i(j)$ must also be reported. The points of $H_i(j)$ that lie in r thus form a prefix of $H_i(j)$. We know that the head point for $H_i(j)$ was reported. So, for every head point that was reported, we scan its linked list and report its points until we reach a point that is outside of r. We charge this additional work to the output size. □ □

The data structure $D_u^{||}$ is that of Lemma 5, where $x = L\alpha_{q-1}^{(i)}(N/L)$ and D is the data structure of Theorem 4 with parameter $p = q - 1$.

For the space analysis, we strengthen our claimed space bound to $S_p(N) \leq cNp\alpha_p(N/L)$ for a sufficiently large constant c. Our induction hypothesis gives that $S_{q-1}(N) \leq cN(q-1)\alpha_{q-1}(N/L)$. Consider a single level i of $\mathcal{T}$. Each node u in level i requires at most $S_{q-1}(|S_u|L/x)+c|S_u|$ space. Here the $c|S_u|$ term includes the linked lists of Lemma 5, as well as the data structures D_u^+, D_u^-, and $\mathcal{B}_u$. By the induction hypothesis and simple arithmetic, the total space for u is less than or equal to $c|S_u|q$. The total space required at level i is thus at most cNq, since every point in S lies in the x-range of exactly one node at level i. Since $\mathcal{T}$ has $\alpha_q(N/L)$ levels, the total space for $\mathcal{T}$ is $cNq\alpha_q(N/L)$, as required.

We now show how queries are answered. Given a query with range $r = [x_1, x_2] \times [y, \infty)$ and color set Q, we begin by finding the lowest node u in $\mathcal{T}$ such that $x_1, x_2 \in I(u)$. We do so by predecessor search for x_1 and x_2 in $\mathcal{B}_v$ for each ancestor v of u, starting from the root of $\mathcal{T}$. This process takes $O(\sum_v(1 + \log(|\mathcal{B}_v|))) = O(\alpha_q(N/L) + \log N) = O(\log N)$ time.

If u is a leaf, we query the naive three-sided data structure stored at u with range r and color set Q and we are done in $O(|Q|\log(L/|Q|) + \log N + K)$ time. If u is an internal node, then we know that u has two different children v_1 and v_2 such that $x_1 \in I(v_1)$ and $x_2 \in I(v_2)$. Let x_1' be the left boundary of the x-range of the right sibling of v_1 and let x_2' be the left boundary of $I(v_2)$. We decompose r into three subranges $r^+ = [x_1, x_1') \times [y, \infty)$, $r^{||} = [x_1', x_2') \times [y, \infty]$, and $r^- = [x_2', x_2] \times [y, \infty]$, as shown in Figure 2.

Amongst the points of S_{v_1}, the range r^+ contains the same points as $[x_1, \infty] \times [y, \infty]$, thus we handle this subrange in $O(|Q|\log(L/|Q|) + \log N + K)$ time via a two-sided query to $D_{v_1}^+$. We can handle r^- symmetrically with a two-sided query to $D_{v_2}^-$. Finally, we handle $r^{||}$ by recursing in $D_u^{||}$ with the range $r^{||}$. At each level of recursion we perform $O(|Q|\log(L/|Q|) + \log N)$ work that can't be charged to the output size. So, the total query time is $O(p(|Q|\log(L/|Q|) + \log N) + K)$.

We can reduce the $O(p \log N)$ term to $O(\log N)$ by fractional cascading. The $O(p \log N)$ term comes from two $O(\log N)$ time algorithms that we invoke at each of the $O(p)$ levels of recursion: first, finding the node u in $\mathcal{T}$ at which we can decompose r into an aligned query and two-sided queries, and second, the

two-sided CRR queries. Both of these algorithms amount to predecessor search for x_1 and x_2 in different sets of elements. In finding node u we search for x_1 and x_2 in $\mathcal{B}_v$ for each ancestor v of u. The two-sided CRR query algorithms search for x_1 or x_2 in layers of maxima.

