Space-Efficient Dynamic Orthogonal Point Location, Segment Intersection, and Range Reporting

Guy E. Blelloch*

Abstract

We describe an asymptotically optimal data-structure for dynamic point location for horizontal segments. For n line-segments, queries take $O(\log n)$ time, updates take $O(\log n)$ amortized time and the data structure uses $O(n)$ space. This is the first structure for the problem that is optimal in space and time (modulo the possibility of removing amortization). We also describe dynamic data structures for orthogonal range reporting and orthogonal intersection reporting. In both data structures for n points (segments) updates take $O(\log n)$ amortized time, queries take $O(\log n + k \log n / \log\log n)$ time, and the structures use $O(n)$ space, where k is the size of the output. The model of computation is the unit cost RAM.

1 Introduction

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of horizontal segments in $\Re^2$, where each $s_i = (x_i, x'_i, y_i), x_i < x'_i$. We consider two well-studied problems that maintain a data structure for S while supporting queries: *horizontal point location* supports queries on a point $p = (x_p, y_p)$ returning the segment $\max_y\{(x, x', y) \in S | x \leq x_p \leq x', y \leq y_p\}$, and *orthogonal segment intersection* supports queries on a vertical segment $s = (x_s, y_s, y'_s)$ returning the segments $\{(x, x', y) \in S | x \leq x_s \leq x', y_s \leq y \leq y'_s\}$. The dynamic versions allow for the insertion and deletion of segments in S. Tables 1 and 2 summarize previous bounds along with our bounds on these problems. Our results are the first which are asymptotically optimal (modulo the possibility of removing amortization) for horizontal point location. For orthogonal segment intersection they are the first results that use $O(n)$ space and polylogarithmic query and update times. The horizontal point location problems has applications to retroactive data structures [6]. In particular our results allow for a fully retroactive ordered dictionary with $O(\log n)$ amortized time retroactive updates and $O(\log n)$ time retroactive queries.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\Re^2$. The *orthogonal range reporting* problem is to maintain a data structure for P while supporting queries on a box $b = (x_b, x'_b, y_b, y'_b)$ returning the points $\{(x, y) \in P | x_b \leq x \leq x'_b, y_b \leq y \leq y'_b\}$. The dynamic version allows for the insertion and deletion of points in P. Table 3 summarizes bounds on the problem.

To develop our structures we introduce a reasonably general tree structure, which we refer to as compact subset trees, and use it for all the problems we consider. As is standard in many geometric structures, the tree is used to store auxiliary ordered lists within each node (*e.g.*, segment trees). Similarly to the static range-reporting structures of Chazelle [4] we save space by not storing the data within the tree but rather just supplying a way to follow a path back to the data, which is stored at the root. Within the tree we make use of locally allocated data structures in which the pointers within a structure only use $O(\log l)$ bits for a structure using l locations. By using fractional cascading [5, 10] with bridges between the lists and locally allocated data structures between the bridges in each list we need only $O(\log\log n)$ bits per list element on average.

To achieve $O(\log n)$ time we use and extend ideas from Mortensen [11, 12] to support trees with an $O(\log^\epsilon n)$ branching factor for some $0 < \epsilon < 1$ and hence a depth of $O(\log n / \log\log n)$. Mortensen described a method to assign colors to the elements of a list L and support reporting all elements between two elements of the list which are assigned any subset of colors in $O(\log\log |L| + k)$ time, for k results. We extend this to allow predecessor queries on subsets of colors.

2 Preliminaries

We assume the RAM model with words of size $w = \Theta(\log n)$ where n is the number of bits in our data structure. We sometimes refer to space in number of words and sometimes in bits, but try to be specific. We use table lookup to implement certain operations on words. In all cases the table lookup runs in $O(\alpha)$ time using static shared tables of size $O(2^{w/\alpha})$. Since

---

*Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. This work was supported in part by the National Science Foundation as part of the Aladdin Center (www.aladdin.cmu.edu) under grant CCR-0122581. This version of Jan 17, 2008 differs from the version published in SODA 2008 in that some typos have been fixed.

