# Optimal Color Range Reporting in One Dimension

Yakov Nekrich[1] and Jeffrey Scott Vitter[1]

The University of Kansas. `yakov.nekrich@googlemail.com, jsv@ku.edu`

**Abstract.** Color (or categorical) range reporting is a variant of the orthogonal range reporting problem in which every point in the input is assigned a *color*. While the answer to an orthogonal point reporting query contains all points in the query range $Q$, the answer to a color reporting query contains only distinct colors of points in $Q$. In this paper we describe an $O(N)$-space data structure that answers one-dimensional color reporting queries in optimal $O(k+1)$ time, where $k$ is the number of colors in the answer and $N$ is the number of points in the data structure. Our result can be also dynamized and extended to the external memory model.

## 1 Introduction

In the orthogonal range reporting problem, we store a set of points $S$ in a data structure so that for an arbitrary range $Q = [a_1, b_1] \times \ldots \times [a_d, b_d]$ all points from $S \cap Q$ can be reported. Due to its importance, one- and multi-dimensional range reporting was extensively studied in computational geometry and database communities. The following situation frequently arises in different areas of computer science: a set of $d$-dimensional objects $\{\,(t_1, t_2, \ldots, t_d)\,\}$ must be preprocessed so that we can enumerate all objects satisfying $a_i \le t_i \le b_i$ for arbitrary $a_i, b_i$, $i = 1, \ldots, d$. This scenario can be modeled by the orthogonal range reporting problem.

The objects in the input set can be distributed into *categories*. Instead of enumerating all objects, we may want to report distinct categories of objects in the given range. This situation can be modeled by the color (or categorical) range reporting problem: every point in a set $S$ is assigned a color (category); we pre-process $S$, so that for any $Q = [a_1, b_1] \times \ldots \times [a_d, b_d]$ the distinct colors of points in $S \cap Q$ can be reported.

Color range reporting is usually considered to be a more complex problem than point reporting. For one thing, we do not want to report the same color multiple times. In this paper we show that complexity gap can be closed for one-dimensional color range reporting. We describe color reporting data structures with the same space usage and query time as the best known corresponding structures for point reporting. Moreover we extend our result to the external memory model.

*Previous Work.* We can easily report points in a one-dimensional range $Q = [a, b]$ by searching for the successor of $a$ in $S$, $succ(a, S) = \min\{\, e \in S \mid e \geq a \,\}$. If $a' = succ(a, S)$ is known, we can traverse the sorted list of points in $S$ starting at $a'$ and report all elements in $S \cap [a, b]$. We can find the successor of $a$ in $S$ in $O(\sqrt{\log N/ \log\log N})$ time [4]; if the universe size is $U$, i.e., if all points are positive integers that do not exceed $U$, then the successor can be found in $O(\log\log U)$ time [22]. Thus we can report all points in $S \cap [a, b]$ in $O(\text{tpred}(N) + k)$ time for $\text{tpred}(N) = \min(\sqrt{\log N/ \log\log N}, \log\log U)$. Henceforth $k$ denotes the number of elements (points or colors) in the query answer. It is not possible to find the successor in $o(\text{tpred}(N))$ time unless the universe size $U$ is very small or the space usage of the data structure is very high; see e.g., [4]. However, reporting points in a one-dimensional range takes less time than searching for a successor. In their fundamental paper [14], Miltersen et al. showed that one-dimensional point reporting queries can be answered in $O(k)$ time using an $O(N \log U)$ space data structure. Alstrup et al. [1] obtained another surprising result: they presented an $O(N)$-space data structure that answers point reporting queries in $O(k)$ time and thus achieved both optimal query time and optimal space usage for this problem. The data structure for one-dimensional point reporting can be dynamized so that queries are supported in $O(k)$ time and updates are supported in $O(\log^{\varepsilon} U)$ time [16]; henceforth $\varepsilon$ denotes an arbitrarily small positive constant. We refer to [16] for further update-query time trade-offs. Solutions of the one-dimensional point reporting problem are based on finding an arbitrary element $e$ in a query range $[a, b]$; once such $e$ is found, we can traverse the sorted list of points until all points in $[a, b]$ are reported. Therefore it is straightforward to extend point reporting results to the external memory model.

Janardan and Lopez [10] and Gupta et al. [9] showed that one-dimensional color reporting queries can be answered in $O(\log N + k)$ time, both in the static and the dynamic scenarios. Muthukrishnan [17] described a static $O(N)$ space data structure that answers queries in $O(k)$ time if all point coordinates are bounded by $N$. We can obtain data structures that use $O(N)$ space and answer queries in $O(\log\log U + k)$ or $O(\sqrt{\log N/ \log\log N} + k)$ time using the reduction-to-rank-space technique. No data structure that answers one-dimensional color reporting queries in $o(tpred(N)) + O(k)$ time was previously known. A dynamic data structure of Mortensen [15] supports queries and updates in $O(\log\log N + k)$ and $O(\log\log N)$ time respectively if the values of all elements are bounded by $N$.