We construct a global catalog graph for all $O(p)$ levels of recursion. Each node u in a tree $\mathcal{T}$ at recursion level q is associated with a catalog C_u that contains the elements of $\mathcal{B}_u$. If u is the root of $\mathcal{T}$, then C_u is linked to C_v for some node v in recursion level $q+1$ that uses $\mathcal{T}$ in $D_v^{||}$. If u is not the root of $\mathcal{T}$, then C_u is instead linked to C_v where v is the parent of u. If u is an internal node, then C_u is linked to C_v where v is the root of a tree at recursion level $q-1$ in $D_u^{||}$ as well as C_w for each child w of u. In this case, C_u is also linked to the catalogs for the first layers of maxima in D_u^+ and D_u^-. Our catalog graph thus has locally bounded degree of 5: although there are a non-constant number of links to children, they only contribute to an increase of the locally bounded degree by one since the ranges of the catalogs of the children are all disjoint.

An issue that we have thus far ignored is that, when we recurse, we change the query x-coordinates from x_1 and x_2 to x_1' and x_2'. Typically, in fractional cascading, one searches for the same query elements in every catalog. However, it is straightforward to extend fractional cascading without penalty so that it is possible to switch the query value to the result of a previous search in an adjacent catalog. When we recurse, this is precisely what we need to do: x_1' and x_2' are the results of searches in $\mathcal{B}_u$ (now C_u) and these values become the new query values in the root of the tree for $D_u^{||}$.

Now, we must count the number of catalogs that we visit to give a bound on running time. At each level of the tree at each recursion level, there are a constant number of catalogs that we search and cannot charge to the output size. Thus, the $O(p \log N)$ term is reduced by fractional cascading to $O(\log N) + \sum_{q=4}^{p} O(\alpha_q(N/L)) = O(\log N)$. This completes the proof of Theorem 4.

By fixing p to a constant in Theorem 4, we obtain trade-off 1 of Theorem 1. To get the other two trade-offs, we need one final step to reduce space to $O(N)$.

Theorem 5. *There exists a data structure for three-sided CRR that requires $O(N)$ space and $O(|Q|(p \log(L/|Q|) + \log(p\alpha_p(N/L))) + \log N + K)$ query time, for any $p \geq 3$.*

Proof. We divide our set of points S into columns of $O(Lp\alpha_p(N/L))$ points. To handle queries that lie entirely within one of these columns, we build the naive data structure for three-sided queries in each column. These data structures require linear space and $O(|Q| \log(Lp\alpha_p(N/L)/|Q|) + K) = O(|Q|(\log(L/|Q|) + \log(p\alpha_p(N/L))) + K)$ query time. To handle queries that do not lie entirely within one column, we reuse the technique of Theorem 4 to decompose the query into two-sided queries in two columns and an aligned query. By Lemma 5, where $x = O(Lp\alpha_p(N/L))$ and D is the data structure of Theorem 4, we can handle aligned queries in linear space and $O(|Q|p \log(N/L) + \log N + K)$ query time. □ □

By fixing p to a constant in Theorem 5, we obtain trade-off 2 of Theorem 1. Finally, by setting $p = \alpha(N/L)$, we obtain trade-off 3. This completes the proof of Theorem 1.

5 Dynamic One-Dimensional CRR

This section proves Theorem 2. All BSTs in this paper are slightly augmented, such that there is a doubly-linked list organizing the nodes in their sorted order, so that the predecessor/successor of any node can be found in constant time. During an insertion/deletion, maintaining the linked list causes only $O(1)$ extra time.