	Model	Space	Query	Insert	Delete
Mehlhorn and Naher [10]	pointer	$n \log n$	$\lg n \lg\lg n$	$\dagger \lg n \lg\lg n$	$\dagger \lg n \lg\lg n$
Dietz and Raman [7]	RAM	$n \log n$	$\lg n \lg\lg n$	$\lg n \lg\lg n$	$\lg n \lg\lg n$
Arge, *et. al.* [2]*	RAM	n	$\ddagger \lg n$	$\ddagger \dagger \lg n \lg^{1+\epsilon} \lg n$	$\ddagger \dagger \lg^2 n / \lg\lg n$
In this paper	RAM	n	$\lg n$	$\dagger \lg n$	$\dagger \lg n$

Table 1: Horizontal Point Location († amortized, ‡ expected, * works for non-horizontal segments). Space bounds are in words.

Segment Intersection	Model	Space	Query	Insert	Delete
Mehlhorn and Naher [10]	pointer	$n \lg n$	$\lg n \lg\lg n + k$	$\dagger \lg n \lg\lg n$	$\dagger \lg n \lg\lg n$
Dietz and Raman [7]	RAM	$n \lg n$	$\lg n \lg\lg n + k$	$\lg n \lg\lg n$	$\lg n \lg\lg n$
Mortensen [11]	RAM	$n \frac{\lg n}{\lg\lg n}$	$\lg n + k$	$\lg n$	$\lg n$
In this paper	RAM	n	$\lg n + k \frac{\lg n}{\lg\lg n}$	$\dagger \lg n$	$\dagger \lg n$

Table 2: Orthogonal Segment Intersection

Range Reporting	Model	Space	Query	Insert	Delete
Mehlhorn and Naher [10]	pointer	$n \lg n$	$\lg n \lg\lg n + k$	$\dagger \lg n \lg\lg n$	$\dagger \lg n \lg\lg n$
Dietz and Raman [7]	RAM	$n \lg n$	$\lg n \lg\lg n + k$	$\lg n \lg\lg n$	$\lg n \lg\lg n$
Nekrich [13]	RAM	$n \lg\lg n$	$\lg n + k \log\log n$	$\lg^2 n$	$\lg n \log\log n$
Mortensen [12]	RAM	$n \lg^{7/8+\epsilon} n$	$\lg n + k$	$\lg n$	$\lg n$
In this paper	RAM	n	$\lg n + k \frac{\lg n}{\lg\lg n}$	$\dagger \lg n$	$\dagger \lg n$

Table 3: Orthogonal Range Reporting

we only use a constant number of such operations the space for the tables is $O(n)$ bits for some constant α.

For a set of points S we use the notation $\max_x(S)$ and $\max_y(S)$ to indicate the point from S with the maximum x coordinate or y coordinate respectively. We assume that $\max \emptyset = \bot$ and $\min \emptyset = \top$. For a node u in a rooted tree T we use $P(u)$ to indicate the path from the root to the node u, and use $p(u)$ to indicate the parent of u. We say a *tree is ordered* if the children of every node are totally ordered. For two lists L_1 and L_2 we use $L_1 \subset L_2$ to indicate that L_1 is a subsequence of L_2 (*i.e.*, L_1 is any subset of the elements of L_2 maintaining the same order.)

We use various data structures that maintain *colors* on the elements. Consider a set of colors $\mathcal{C}$, and the set of integers $I_m = [0 \ldots m-1]$. The *colored predecessor problem* maintains a set $S \subset I_m \times \mathcal{C}$ while supporting the following operations, where $C \subset \mathcal{C}$:

`insert`(S, i, c): inserts (i, c) into S,

`delete`(S, i, c): delete (i, c) from S,

`pred`(S, i, C) $= \max\{i' : (i', c) \in L | c \in C, i' < i\}$,

`report`(S, i, i'', C) $= \{i' : (i', c) \in S | c \in C, i \leq i' \leq i''\}$.