Recently, the one- and two-dimensional color range reporting problems in the external memory model were studied in several papers [11, 18, 12]. Larsen and Pagh [11] described a data structure that uses linear space and answers one-dimensional color reporting queries in $O(k/B + 1)$ I/Os if values of all elements are bounded by $O(N)$. In the case when values of elements are unbounded the best previously known data structure needs $O(\log_B N + k/B)$ I/Os to answer a query; this result can be obtained by combining the data structure from [2] and reduction of one-dimensional color reporting to three-sided[1] point reporting [9].

In another recent paper [5], Chan et al. described a data structure that supports the following queries on a set of points whose values are bounded by $O(N)$: for any query point $q$ and any integer $k$, we can report the first $k$ colors that occur after $q$. This data structure can be combined with the result from [1] to answer queries in $O(k + 1)$ time. Unfortunately, the solution in [5] is based on the hive graph data structure [6]. Therefore it cannot be used to solve the problem in external memory or to obtain a dynamic solution.

*Our Results.* As can be seen from the above discussion and Table 1, there are significant complexity gaps between color reporting and point reporting data structures in one dimension. We show in this paper that it is possible to close these gaps.

In this paper we show that one-dimensional color reporting queries can be answered in constant time per reported color for an arbitrarily large size of the universe. Our data structure uses $O(N)$ space and supports color reporting queries in $O(k + 1)$ time. This data structure can be dynamized so that query time and space usage remain unchanged; the updates are supported in $O(\log^{\varepsilon} U)$ time where $U$ is the size of the universe. The new results are listed at the bottom of Table 1.

Our internal memory results are valid in the word RAM model of computation, the same model that was used in e.g. [1, 16, 17]. In this model, we assume that any standard arithmetic operation and the basic bit operations can be performed in constant time. We also assume that each word of memory consists of $w \geq \log U \geq \log N$ bits, where $U$ is the size of the universe. That is, we make a reasonable and realistic assumption that the value of any element fits into one word of memory.

Furthermore, we also extend our data structures to the external memory model. Our static data structure uses linear space and answers color reporting queries in $O(1+k/B)$ I/Os. Our dynamic external data structure also has optimal space usage and query cost; updates are supported in $O(\log^{\varepsilon} U)$ I/Os.

In Section 2 we describe a static data structure for color reporting in one dimension. The key component of our solution is a data structure that supports *highest range ancestor* queries. In Section 3 we show how our static data structure can be adopted to the external memory model. We show how to dynamize our data structure in Sections 4, 5, and 6. Details of our dynamic solution and its modification for the external memory model are provided in the full version of this paper [19].

[1] A three-sided range query is a two-dimensional orthogonal range query that is open on one side. For instance, queries $[a, b] \times [0, c]$ and $[a, b] \times [c, +\infty]$ are three-sdied queries.

| Ref. | Query Type | Space Usage | Query Cost | Universe | Update Cost |
|---|---|---|---|---|---|
| [1] | Point Reporting | $O(N)$ | $O(k+1)$ | | static |
| [16] | Point Reporting | $O(N)$ | $O(k+1)$ | | $O(\log^{\varepsilon} U)$ |
| [9, 10] | Color Reporting | $O(N)$ | $O(\log N + k)$ | | $O(\log N)$ |
| [17] | Color Reporting | $O(N)$ | $O(k+1)$ | $N$ | static |
| [17] | Color Reporting | $O(N)$ | $O(\log\log U + k)$ | $U$ | static |
| [17] | Color Reporting | $O(N)$ | $O(\sqrt{\log N/\log\log N} + k)$ | | static |
| [15] | Color Reporting | $O(N)$ | $O(\log\log N + k)$ | $N$ | $O(\log\log N)$ |
| [5]+[1] | Color Reporting | $O(N)$ | $O(k+1)$ | | |
| Our | Color Reporting | $O(N)$ | $O(k+1)$ | | static |
| Our | Color Reporting | $O(N)$ | $O(k+1)$ | | $O(\log^{\varepsilon} U)$ |

**Table 1.** Selected previous results and new results for one-dimensional color reporting. The fifth and the sixth row can be obtained by applying the reduction to rank space to the result from [17].

## 2 Static Color Reporting in One Dimension

We start by describing a static data structure that uses $O(N)$ space and answers color reporting queries in $O(k+1)$ time.

All elements of a set $S$ are stored in a balanced binary tree $\mathcal{T}$. Every leaf of $\mathcal{T}$, except for the last one, contains $\log N$ elements, the last leaf contains at most $\log N$ elements, and every internal node has two children. For any node $u \in \mathcal{T}$, $S(u)$ denotes the set of all elements stored in the leaf descendants of $u$. For every color $z$ that occurs in $S(u)$, the set $Min(u)$ ($Max(u)$) contains the minimal (maximal) element $e \in S(u)$ of color $z$. The list $L(u)$ contains the $\log N$ smallest elements of $Min(u)$ in increasing order. The list $R(u)$ contains the $\log N$ largest elements of $Max(u)$ in decreasing order. For every internal non-root node $u$ we store the list $L(u)$ if $u$ is the right child of its parent; if $u$ is the left child of its parent, we store the list $R(u)$ for $u$. All lists $L(u)$ and $R(u)$, $u \in \mathcal{T}$, contain $O(N)$ elements in total since the tree has $O(N/\log N)$ internal nodes.