5.1 Structure

As before, let S be the set of points in the input to the *one-dimensional* CRR. We will refer to each point in S as a *key*. For convenience, assume that $-\infty$ is a dummy key in S with an arbitrary color. The base tree of our structure is a BST $\mathcal{T}$ on S. Each node u of $\mathcal{T}$ has capacity L, namely, it accommodates a set S_u

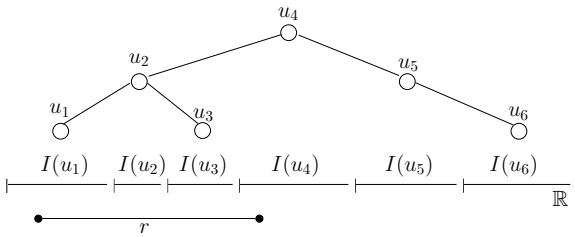


Figure 3: A BST with fat nodes

of $\Theta(L)$ consecutive keys[4] in S. $\mathcal{T}$ is created on the obvious total order of all the nodes. The ordering also associates each node u with a range $I(u) \subseteq \mathbb{R}$ in the form of $[\alpha, \beta)$, where α is the smallest key in u, and β is the smallest key in the node succeeding u (if such a node does not exist, $\beta = \infty$). See Figure 3. For each node u in $\mathcal{T}$, we organize the keys of S_u in L BSTs $T_u(1), ..., T_u(L)$, where $T_u(i)$ manages the color-i keys in S_u. Index the root addresses of $T_u(1), ..., T_u(L)$ in a BST $\mathcal{R}_u$.

Denote by D_u the set of keys stored in the subtree of u. Let $\ell(u)$ ($\gamma(u)$) be the left (right) child of node u, or $\emptyset$ if the child does not exist. Define $4L$ *boundary keys* as follows:

- For each $i \in [L]$, $B_u^{\ell-}(i)$ ($B_u^{\ell+}(i)$) is the minimum (maximum) key of color i in $D_{\ell(u)}$. If $D_{\ell(u)}$ has no key of this color, $B_u^{\ell-}(i) = B_u^{\ell+}(i) = \emptyset$.
- $B_u^{\gamma-}(i)$ and $B_u^{\gamma+}(i)$ are defined analogously with respect to $D_{\gamma(u)}$.

The $4L$ boundary keys can be divided into L *boundary groups*, where the i-th group is a 4-tuple $(B_u^{\ell-}(i), B_u^{\ell+}(i), B_u^{\gamma-}(i), B_u^{\gamma+}(i))$. These groups are indexed by their colors with a BST $\mathcal{B}_u$ (i.e., each node of $\mathcal{B}_u$ stores a boundary group). For update efficiency, we keep some *down pointers*. Specifically for each $i \in [L]$, the color-i node of $\mathcal{B}_u$ stores two down pointers referencing the color-i nodes in $\mathcal{B}_{\ell(u)}$ and $\mathcal{B}_{\gamma(u)}$, respectively. If $\ell(u)$ ($\gamma(u)$) $= \emptyset$, the down pointer to $\mathcal{B}_{\ell(u)}$ ($\mathcal{B}_{\gamma(u)}$) is *nil*. Finally, the keys of each S_i are indexed with a BST Σ_i, called an *exclusive BST*. Each key k in $\mathcal{T}$ (including boundary keys) is associated with a *shortcut pointer*, referencing node k in the corresponding exclusive BST.

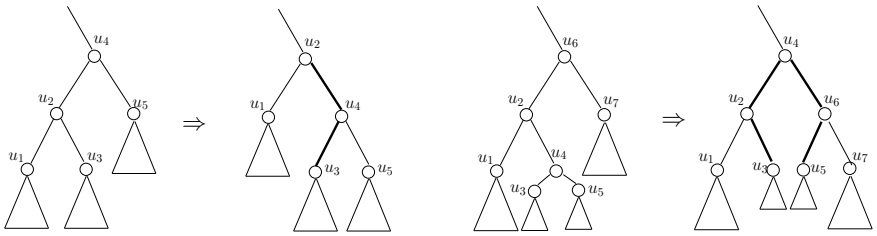
All the exclusive BSTs occupy totally $O(N)$ space. At each node u of $\mathcal{T}$, all the secondary structures require $O(|S_u| + L)$ space. Hence, the overall size of our structure is $\sum_u O(|S_u| + L) = O(N)$, noticing that $\mathcal{T}$ has $O(N/L)$ nodes.