For a set of colors $\mathcal{C}$ the *colored list problem* maintains a list of elements L such that each $e \in L$ is assigned a set of colors $C_e \subset \mathcal{C}$, while supporting the following operations:

`insert`(L, e): for $e \in L$ create a new element e' in L immediately following e and return e'

`delete`(L, e): for $e \in L$ with $C_e = \emptyset$, delete e from L,

`color`(L, e, c): for $e \in L$ add color $c \in \mathcal{C}$ to C_e,

`uncolor`(L, e, c): for $e \in L$ remove color c from C_e,

`compare`(L, e, e'): compares the locations of e and e' in L returning $<$, $=$ or $>$,

`pred`(L, e, C): for $e \in L$, $C \subset \mathcal{C}$,
return $\max\{e' \in L | C_{e'} \cap C \neq \emptyset, e' < e\}$,

`report`(L, e, e'', C): for e and e'' in L, $C \subset \mathcal{C}$,
return $\{e' \in L | C_{e'} \cap C \neq \emptyset, e < e' < e''\}$.

In this paper for both colored predecessors and lists we assume $|\mathcal{C}| = O(w)$. We can therefore represent subsets of colors as a bit vector in a constant number of words. For an element e in a list L we define its *weight* $||e||$ as $|C_e|$, and define the weight of a list $||L||$ as $\sum_{e \in L}(1 + ||e||)$.

We use two additional operations which can be implemented with the operations above but for which we need more efficient versions. The `reportAll`(L, C) query reports all elements in L with colors in C. The `filter`$(\mathcal{L}, f, g, l)$ operation takes a sequence of colored lists $\mathcal{L}$, a constant-time predicate function f on elements of that list, a constant-time color map function g and a size parameter l. It copies all elements from $\mathcal{L}$ that pass the predicate f into a new sequence of lists each of

size l (except perhaps the last). The order of elements (within each colored list and across lists) is preserved and the color map function g maps the current colors of each element to a possibly new set of colors.

We say that a data structure is *locally allocated* if all of its memory is allocated from within a contiguous region of memory. In such a structure local pointers within the structure need only be big enough to point to any location within the region allowing us to pack multiple pointers within a word. For a structure that uses a polylogarithmic number of locations, for example, such local pointers only require $O(\log \log n)$ bits. We refer to the local pointers within a structure as *short pointers* and pointers to the outside as *external pointers* and assume external pointers require $O(\log n)$ bits. We assume each region contains a number of bits that is within a constant factor of the number of bits used by the structure. Since the number of bits required by a dynamic structure can grow and shrink, the region might need to be resized, which could require moving the contents of the region to a new location. If a structure touches a constant fraction of the locations it allocates, the cost of these resizings can be amortized against the operations on the structure between resizings.

3 The Compact Sublist Tree Problem

A *sublist tree* $\mathcal{T}$ is a pair $(T, \mathcal{L})$ where T is a rooted tree with node set U, and $\mathcal{L} = \{L_u, u \in U\}$ is a collection of lists with the condition that for all nodes except the root, $L_u \subset L_{p(u)}$. We refer to the list associated with the root r of T as the *root list* L_r, and refer to $|L_r|$ as the *root size*. The root list defines a total order on the elements. We assume $|U| \in O(|L_r|)$. Although not as general as fractional cascading [5], sublist trees are sufficient for the problems we consider.

To achieve the desired space bounds we consider a compact version of sublist trees that only allows data to be associated with elements in the root list. For elements in other lists one must return to the root to retrieve data. To traverse the tree we assume an algorithm can hold a *handle* to an element within any list L_u, but only handles in the root are persistent across updates—*i.e.*, an update invalidates all handles except for those to elements in L_r and any returned by the update. The *compact sublist tree problem* is to support a sublist tree with the following operations.

`up`$(\mathcal{T}, h)$: for handle h in L_u return a handled to the same element in $L_{p(u)}$.

`down`$(\mathcal{T}, h, c)$: for h to element e in L_u and a child c of u, return a handle to e's predecessor in L_c.

`insertRoot`$(\mathcal{T}, h)$: for h in L_r insert a new element immediately following h and return its handle.

`copy`$(\mathcal{T}, h, c)$: for h to element e in L_u and for child c of u, copy e into L_c returning the new handle.

`delete`$(\mathcal{T}, h)$: for h to element e in L_u, delete e from L_u returning the handle to `up`$(\mathcal{T}, h)$.