We define the middle value $m(u)$ for an internal node $u$ as the minimal value stored in the right child of $u$, $m(u) = \min\{\, e \mid e \in S(u_r)\,\}$ where $u_r$ is the right child of $u$. The following *highest range ancestor* query plays a crucial role in the data structures of this and the following sections. The answer to the highest range ancestor query $(v_l, a, b)$ for a leaf $v_l$ and values $a < b$ is the highest ancestor $u$ of $v_l$, such that $a < m(u) \leq b$; if $S \cap [a,b] = \emptyset$, the answer is undefined. The following fact elucidates the meaning of the highest range ancestor.

**Fact 1** *Let $v_a$ be the leaf that holds the smallest $e \in S$, such that $e \geq a$; let $v_b$ be the leaf that holds the largest $e \in S$, such that $e \leq b$. Suppose that $S(v_l) \cap [a,b] \neq \emptyset$ for some leaf $v_l$ and $u$ is the answer to the highest range ancestor query $(v_l, a, b)$. Then $u$ is the lowest common ancestor of $v_a$ and $v_b$.*

*Proof*: Let $w$ denote the lowest common ancestor of $v_a$ and $v_b$. Then $v_a$ and $v_b$ are in $w$'s left and right subtrees respectively. Hence, $a < m(w) \leq b$ and $w$ is not an ancestor of $u$. If $w$ is a descendant of $u$ and $w$ is in the right subtree of $u$, then $m(u) \leq a$. If $w$ is in the left subtree of $u$, then $m(w) > b$. $\square$

We will show that we can find $u$ without searching for $v_a$ and $v_b$ and answer highest range ancestor queries on a balanced tree in constant time.

For every leaf $v_l$, we store two auxiliary data structures. All elements of $S(v_l)$ are stored in a data structure $D(v_l)$ that uses $O(|S(v_l)|)$ space and answers color reporting queries on $S(v_l)$ in $O(k+1)$ time. We also store a data structure $F(v_l)$ that uses $O(\log N)$ space; for any $a < b$, such that $S(v_l) \cap [a,b] \neq \emptyset$, $F(v_l)$ answers the highest range ancestor query $(v_l, a, b)$ in $O(1)$ time. Data structures $D(v_l)$ and $F(v_l)$ will be described later in this section. Moreover, we store all elements of $S$ in the data structure described in [1] that supports one-reporting queries: for any $a < b$, some element $e \in S \cap [a,b]$ can be found in $O(1)$ time; if $S \cap [a,b] = \emptyset$, the data structure returns a dummy element $\perp$. Finally, all elements of $S$ are stored in a slow data structure that uses $O(N)$ space and answers color reporting queries in $O(\log n + k)$ time. We can use e.g. the data structure from [10] for this purpose.

*Answering Queries.* All colors in a query range $[a,b]$ can be reported with the following procedure. Using the one-reporting data structure from [1], we search for some $e \in S \cap [a,b]$ if at least one such $e$ exists. If no element $e$ satisfying $a \leq e \leq b$ is found, then $S \cap [a,b] = \emptyset$ and the query is answered. Otherwise, let $v_e$ denote the leaf that contains $e$. Using $F(v_e)$, we search for the highest ancestor $u$ of $v_e$ such that $a \leq m(u) \leq b$. If no such $u$ is found, then all $e$, $a \leq e \leq b$, are in $S(v_e)$. We can report all colors in $S(v_e) \cap [a,b]$ using $D(v_e)$. If $F(v_e)$ returned some node $u$, we proceed as follows. Let $u_l$ and $u_r$ denote the left and the right children of $u$. We traverse the list $L(u_r)$ until an element $e' > b$ is found or the end of $L(u_r)$ is reached. We also traverse $R(u_l)$ until an element $e' < a$ is found or the end of $R(u_l)$ is reached. If we reach neither the end of $L(u_r)$ nor the end of $R(u_l)$, then the color of every encountered element $e \in L(u_r)$, $e \leq b$, and $e \in R(u_l)$, $e \geq a$, is reported. Otherwise the range $[a,b]$ contains at least $\log N$ different colors. In the latter case we can use any data structure for one-dimensional color range reporting [10, 9] to identify all colors from $S \cap [a,b]$ in $O(\log n + k) = O(k+1)$ time.