5.2 Insertion

Let all the BSTs be implemented as AVL-trees. For any non-root node u, denote its parent as $\rho(u)$. Suppose that we need to insert a key k of color $c \in [L]$. First, k is added to Σ_c in $O(\log N)$ time. Then, we identify the node v in $\mathcal{T}$ such that $k \in I(v)$, which costs $O(\log \frac{N}{L})$ time, i.e., same as the height of $\mathcal{T}$. For each node u on the path from $root(\mathcal{T})$ to v, update two of its boundary keys of color c. To illustrate, assume that v is in the left (the case of right is symmetric) subtree of u; we adjust the boundary key $B_u^{\ell-}(c)$ to $\min\{B_u^{\ell-}(c), k\}$, and $B_u^{\ell+}(c)$ to $\max\{B_u^{\ell+}(c), k\}$. These boundary keys can be found efficiently using the down pointers. First, the (color-c) boundary group of $root(\mathcal{T})$ is retrieved by searching $\mathcal{B}_{root(\mathcal{T})}$ in $O(\log L)$ time. In general, having found a boundary group at a parent node $\rho(u)$, we follow a down pointer of the group to obtain the boundary group of the same color at u in $O(1)$ time. Hence, the entire process takes $O(\log L + \log \frac{N}{L}) = O(\log N)$ time.

At v, k is inserted into $T_v(c)$ in $O(\log L)$ time. If $|S_v|$ is now at most L, the insertion is completed. Otherwise, v overflows, and needs to be split into v_1 and v_2. For this purpose, we sort all the keys in S_v in $O(L \log L)$ time, divide the sorted list in two equal halves, and take the first (second) half as S_{v_1} (S_{v_2}). Then, the secondary BSTs $T_{v_1}(1), ..., T_{v_1}(L), \mathcal{R}_{v_1}$ and $T_{v_2}(1), ..., T_{v_2}(L), \mathcal{R}_{v_2}$ can be built in $O(L)$ time. Nodes v_1

[4]The only exception is when $\mathcal{T}$ has only one node u, in which case $|S_u|$ can be anywhere from 1 to L.



(a) Single rotation (b) Double rotation

Figure 4: Rotations in an AVL-tree

and v_2 need to be incorporated in $\mathcal{T}$. For this purpose, we first replace v with v_1 in $\mathcal{T}$ and set $\mathcal{B}_{v_1}$ directly to $\mathcal{B}_v$, which takes $O(1)$ time. Then, v_2 is inserted in the right subtree of v (now v_1) as in the BST. When this is done, v_2 is a leaf of $\mathcal{T}$, and some boundary keys of the nodes on the path from $\rho(v_2)$ to v_1 may now be incorrect. A *boundary correction* is carried out to fix them in a bottom-up manner:

- Suppose, without loss of generality, that v_2 is the left child of $\rho(v_2)$. For each $i \in [L]$, set $B^{\ell-}_{\rho(v_2)}(i) = mink(S_{v_2}, i)$ and $B^{\ell+}_{\rho(v_2)}(i) = maxk(S_{v_2}, i)$, where $mink(X, i)$ and $maxk(X, i)$ return the smallest and largest color-i key in a set X of keys, respectively. We then build $\mathcal{B}_{\rho(v_2)}$ in $O(L)$ time.
- In general, having fixed node u, the correction process works at $\rho(u)$ as follows, again assuming u to be the left child of $\rho(u)$. For each $i \in [L]$, we set:

$$\begin{aligned} B^{\ell-}_{\rho(u)}(i) &= \min\{B^{\ell-}_u(i), mink(S_u, i), B^{\gamma-}_u(i)\} \\ B^{\ell+}_{\rho(u)}(i) &= \max\{B^{\ell+}_u(i), maxk(S_u, i), B^{\gamma+}_u(i)\}. \end{aligned}$$

The correction spends $O(L)$ time per level, noticing that $mink(S_u, i)$ and $maxk(S_u, i)$ of all $i \in [L]$ can be retrieved in $O(\sum_i \log |T_u(i)|) = O(L)$ time, where $|T_u(i)|$ is the number of nodes in $T_u(i)$.