We also support the colored list operations `color`, `uncolor`, `compare`, `pred`, and `report` on each list L_u (see Section 2). In addition to "user" assigned colors we assign each child c of a node u a color o_c and color elements in L_u with o_c iff they appear in L_c. We define the *weight* of a sublist tree $(T, \mathcal{L})$ as $\sum_{L \in \mathcal{L}} ||L||$.

Theorem 3.1. *The sublist tree problem with root size n, weight m, maximum degree $d = O(\log^{\epsilon} n)$, for some $0 < \epsilon < 1$, and with $O(d)$ colors can be solved such that* `up` *and* `compare` *take $O(1)$ time,* `down` *and* `pred` *take $O(\log \log n)$ time,* `insertRoot`*,* `copy`*,* `delete`*,* `color`*, and* `uncolor` *take $O(\log \log n)$ amortized time, and* `report` *takes $O(\log \log n + k)$ time where k is the size of the result. The space is bounded by $O(n \log n + m \log \log n)$ bits.*

The remainder of this section is the proof.

As is standard in fractional cascading [5, 10], we maintain cross linkings between each list L_u and its neighboring lists. In particular, we maintain a partitioning of each list into blocks, place a *bridge* preceding each block, and associate each bridge with the block that follows it. For every bridge in a node u we include a *bridge copy* in every neighbor of u (parent and children). We form a list A_u from the union of L_u and bridge copies placed in the appropriate order with respect to the positions of their bridges[1]. We refer to elements from L_u as the *proper elements* of A_u. For each child c of a node u the data structure assigns an internal color r_c to all bridge copies in A_u for which the corresponding bridge is in c. We maintain the A_u so they are partitioned into blocks of weight $\Theta(\log^3 n)$ (bridge copies have constant weight), except perhaps the last block.

The *child blocks* of an element $e \in L_u$ are the blocks in which e appears in the children of u. The *parent block* is the block in which e appears in its parent. We will refer to a pointer to a bridge as a *bridge pointer*. All cross pointers between neighboring lists in the sublist tree consist of a combination of bridge pointers and short pointers within a block. For the root list we maintain an auxiliary doubly linked list for all elements to store any associated data.

[1] For a bridge in u there can be a choice of where to put the bridge copy in $p(u)$. We assume any choice is valid.

Blocks. We use locally allocated structures for the blocks. For each block b we keep its elements properly ordered in a doubly linked list and store the elements with their colors in an instance of the colored list data structure. Using the results of Lemma 5.4 and the fact that blocks have weight $W = O(\log^3 n)$, `insert`, `delete`, `color`, `uncolor` and `pred` take $O(\log\log n)$ time, `report` takes $O(\log\log n + k)$ time, `compare` takes $O(1)$ time, and `reportAll` takes $O(k)$ time. The space for the structure is $O(W \log\log n)$ bits. All short pointers within a block only require $O(\log\log n)$ bits.

For each element in b we also maintain a *parent pointer* and *child pointers* defined as follows and illustrated in Figure 1. For a bridge copy, the parent or child pointer is a bridge pointer to the corresponding bridge (the other is empty). These are external pointers and require $O(\log n)$ bits. For a proper element e each child pointer consists of a short-pointer to the location of e within a child block of e. The parent pointer consist of a short-pointer to the location of e within its parent block, and an additional short pointer to the *previous parent-bridge copy* in b, if any. Finally for each block b we keep a *head bridge pointer* to the parent block of the first element in b. This is an external pointer.

Bridge structure and space bounds. The bridges for each list L_u are placed into a colored list data structure. Each bridge is colored with all the colors that appear in its block. We call this the *bridge structure*. Using the results of Lemma 5.3 we have that `compare` takes $O(1)$ time, `color`, `uncolor`, and `pred` take $O(\log\log n)$ time, `report` takes $O(\log\log n + k)$ time, and `insert` and `delete` take $O(\log^{2+\epsilon} n \log\log n)$ time. The structure uses $O(|L_u| \log^{\epsilon-2} n)$ bits and is therefore a low-order term relative to the desired bounds. We also maintain for each bridge a pointer to each of its bridge copies. Each such pointer consists of a bridge-pointer to the block in which the copy appears and a short-pointer to the location of the copy within that block.