*Leaf Data Structures.* A data structure $D(v_l)$ answers color reporting queries on $S(v_l)$ as follows. In [9], the authors show how a one-dimensional color reporting query on a set of $m$ one-dimensional elements can be answered by answering a query $[a,b] \times [0,a]$ on a set of $m$ uncolored two-dimensional points. A standard priority search tree [13] enables us to answer queries of the form $[a,b] \times [0,a]$ on $m$ points in $O(\log m)$ time. Using a combination of fusion trees and priority search trees, described by Willard [23], we can answer queries in $O(\log m / \log\log N)$ time. The data structure of Willard [23] uses $O(m)$ space and a universal look-

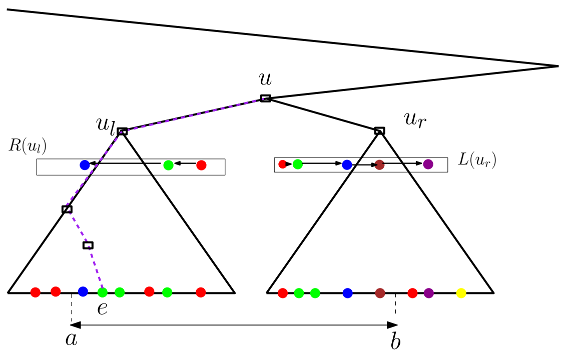

**Fig. 1.** Answering a color reporting query $Q = [a, b]$: $e$ is an arbitrary element in $S \cap [a, b]$, $u$ is the highest range ancestor of the leaf that contains $e$, the path from $e$ to $u$ is indicated by a dashed line. We assume $\log n = 5$, therefore $L(u_r)$ contains 5 elements and the yellow point is not included in $L(u_r)$. To simplify the picture, we assumed that each leaf contains only one point; only relevant parts of $\mathcal{T}$ are on the picture.

up table of size $O(\log^{\varepsilon} N)$ for an arbitrarily small $\varepsilon$. Updates are also supported in $O(\log m/\log\log N)$ time[2].

Since $S(v_l)$ contains $m = O(\log N)$ elements, we can answer colored queries on $S(v_l)$ in $O(\log m/\log\log N) = O(1)$ time. Updates are also supported in $O(1)$ time; this fact will be used in Section 4.

Now we describe how $F(v_l)$ is implemented. Suppose that $S(v_l) \cap [a, b] \neq \emptyset$ for some leaf $v_l$. Let $\pi$ be the path from $v_l$ to the root of $\mathcal{T}$. We say that a node $u \in \pi$ is a *left parent* if $u_l \in \pi$ for the left child $u_l$ of $u$; a node $u \in \pi$ is a *right parent* if $u_r \in \pi$ for the right child $u_r$ of $u$. If $S(v_l)$ contains at least one $e \in [a, b]$, then the following is true.

**Fact 2** *If $u \in \pi$ is a left parent, then $m(u) > a$. If $u \in \pi$ is a right parent, then $m(u) \leq b$.*

*Proof*: If $u \in \pi$ is a left parent, then $S(v_l)$ is in its left subtree. Hence, $m(u)$ is greater than any $e \in S(v_l)$ and $m(u) > a$. If $u$ is the right parent, than $S(v_l)$ is in its right subtree. Hence, $m(u)$ is smaller than or equal to any $e \in S(v_l)$ and $m(u) \leq b$. □

**Fact 3** *If $u_1 \in \pi$ is a left parent and $u_1$ is an ancestor of $u_2 \in \pi$, then $m(u_1) > m(u_2)$. If $u_1 \in \pi$ is a right parent and $u_1$ is an ancestor of $u_2 \in \pi$, then $m(u_2) > m(u_1)$.*

*Proof*: If $u_1$ is a left parent, then $u_2$ is in its left subtree. Hence, $m(u_1) > m(u_2)$ by definition of $m(u)$. If $u_2$ is a right parent, then $u_1$ is in its right subtree. Hence, $m(u_1) < m(u_2)$ by definition of $m(u)$. □

[2] In [23], Willard only considered queries on $N$ points, but extension to the case of any $m \leq N$ is straightforward.

Suppose that we want to find the highest range ancestor of $v_l$ for a range $[a, b]$ such that $S(v_l) \cap [a, b] \neq \emptyset$. Let $\mathcal{K}_1(\pi)$ be the set of middle values $m(u)$ for left parents $u \in \pi$ sorted by height; let $\mathcal{K}_2(\pi)$ be the set of $m(u)$ for right parents $u \in \pi$ sorted by height. By Fact 3, elements of $\mathcal{K}_1$ ($\mathcal{K}_2$) increase (decrease) monotonously. By Fact 2, $m(u) > a$ for any $m(u) \in \mathcal{K}_1$ and $m(u) < b$ for any $m(u) \in \mathcal{K}_2$. Using fusion trees [7], we can search in $\mathcal{K}_1$ and find the highest node $u_1 \in \pi$ such that $u_1$ is a left parent and $m(u_1) \leq b$. We can also search in $\mathcal{K}_2$ and find the highest node $u_2 \in \pi$ such that $u_2$ is a right parent and $m(u_2) > a$. Let $u$ denote the higher node among $u_1$, $u_2$. Then $u$ is the highest ancestor of $v_l$ such that $m(u) \in [a + 1, b]$.