If $\mathcal{T}$ is unbalanced, we perform a single- or double-rotation as in the AVL-tree. Each rotation modifies $O(1)$ pointers, as illustrated by bold lines in Figure 4. In our context, a rotation is followed by boundary correction. Specifically, in Figure 4a, we launch a correction process to fix the boundary keys from u_4 to u_2 whereas, in Figure 4b, two processes are invoked to fix the paths from u_2 to u_4 and from u_6 to u_4, respectively. Since only two levels are corrected, all processes take $O(L)$ time. The above description has not included the maintenance of down and shortcut pointers, but this can be easily taken care of at no extra cost asymptotically.

If no overflow happens, an insertion terminates in $O(\log N)$ time. Otherwise, the cost is $O(L \log N)$. However, as a node u overflows only after $\Omega(L)$ keys have been newly inserted in S_u, each update is charged $O(\log N)$ time.

5.3 Deletion

Suppose that we are deleting a key k of color $c \in [L]$. The deletion algorithm starts by removing k from Σ_c in $O(\log N)$ time. Then, we find in $O(\log \frac{N}{L})$ time the node v in $\mathcal{T}$ such that $k \in I_v$, and remove k from $T_v(c)$ in $O(\log L)$ time. As k may be serving as a boundary key at the nodes on the path from v to $root(\mathcal{T})$, we fix their boundary keys in a way similar to a boundary correction process, except that here we focus on only a unique color c. Using down pointers, all the boundary keys to be fixed can be identified in $O(\log N)$ time.

If now S_v has at least $L/4$ keys, the deletion is completed. Otherwise, v underflows, which is treated by a *share* or *merge* (reminiscent of underflow-handing in a B-tree). Specifically, the algorithm first identifies in $O(\log \frac{N}{L})$ time the successor v' of v in $\mathcal{T}$. If $|S_v| + |S_{v'}| > \frac{7}{8}L$, we perform a share, which re-distributes the keys in S_v and $S_{v'}$ evenly. This, in turn, necessitates rebuilding T_v and $T_{v'}$, and a boundary correction process from either v or v' (whichever is lower) to $root(\mathcal{T})$. The total time required is $O(L \log L)$.

On the other hand, if $|S_v| + |S_{v'}| \leq \frac{7}{8}L$, we perform a merge, which combines the keys of $S_{v'}$ into S_v, and then re-builds T_v. This costs $O(L)$ time. Next, the algorithm removes node v' from $\mathcal{T}$. Specifically, if v' is a leaf, we simply delete it; otherwise, if v' has (or does not have) a right subtree, we replace it by its successor (or predecessor) v'' in $\mathcal{T}$. In any case, we carry out two boundary correction processes on the paths from $\rho(v'')$ to $root(\mathcal{T})$, and from v to $root(\mathcal{T})$, respectively. $\mathcal{T}$ may have become unbalanced, which is remedied by rotations (as in insertion) using $O(L \log N)$ time. Hence, a merge can be accomplished in $O(L \log N)$ time. Overall a deletion requires $O(\log N)$ time amortized.

5.4 Query

Given a CRR query with search interval r and color set Q, we find in $O(\log N)$ time the highest node u in $\mathcal{T}$ whose $I(u)$ intersects r (e.g., in Figure 3, $u = u_4$ for the r shown). Search $\mathcal{R}_u$ to fetch the root addresses of all $T_u(i)$ with $i \in Q$, which takes $O(|Q| \log \frac{L}{|Q|})$ time (this is a concurrent lookup operation). Perform a range query in each such tree to report the keys appearing in r. As the $|Q|$ trees manage at most L keys in total, the $|Q|$ range queries incur $O(|Q| \log \frac{L}{|Q|})$ time plus the linear cost for result outputting.