To analyze the total space taken by the structures we first consider the space for all bridge pointers. A bridge in a node u with d children will have $d + 1$ bridge copies each contributing $O(\log n)$-bits (a bridge pointer in each direction). Every block also has a head bridge pointer, which can be charged to the previous bridge. Consider all the bridges in each node except for the last bridge in the node. Because the block sizes are maintained with weight $\Theta(\log^3 n)$, and the degree is bounded by $O(\log^\epsilon n)$, there are at most $O(m/\log^3 n)$ such bridges each contributing $O(\log^{1+\epsilon} n)$ bits for a total $o(m)$ bits. The last bridge in each node can have a total of $O(n)$ children across all of them, and will therefore contribute another $O(n \log n)$ bits. Now consider the space taken by all local pointers and the block structures. Within a block every element e requires only $O(\log\log n)$ bits per color so the total number of bits across all elements is $O(m \log\log n)$. The tree T itself will require $O(n \log n)$ bits for parent and child pointers. This gives the desired space bounds of $O(n \log n + m \log\log n)$.

Time bounds. We now consider how to implement all the operations. We assume handles to an element e in A_u consist of a pair (b, s) where b is the block in which e belongs and s is a short pointer to the element e within b. We note that updates can change how lists are blocked and this is why handles are invalidated by updates.

The $\texttt{up}(\mathcal{T}, h)$ operation for $h = (b, s)$ can be implemented using the parent pointer of s, which consists of the previous parent-bridge copy, used to find the parent block b' of s, and the desired offset s' within that block. The handle (b', s') is returned. If there is no previous parent-bridge copy we use the head bridge pointer of b. This all takes constant time.

The $\texttt{pred}(\mathcal{T}, h, C)$ operation for $h = (b, s)$ in A_u is implemented using the colored list structure within b to find the previous element s' matching a color from C. If none is found, the bridge structure for node u is used to find the previous block b' that contains one of the colors in C and then the colored list structure in b' can be used to identify the required element s'. The handle (b', s') is returned. All operations take $O(\log\log n)$ time.

The $\texttt{down}(\mathcal{T}, h, c)$ operation for h in L_u is implemented by first using $\texttt{pred}(\mathcal{T}, h, \{o_c\})$ to identify the previous proper element h' from A_u that appears in c, and $\texttt{pred}(\mathcal{T}, h', \{r_c\})$ to identify the previous bridge copy h'' from A_u for a bridge that appears in c. For a child pointer b' of h'', and child pointer s' or h' we return the handle (b', s').

The $\texttt{compare}(\mathcal{T}, h, h')$ operation can be implemented using the colored list structure within a block when elements are in the same block, or the bridge structure when not. This takes constant time.

The $\texttt{report}(\mathcal{T}, h, h', C)$ operation with $h = (b, s)$ and $h' = (b', s')$ is implemented with a report query within b if $b = b'$ and otherwise using the bridge structure to report all the blocks between b and b' that contain a color in C. For each of these blocks we use the `reportAll` operation to report the desired elements. We use `report` in b and b' to report elements in those blocks starting at h for b and finishing at h' for b'. The overall time is $O(\log\log n + k)$ time where k is the number of elements reported.

The $\texttt{copy}(\mathcal{T}, h, c)$ operation for h in L_u is implemented by first using $\texttt{down}(\mathcal{T}, h, c)$ to find the handle $h' = (b', s')$ after which to put the copy. The copy of