*Removing Duplicates.* When a query is answered, our procedure returns a color $z$ two times if $z$ occurs in both $S \cap [a, m(u) - 1]$ and $S \cap [m(u), b]$. We can easily produce a list without sorting in which each color occurs exactly once. Let Col denote an array with one entry for every color that occurs in a data structure. Initially $\text{Col}[i] = 0$ for all $i$. We traverse the list of colors $\mathcal{L}$ produced by the above described procedure. Every time when we encounter a color $z$ in $\mathcal{L}$ such that $\text{Col}[z] = 0$, we set $\text{Col}[z] = 1$; when we encounter a color $z$ such that $\text{Col}[z] = 1$, we remove the corresponding entry from $\mathcal{L}$. When the query is answered, we traverse $\mathcal{L}$ once again and set $Col[z] = 0$ for all $z \in \mathcal{L}$.

**Theorem 1.** *There exists an $O(N)$-space data structure that supports one-dimensional color range reporting queries in $O(k + 1)$ time.*

## 3 Color Reporting in External Memory

The static data structure of Section 2 can be used for answering queries in external memory. We only need to increase the sizes of $S(v_l)$, $R(u)$, and $L(u)$ to $B \log_B N$, and use an external memory variant of the slow data structure for color reporting [2]. This approach enables us to achieve $O(1 + k/B)$ query cost, but one important issue should be addressed. As explained in Section 2, the same color can be reported twice when a query is answered. However, we cannot get rid of duplicates in $O(1 + k/B)$ I/Os using the method of Section 2 because of its random access to the list of reported colors. Therefore we need to make further changes in our internal memory solution. For an element $e \in S$, let $prev(e)$ denote the largest element $e' \leq e$ of the same color. For every element $e$ in $L(u)$ and any $u \in T$, we also store the value of $prev(e)$.

We define each set $S(v_l)$ for a leaf $v_l$ to contain $B \log_B N$ points. Lists $L(v)$ and $R(v)$ for an internal node $v$ contain $B \log_B N$ leftmost points from $Min(v)$ (respectively, $B \log_B N$ rightmost points from $Max(v)$). Data structures $F(v_l)$ are implemented as in Section 2. A data structure $D(v_l)$ supports color reporting queries on $S(v_l)$ and is implemented as follows. We can answer a one-dimensional color reporting query by answering a three-sided point reporting query on a set $\Delta$ of $|S(v_l)|$ two-dimensional points; see e.g., [9]. If $B \geq \log_2 N$, $S(v_l)$ and $\Delta$ contain $O(B^2)$ points. In this case we can use the data structure from [2] that uses linear space and answers three-sided queries in $O(\log_B |S(v_l)| + k/B) = O(1 + k/B)$

I/Os. If $B < \log N$, $S(v_l)$ and $\Delta$ contain $O(\log^2 N)$ points. Using the data structure from [7], we can find the predecessor of any value $v$ in a set of $O(\log^2 N)$ points in $O(1)$ I/Os. Therefore we can apply the rank-space technique [8] and reduce three-sided point reporting queries on $\Delta$ to three-sided point reporting queries on a grid of size $|\Delta|$ (i.e., to the case when coordinates of all points are integers bounded by $|\Delta|$) using a constant number of additional I/Os. Larsen and Pagh [11] described a linear-space data structure that answers three-sided point reporting queries for $m$ points on an $m \times m$ grid in $O(1 + k/B)$ I/Os. Summing up, we can answer a three-sided query on a set of $B \log_B N$ points in $O(1+k/B)$ I/Os. Hence, we can also answer a color reporting query on $S(v_l)$ in $O(1 + k/B)$ I/Os using linear space.

A query $Q = [a, b]$ is answered as follows. We find the highest range ancestor $u$ for any $e \in S \cap [a, b]$ exactly as in Section 2. If $u$ is a leaf, we answer the query using $D(u)$. Otherwise the reporting procedure proceeds as follows. We traverse the list $R(u_l)$ for the left child $u_l$ of $u$ until some point $p < a$ is found. If $e \geq a$ for all $e \in R(u_l)$, then there are at least $B \log_B N$ different colors in $[a, b]$ and we can use a slow data structure to answer a query in $O(\log_B N + \frac{k}{B}) = O(1 + \frac{k}{B})$ I/Os. Otherwise we traverse $L(u_r)$ and report all elements $e$ such that $prev(e) < a$. If $prev(e) \geq a$ for $e \in L(u)$, then an element of the same color was reported when $R(u_l)$ was traversed. Traversal of $L(u_r)$ stops when an element $e > b$ is encountered or the end of $L(u_r)$ is reached. In the former case, we reported all colors in $[a, b]$. In the latter case the number of colors in $[a, b]$ is at least $B \log_B N$. This is because every element in $L(u_r)$ corresponds to a distinct color that occurs at least once in $[a, b]$. Hence, we can use the slow data structure and answer the query in $O(\log_B N + \frac{k}{B}) = O(\frac{k}{B} + 1)$ I/Os.