Next, we retrieve the points in the query result that are outside $I(u)$. If r contains at least an endpoint of $I(u)$, search $\mathcal{B}_u$ to retrieve the boundary keys $B_u^{\ell+}(i)$ and $B_u^{\gamma-}(i)$ for each color $i \in Q$. This again takes $O(|Q| \log \frac{L}{|Q|})$ time. For each color $i \in Q$, if neither $B_u^{\ell+}(i)$ nor $B_u^{\gamma-}(i)$ is covered by r, we can assert that no other key of color i appears in r. Otherwise, assume $B_u^{\ell+}(i) \in r$ (the case with $B_u^{\gamma-}(i)$ is similar); we follow the shortcut pointer of $B_u^{\ell+}(i)$ to its node in Σ_i, and retrieve all the qualifying keys of color i in linear time (avoid reporting the keys in $I(u)$ though). The overall query cost is therefore $O(|Q| \log \frac{L}{|Q|} + \log N + K)$.

6 Three-Sided CRR for $L = O(\log N)$

There is a solution to three-sided CRR that requires linear space and $O(2^L L + \log N + K)$ query time [2]. This solution is optimal for $L \leq \log \log N - \log \log \log N$, since in this case $2^L L = O(\log N)$. By tackling the three-sided CRR problem from another angle, we obtain a solution that is optimal for greater settings of L: namely, $L = O(\log N)$. We start with a simple dynamic 1d data structure:

Lemma 6. *For $L = O(\log N)$, there is a structure that consumes $O(N)$ space, answers a 1d CRR query in $O(\log N + K)$ time, and handles an insertion or a deletion in $O(\log N)$ time.*

Proof. Borrowing an idea in [8], we will need an interval tree [14], which indexes N intervals in $\mathbb{R}$ using $O(N)$ space such that, given a value q, all intervals containing q can be reported in $O(\log N + K)$ time, where K is the number of such intervals. This is known as a *stabbing query*. Insertion/deletion of an interval can be done in $O(\log N)$ time.

For each S_i, sort its keys in ascending order, and create $|S_i|$ intervals as follows. The first interval starts from $-\infty$ and ends at the smallest key in S_i. In general, the j-th interval ($2 \leq j \leq |S_i|$) starts from the $(j-1)$-th key (of the sorted order) and ends at the j-th key. Build an interval tree T on all the intervals thus obtained from $S_1, ..., S_L$. Since there are $O(N)$ intervals, T uses $O(N)$ space. Given a CRR query with $r = [x, y]$ and a color set Q, we perform a stabbing query on T to retrieve all the intervals containing x. Since, for each $i \in [L]$, at most two intervals generated from S_i are fetched, the query time is $O(\log N + L) = O(\log N)$. The successor of x in S_i ($i \in Q$) must be at a boundary of a retrieved interval that originated from S_i, and hence, can be determined in $O(1)$ time. After this, all the keys of $S_i \cap r$ can be reported in linear time. It is easy to see that when a key of any color is inserted/deleted, T can be maintained in $O(\log N)$ time. □ □

A data structure is usually *ephemeral* because, once updated, the previous version is lost. In contrast, a *persistent* structure Γ^p retains all the historical versions of an ephemeral structure Γ, after Γ has gone through a sequence of updates. Driscoll et al. [13] developed a technique to make a structure Γ persistent, provided that Γ can be modeled as a graph where each node has constant in-degree. Their technique yields a Γ^p whose space grows on average by $O(s)$ after each update, where s is the average footprint per update on

Γ. Γ^p can be constructed in the same time as performing all the corresponding updates on Γ. Furthermore, Γ^p inherits the query complexity of Γ, with only an $O(\log N)$ additive cost to identify the correct version.