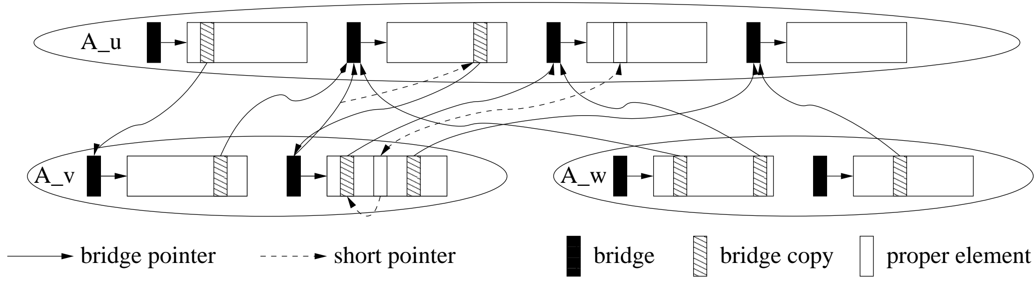

Figure 1: Pointer structure for a list A_u and two child lists A_v and A_w. Only some of the pointers are shown.

h then needs to be inserted immediately following s' in b'. Insertion in the doubly linked lists is straightforward. Insertion into the colored lists is described in Lemma 5.4. We also need to execute $\texttt{color}(\mathcal{T}, h, \{o_c\})$ to mark the fact that h now appears in c. Either the `color` operation or the insertion of the element in b' can cause a block to overflow. An overflow creates a `splitBlock` operation which is described below.

The $\texttt{color}(\mathcal{T}, h, c)$ operation for $h = (b, s)$ in A_u is implemented by coloring s in b and if it is the only element in the block of the given color, then the color is also added to b in the bridge structure. This takes $O(\log\log n)$ time. If the weight of the block becomes too large because of the extra color then the block will overflow invoking a `splitBlock` operation.

A `splitBlock` takes a block and breaks it into two blocks of approximately equal weight. When applied to a block b in a list A_u it consist of (1) finding an element of b that partitions b approximately in half based on weight, (2) creating a bridge between the two parts, (3) copying the elements from b into two new blocks b_1 and b_2 on either side of the bridge, (4) placing a copy of the new bridge in each neighbor, and (5) updating the neighboring copies of elements in b to point to their locations within b_1 or b_2. Copying the elements into b_1 and b_2 takes $O(|b| \log\log n)$ time by reinserting in the colored list data structure.

For step (4) let h be the handle of the element just before the split point found in step 1 and r be the new bridge generated in step 2. The bridge copies of r can be inserted into each child c of u by finding $h' = \texttt{down}(\mathcal{T}, h, c)$ and inserting the copy immediately following h'. Assuming u is child c' of $p(u)$, the bridge copy of r can be inserted into the parent by finding $h' = \texttt{up}(\mathcal{T}, h)$ and inserting the copy following h' and then adding to the copy color $r_{c'}$. This takes $O(\log^\epsilon n \log\log n)$ time. We also need to insert the new bridge into the bridge structure. This all takes $O(\log^3 n)$ time.

For step (5) we need to update parent pointers of child elements and child pointers of parent elements of b. For a child element s in a block b' we have to update both the short pointer to the corresponding element in b and perhaps also the pointer to the previous parent bridge within b'. For a parent element we need only update its child pointer. Since a handle for each required element and bridge copy can be found in $O(\log\log n)$ time this takes $O(|b| \log\log n)$ time. The head bridge pointers to b also need to be updated. Note that each blocks with such a head bridge copy must contain a copy of an element from b. We can therefore charge these updates against the elements of b.

Since the blocks are kept at weight $\Theta(\log^3 n)$, the overall time for a `splitBlock` is $O(\log^3 n \log\log n)$ worst case time. This can be amortized against the previous $\Theta(\log^3 n)$ operations that were applied in order to cause the split. The amortized cost of each operation is therefore $O(\log\log n)$. The `delete` and `uncolor` operations can be implemented symmetrically to `copy` and `color`. Merging of blocks might be required and this is similar to `splitBlock`.

4 Balanced Sublist Trees

So far we have assumed the tree structure T of a sublist tree $\mathcal{T} = (T, \mathcal{L})$ is static. Since it is hard to bound update times on the tree without putting some limits on the way lists can be colored, here we specialize sublist trees in a way that is sufficient for our applications.