**Theorem 2.** *There exists a linear-space data structure that supports one-dimensional color range reporting queries in $O(k/B + 1)$ I/Os.*

## 4 Base Tree for Dynamic Data Structure

In this section we show how the base tree and auxiliary data structures of the static solution can be modified for usage in the dynamic scenario. To dynamize the data structure of Section 2, we slightly change the balanced tree $\mathcal{T}$ and secondary data structures: every leaf of $\mathcal{T}$ now contains $\Theta(\log^2 N)$ elements of $S$ and each internal node has $\Theta(1)$ children. We store the lists $L(u)$ and $R(u)$ in each internal non-root node of $u$. We associate several values $m_i(u)$ to each node $u$: for every child $u_i$ of $u$, except the leftmost child $u_1$, $m_i(u) = \min\{\, e \mid e \in S(u_i) \,\}$. The highest range ancestor of a leaf $v_l$ is the highest ancestor $u$ of $v_l$ such that $a < m_i(u) \leq b$ for at least one $i \neq 1$. Data structures $D(v_l)$ and $F(v_l)$ are defined as in Section 2. We also maintain a data structure of [16] that reports an arbitrary element $e \in S \cap [a, b]$ if the range $[a, b]$ is not empty.

We implement the base tree $\mathcal{T}$ as the weight-balanced B-tree [3] with the leaf parameter $\log^2 N$ and the branching parameter 8. This means that every internal node has between 2 and 32 children and each leaf contains between $2 \log^2 N$ and $\log^2 N$ elements. Each internal non-root node on level $\ell$ of $\mathcal{T}$ has between $2 \cdot 8^{\ell} \log^2 N$ and $(1/2) \cdot 8^{\ell} \log^2 N$ elements in its subtree. If the number of elements in some node $u$ exceeds $2 \cdot 8^{\ell} \log^2 N$, we split $u$ into two new nodes, $u'$ and $u''$. In this case we insert a new value $m_i(w)$ for the parent $w$ of $u$. Hence, we may have to update the data structures $F(v_l)$ for all leaf descendants of $w$. A weight-balanced B-tree is engineered in such a way that a split occurs at most once in a sequence of $\Omega(8^{\ell} \log^2 N)$ insertions (for our choice of parameters). Since $F(v_l)$ can be updated in $O(1)$ time, the total amortized cost incurred by splitting nodes is $O(1)$. When an element $e$ is deleted, we delete it from the set $S(v_l)$. If $e = m_i(u)$ for a deleted element $e$ and some node $u$, we do not change the value of $m_i(u)$. We also do not start re-balancing if some node contains too few elements in its subtree. But we re-build the entire tree $\mathcal{T}$ if the total number of deleted elements equals $n_0/2$, where $n_0$ is the number of elements that were stored in $\mathcal{T}$ when it was built the last time. Updates can be de-amortized without increasing the cost of update operations by scheduling the procedure of re-building nodes (respectively, re-building the tree) [3]

*Auxiliary Data Structures.* We implement $D(v_l)$ in the same way as in Section 2. Hence color queries on $S(v_l)$ are answered in $O(\log |S(v_l)| / \log \log N) = O(1)$ time and updates are also supported in $O(1)$ time [23].

We need to modify data structures $F(v_l)$, however, because $\mathcal{T}$ is not a binary tree in the dynamic case. Let $\pi$ denote a path from $v_l$ to the root for some leaf $v_l$. We say that a node $u$ is an $i$-node if $u_i \in \pi$ for the $i$-th child $u_i$ of $u$.

**Fact 4** *Suppose that $S(v_l) \cap [a, b] \neq \emptyset$ and $\pi$ is the path from $v_l$ to the root. If $u \in \pi$ is an $i$-node, then $m_j(u) < b$ for $1 \leq j \leq i$ and $m_j(u) > a$ for $j > i$.*

We say that a value $m_j(u)$ for $u \in \pi$ is a *left value* if $j \leq i$ and $u$ is an $i$-node. A value $m_j(u)$ for $u \in \pi$ is a *right value* if $j > i$ and $u$ is an $i$-node.

**Fact 5** *If $m_j(u_1)$ is a left value and $u_1 \in \pi$ is an ancestor of $u_2 \in \pi$, then $m_j(u_1) \leq m_f(u_2)$ for any $f$. If $m_j(u_1)$ is a right value and $u_1 \in \pi$ is an ancestor of $u_2 \in \pi$, then $m_j(u_1) > m_f(u_2)$ for any $f$.*

It is easy to check Facts 4 and 5 using the same arguments as in Section 2.

We store all left values $m_j(u)$, $u \in \pi$, in a set $\mathcal{K}_1$; $m_j(u)$ in $\mathcal{K}_1$ are sorted by the height of $u$. We store all right values $m_j(u)$, $u \in \pi$, in a set $\mathcal{K}_2$; $m_j(u)$ in $\mathcal{K}_2$ are also sorted by the height of $u$. Using fusion trees on $\mathcal{K}_1$, we can find the highest node $u_1$, such that at least one left value $m_g(u_1) > a$. We can also find the highest $u_2$ such that at least one right value $m_f(u_2) \leq b$. Since $\mathcal{K}_1$ and $\mathcal{K}_2$ contain $O(\log N)$ elements, we can support searching and updates in $O(1)$ time; see [7, 21]. By Fact 4, $a < m_g(u_1) \leq b$ and $a < m_f(u_2) \leq b$. If $u$ is the higher node among $u_1$, $u_2$, then $u$ is an answer to the highest range ancestor query $[a, b]$ for a node $v_l$.