In general, three-sided range searching can be supported by the persistent counterpart of a structure designed for 1d range searching. Imagine that we move a horizontal sweeping line downwards, starting from the top of $\mathbb{R}^2$. Whenever the line hits a point $p \in S$, insert the x-coordinate of p in a 1d-CRR structure Γ. Denote by $\Gamma(\lambda)$ the version of Γ when the sweeping line intersects the y-axis at λ. Given a three-sided query with $r = [x_1, x_2] \times [y, \infty)$, we answer it by performing a 1d-CRR query with $r = [x_1, x_2]$ on $\Gamma(y)$. Based on this idea, we prove:

Lemma 7. *For $L = O(\log N)$, there is a linear-space structure that answers a three-sided CRR query in $O(\log N + K)$ time, and can be built in $O(N \log N)$ time.*

Proof. Our structure in Lemma 6 is indeed a graph where nodes have constant in-degree. As shown in [6], each insertion in the interval tree takes $O(\log N)$ time, and leaves only $O(1)$ footprint on average[5]. □ □

7 Open Problems

It remains open whether three-sided CRR can be solved in linear space and $O(|Q| \log(L/|Q|) + \log N + K)$ query time. In the RAM model, 1d CRR can be solved in linear space and $O(|Q| + K)$ query time, since traditional 1d range searching can be solved in linear space and $O(1 + K)$ query time [5]. However, the same cannot be said for the concurrent predecessor search problem, since traditional predecessor search in the RAM model using polynomial space requires super-constant query time [18]. It is open whether concurrent predecessor search can be solved in linear space and $O(T_{\text{pred}} + |Q| + K)$ query time, where T_{pred} is the cost of traditional predecessor search in the RAM model.

Another problem for which a concurrent variant is interesting is point location. In concurrent point location we must preprocess L subdivisions of the plane $S_1, S_2, \ldots, S_L$. Given a query point q and a set of colors $Q \subseteq [L]$, we must locate q in S_i for $i \in Q$. This concurrent point location problem is closely related to a 2d generalization of fractional cascading studied by Chazelle and Liu [11]. Chazelle and Liu show that no efficient bounds can be achieved for the 2d generalization of fractional cascading without using at least quadratic space. However, the hard instance in their lower bound uses non-orthogonal subdivisions of the plane. In the special case of concurrent orthogonal point location, there is a simple linear-space data structure that requires $O(L + \log N + K)$ query time: decompose the subdivisions into rectangles and store these rectangles in the rectangle-stabbing data structure of Chazelle [8]. It is open whether the L term in this query time can be reduced to $O(|Q| \log(L/|Q|))$ as in our CRR data structures.

References

[1] P. Afshani. Improved pointer machine and I/O lower bounds for simplex range reporting and related problems. In *Symposium on Computational Geometry (SoCG)*, pages 339–346, 2012.

[2] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting in three and higher dimensions. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 149–158, 2009.

[3] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *Symposium on Computational Geometry (SoCG)*, pages 240–246, 2010.

[4] P. K. Agarwal, S. Govindarajan, and S. Muthukrishnan. Range searching in categorical data: Colored range searching on grid. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 17–28, 2002.

[5] The result of [6] holds in the *offline* setting, i.e., all the insertions are known in advance. This is sufficient for our purpose here.

[5] S. Alstrup, G. S. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.

[6] A. Boroujerdi and B. M. E. Moret. Persistency in computational geometry. In *Proceedings of the Canadian Conference on Computational Geometry (CCCG)*, pages 241–246, 1995.

[7] B. Chazelle. On the convex layers of a planar set. *IEEE Transactions on Information Theory*, 31(4):509–517, 1985.

[8] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal of Computing*, 15(3):703–724, 1986.

[9] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[10] B. Chazelle and L. J. Guibas. Fractional cascading: II. applications. *Algorithmica*, 1(2):163–191, 1986.

[11] B. Chazelle and D. Liu. Lower bounds for intersection searching and fractional cascading in higher dimension. *Journal of Computer and System Sciences (JCSS)*, 68(2):269–284, 2004.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[13] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences (JCSS)*, 38(1):86–124, 1989.

[14] H. Edelsbrunner. A new approach to rectangle intersections, part I. *International Journal of Computer Mathematics*, 13:209–219, 1983.

[15] E. M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.

[16] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5(2):215–241, 1990.

[17] M. H. Overmars. Efficient data structures for range searching on a grid. *J. Algorithms*, 9(2):254–275, 1988.

[18] M. Patrascu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.