A *doubly ordered set* E_{xy} is a set of elements E with two independent total orderings $<_x$ and $<_y$ on the elements. An *ordered sublist tree* is a sublist tree $(T, \mathcal{L})$ in which T is ordered. We say that an ordered sublist tree $(T, \mathcal{L})$ *supports* a doubly ordered set E_{xy} if (1) each leaf of T contains a unique element of E_{xy}, (2) the elements at leafs are ordered in T based on $<_x$, and (3) for every internal node u, L_u contains the union of its descendants ordered by $<_y$.

A *coloring* of a sublist tree $(T, \mathcal{L})$ is an assignment of colors to all elements in all $L_u \in \mathcal{L}$. A *coloring scheme* for a class of doubly ordered sets $\mathcal{E}_{xy}$ is a mapping

from every set $E_{xy} \in \mathcal{E}_{xy}$ and every sublist tree that supports E_{xy} to a coloring of sublist tree. We place some restrictions on the coloring scheme so that we can bound the cost of updates to the tree. An *efficient coloring scheme* is one in which (1) for any tree with branching factor $O(\log^{\epsilon} n), 0 < \epsilon < 1$ and depth d every element has $O(d)$ colors total across all nodes, (2) insertion of an element at leaf x only affects $O(d)$ colors and only on the path $P(x)$, (3) when a node $u \in T$ is split into u_1 and u_2 partitioning the children of u, at most $O(|L_u|)$ colors are affected in neighboring nodes, and (4) all color changes due to a split are local—they depend only on the existing colors of an element, the colors of its copy in the split parent or child, and the side to which an element split. Note that an efficient coloring scheme can change more than $O(|L_u|)$ colors on the elements in L_u during a split.

Since color updates are local we assume that the user supplies a color function g that maps for each element in L_u a set of of existing local colors and the side of the split to a set of new colors for the element. Similar functions g_p and g_c are given for recoloring the child and parent neighbor copies of elements in L_u. Since the number of "local" colors is small, these functions can always be implemented in constant time using table lookup.

THEOREM 4.1. *An ordered sublist tree* $(T, \mathcal{L})$ *that supports a class of doubly ordered sets* $\mathcal{E}_{xy}$ *with an efficient coloring scheme can be maintained in* $O(n)$ *words* ($O(n \log n)$ *bits) such that* T *has depth* $O(\log n / \log \log n)$ *and new elements can be inserted or deleted from* E_{xy} *(and hence* T*) in* $O(\log n)$ *amortized time.*

Proof. (sketch). We maintain the tree T as weight balanced B tree (WBB trees) [3, 12]. Let $l(u)$ be the number of leaf descendants of a node u. A WBB tree has a branching parameter $\rho > 6$ such that all leafs are at the same level and for all non-root internal nodes at height h, $\rho^h < |l(u)| < 4\rho^h$, and for the root $|l(u)| < 4\rho^h$, $\rho \geq 2$. This implies all internal nodes have degree less than 4ρ. We use $\rho = \log^{\epsilon} n, 0 < \epsilon < 1$ giving a tree of depth $d \in O(\log n / \log \log n)$. We only consider insertions—deletions can be handled using lazy deletion [14].

In a WBB tree a node u is split into two nodes u_1 and u_2 when its weight becomes too large. The children are divided, selecting the divide point to best balance $l(u_1)$ and $l(u_2)$. The nodes u_1 and u_2 are added as children to the parent, or a new root is generated if the root is split. As long as the time for splitting a node u is bounded by $O(|L_u| \log \log n)$, we can amortize the split against the $\Omega(|L_u|)$ insertions into L_u since the last split. We note that taking $\Omega(||L_u||)$ time will not suffice.

The `split` operation can be implemented by (1) copying all elements from A_u into two new lists A_{u_1} and A_{u_2}, (2) creating new bridges for these lists and inserting copies of these bridges into the neighboring lists, (3) deleting the bridge copies in neighbors for bridges in A_u, (4) updating all child and parent pointers for elements in neighboring lists to point to their copies in A_{u_1} or A_{u_2}, and (5) performing any color updates.