## 5 Fast Queries, Slow Updates

In this section we describe a dynamic data structure with optimal query time. Our improvement combines an idea from [15] with the highest range ancestor approach. We also use a new solution for a special case of two-dimensional point reporting problem presented in the full version of this paper [19] in Section A.1.

Let $\mathit{height}(u)$ denote the height of a node $u$. For an element $e \in S$ let $h_{\min}(e) = \mathit{height}(u')$, where $u'$ is the highest ancestor of the leaf containing $e$, such that $e \in \mathit{Min}(u')$. We define $h_{\max}(e)$ in the same way with respect to $\mathit{Max}(u)$. All colors in a range $[a, b]$ can be reported as follows. We identify an arbitrary $e \in S \cap [a, b]$. Using the highest range ancestor data structure, we can find the lowest common ancestor $u$ of the leaves that contain the successor of $a$ and the predecessor of $b$. Let $u_f$ and $u_g$ be the children of $u$ that contain the successor of $a$ and the predecessor of $b$. Let $a_f = a$, $b_g = b$; let $a_i = m_i(u)$ for $f < i \leq g$ and $b_i = m_{i+1}(u) - 1$ for $f \leq i < g$. We can identify unique colors of relevant points stored in each node $u_j$, $f < j \leq g$, by finding all $e \in [a_j, b_j]$ such that $e \in \mathit{Min}(u_j)$. This condition is equivalent to reporting all $e \in [a_j, b_j]$ such that $h_{\min}(e) \geq \mathit{height}(u_j)$. We can identify all colors of relevant points in $u_f$ by reporting all $e \in [a_f, b_f]$ such that $h_{\max}(e) \geq \mathit{height}(u_f)$. Queries of the form $e \in [a, b]$, $h_{\min}(e) \geq c$, (respectively $e \in [a, b]$, $h_{\max}(e) \geq c$) can be supported using Lemma 1 (see Section A.1 in [19]). While the same color can be reported several times, we can get rid of duplicates as explained in Section 2.

When a new point is inserted into $S$ or when a point is deleted from $S$, we can update the values of $h_{\min}(e)$ and $h_{\max}(e)$ in $O(\log \log U)$ time. We refer to [15, 20] for details.

While updates of data structures of Lemma 1 are fast, re-balancing the base tree can be a problem. As described in Section 4, when the number of points in a node $u$ on level $\ell$ exceeds $2 \cdot 8^{\ell} \log N$, we split it into two nodes, $u'$ and $u''$. As a result, the values $h_{\min}(e)$ for $e$ stored in the leaves of $u''$ can be incremented. Hence, we would have to examine the leaf descendants of $u''$ and recompute their values for some of them. Since the height of $\mathcal{T}$ is logarithmic, the total cost incurred by re-computing the values $h_{\min}(e)$ and $h_{\max}(e)$ is $O(\log N)$. The problem of reducing the cost of re-building the tree nodes is solved as follows. In Appendix A.2 in [19] we describe another data structure that supports fast updates but answering queries takes polynomial time in the worst case. In Section 6 we show how the cost of splitting can be reduced by modifying the definition of $h_{\min}(e)$, $h_{\max}(e)$ and using the slow data structure from [19] when the number of reported colors is sufficiently large.

## 6 Fast Queries, Fast Updates

Let $n(u)$ denote the number of leaves in the subtree of a node $u$. Let $\mathit{Left}(u)$ denote the set of $(n(u))^{1/2}$ smallest elements in $\mathit{Min}(u)$; let $\mathit{Right}(u)$ denote the set of $(n(u))^{1/2}$ largest elements in $\mathit{Max}(u)$. We maintain the values $\overline{h_{\min}}(e)$ and $\overline{h_{\max}}(e)$ for $e \in S$, such that for any $u \in \mathcal{T}$ we have: $\overline{h_{\min}}(e) = h_{\min}(e)$ if $e \in \mathit{Left}(u)$ and $\overline{h_{\min}}(e) \leq h_{\min}(e)$ if $e \in S(u) \setminus \mathit{Left}(u)$; $\overline{h_{\max}}(e) = h_{\max}(e)$ if $e \in \mathit{Right}(u)$ and $\overline{h_{\max}}(e) \leq h_{\max}(e)$ if $e \in S(u) \setminus \mathit{Right}(u)$. We keep $\overline{h_{\min}}(e)$ and $\overline{h_{\max}}(e)$ in data structures of Lemma 1. We also maintain the data structure described in Section A.2 in [19]. This data structure is used to answer queries when the number of colors in the query range is large. It is also used to update the values of $\overline{h_{\min}}(e)$ and $\overline{h_{\max}}(e)$ when a node is split.

To answer a query $[a, b]$, we proceed in the same way as in Section 5. Let $u$, $u_f$, $u_g$, and $a_i$, $b_i$, $f \le i \le g$ be defined as in Section 5. Distinct colors in each $[a_i, b_i]$, $f \le i \le g$, can be reported using the data structure of Lemma 1. If the answer to at least one of the queries contains at least $(n(u_i))^{1/2}$ elements, then there are at least $(n(u_i))^{1/2}$ different colors in $[a, b]$. The total number of elements in $[a, b] \cap S$ does not exceed $n(u) = 16n(u_j)$. Hence, we can employ the data structure from Section A.2 in [19] to report all colors from $[a, b]$ in $O(([a, b] \cap S)^{1/2} + k) = O(k)$ time. If answers to all queries contain less than $(n(u_i))^{1/2}$ elements, then for every distinct color that occurs in $[a, b]$ there is an element $e$ such that $e \in Left(u_i) \cap [a_i, b_i]$, $f \le i < g$, or $e \in Right(u_g) \cap [a_g, b_g]$. By definition of $\overline{h_{\min}}$ and $\overline{h_{\max}}$ we can correctly report up to $(n(u_i))^{1/2}$ leftmost colors in $Left(u_i)$ or up to $(n(u_i))^{1/2}$ rightmost colors in $Right(u_i)$.

When a new element $e$ is inserted, we compute the values of $h_{\min}(e)$, $h_{\max}(e)$ and update the values of $h_{\min}(e_n)$, $h_{\max}(e_n)$, where $e_n$ is the element of the same color as $e$ that follows $e$. This can be done in the same way as in Section 5. When a node $u$ on level $\ell$ is split into $u'$ and $u''$, we update the values of $h_{\min}(e)$ and $h_{\max}(e)$ for $e \in S(u') \cup S(u'')$. If $\ell \le \log\log N$, we examine all $e \in S(u') \cup S(u'')$ and re-compute the values of $prev(e)$, $h_{\min}(e)$, and $h_{\max}(e)$. Amortized cost of re-building nodes $u$ on $\log\log N$ lowest tree levels is $O(\log\log N)$. If $\ell > \log\log N$, $S(u)$ contains $\Omega(\log^5 N)$ elements. We can find $(n(u'))^{1/2}$ elements in $Left(u')$, $Left(u'')$, $Right(u')$, and $Right(u'')$ using the data structure from Lemma 3 in [19]. This takes $O((n(u)^{1/2}) \log N + \log N \log\log N) = O(((n(u))^{7/10})$ time. Since we split a node $u$ one time after $\Theta(n(u))$ insertions, the amortized cost of splitting nodes on level $\ell > \log\log N$ is $O(1)$. Thus the total cost incurred by splitting nodes after insertions is $O(\log\log N)$. Deletions are processed in a symmetric way. We obtain the following result

**Theorem 3.** *There exists a linear-space data structure that supports one-dimensional color range reporting queries in $O(k + 1)$ time and updates in $O(\log^{\varepsilon} U)$ amortized time.*

In [19] we also show how the result of Theorem 3 can be extended to the external memory model.

## References

1. S. Alstrup, G. S. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proc. 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 476–482, 2001.
2. L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 346–357, 1999.
3. L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

4. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.
5. T. M. Chan, S. Durocher, M. Skala, and B. T. Wilkinson. Linear-space data structures for range minority query in arrays. In *13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, pages 295–306, 2012.
6. B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.
7. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.
8. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing (STOC 1984)*, pages 135–143, 1984.
9. P. Gupta, R. Janardan, and M. H. M. Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *Journal of Algorithms*, 19(2):282–317, 1995.
10. R. Janardan and M. A. Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry and Applications*, 3(1):39–69, 1993.
11. K. G. Larsen and R. Pagh. I/O-efficient data structures for colored range and prefix reporting. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 583–592, 2012.
12. K. G. Larsen and F. van Walderveen. Near-optimal range reporting structures for categorical data. In *Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page to appear, 2013.
13. E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.
14. P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *J. Comput. Syst. Sci.*, 57(1):37–49, 1998.
15. C. W. Mortensen. Generalized static orthogonal range searching in less space. Technical report, IT University Technical Report Series 2003-33, 2003.
16. C. W. Mortensen, R. Pagh, and M. Patrascu. On dynamic range reporting in one dimension. In *Proc. 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 104–111, 2005.
17. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002.
18. Y. Nekrich. Space-efficient range reporting for categorical data. In *Proc. 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 113–120, 2012.
19. Y. Nekrich and J. S. Vitter. Optimal color range reporting in one dimension. *CoRR*, abs/1306.5029, 2013.
20. Q. Shi and J. JáJá. Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. *Inf. Process. Lett.*, 95(3):382–388, 2005.
21. M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.
22. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Sys. Theory*, 10:99–127, 1977.
23. D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.