The copy into A_{u_1} and A_{u_2} can be done in $O(|L_u|)$ time by using `filter` (Lemma 5.4). A new bridge is then created for each block. Note that bridge copies of bridges in $p(u)$ need to be copied into both lists, but all other elements from A_u only go into one or the other list. Updating neighboring child and parent pointers can be done in $O(|L_u| \log \log n)$ time. For step (5) first consider updating colors on neighbors of L_u. We note that only $O(|L_u|)$ colors need be updated by the definition of an efficient coloring scheme. Each color change can be determined from g_c or g_p applied to one of the neighboring elements and executing the color change takes $O(\log \log n)$ amortized time. Updating the neighbor colors therefore takes $O(|L_u| \log \log n)$ amortized time. The color changes on the list L_u can be performed by the `filter` used for copying into u_1 and u_2 using the color change function g.

Insertion takes $O(\log n)$ amortized time since we add at most $O(\log n / \log \log n)$ elements and colors across all levels and each takes $O(\log \log n)$ amortized time. Since there are n elements in the tree and with an efficient coloring scheme each has total weight $O(\log n / \log \log n)$, the weight of the tree is $O(n \log n / \log \log n)$ and by Theorem 3.1 the sublist tree uses $O(n \log n)$ bits.

5 Colored Predecessors and Lists

In this section we show various results on data structures for the colored predecessor and list problems.

LEMMA 5.1. *The colored predecessor problem on* $I_m, m \leq 2^w$ *with* $O(\sqrt{w})$ *elements and* $O(\sqrt{w})$ *colors can be supported such that all operations take* $O(\alpha)$ *time, using static shared tables of total size* $O(2^{w/\alpha})$.

Proof. We first describe the $\texttt{pred}(S, i)$ operation that returns the predecessor of an integer i in a set of integers S—*i.e.*, does not use colors. This follows directly from results of Ajtai *et. al.* [1] and Fredman and Willard [9].

We view integer keys as bit strings of length $l \leq w$, and bit locations go from 0 (the least significant bit) to $l - 1$ (the most significant bit). In some places we use three values for bits $\{0, 1, \emptyset\}$ where $\emptyset$ indicates we don't care what the value of that bit is. We use table lookup to implement certain operations on bit-strings

including `msb`(s), which returns the most-significant-bit of s, and `pack`(s, b) which packs the bits in s where the corresponding bit in b is a 1 into a contiguous locations in a new word. All operations take $O(\alpha)$ time with a static shared table of size $O(l^{1/\alpha})$.

Consider a set of l-bit keys $X = x_0, x_1, ..., x_{k-1}$ and a trie built on these keys. Let $B = b_0, b_1, \ldots, b_{r-1}$ be the bit locations ($0 \leq b_i < b_{i+1} < l$) in which there is a branch in the trie. Note that $r < k$. Let the *full sketch* $s_X(y)$ of a key y be the subsequence of bits of y taken from the locations B, and the *partial sketch* $p_X(x), x \in X$ be the same subsequence but wherever a bit position is not at a branch point of the path to x_i include a $\emptyset$ bit. For $k = O(\sqrt{w})$ we can store all the partial sketches $P(X) = p_X(x_0), \ldots, p_X(x_{k-1})$ in a constant number of words. We use $P_0(X)$ to indicate the set of partial sketches of X in which all $\emptyset$ bits are replaced with zeros.

To implement `pred`(X, y) we find the predecessor x_p and successor x_s of $s_X(y)$ in $P_0(X)$. We do this using word parallelism and table lookup in constant time. We note that although the partial sketches properly order the elements of X, they do not necessarily properly locate y in X. To properly place y we identify the `msb` in which x_p and x_s differ from y and pick the lesser of these two b_d. This represents the position at which y diverges (branches) from the existing tree, which is not necessarily in B. We refer to the existing branch from which it diverges as D and it is the keys in this branch for which y might not be properly ordered. We propagate the value of the bit stored at b_d in y ($y[b_d]$) to the lower order bits of $s_X(y)$ to give s', and locate s' in $P_0(X)$. This location will properly locate y with respect to the full keys since it locates y with respect to all the keys in D and y was already located with respect to other keys.

To implement insertion and deletions of an element y we find y in X and then update the partial sketches in $P(X)$. For insertion this involves possibly adding a bit to B (at location b_d), making space in each sketch for the bit, and setting the bit appropriately to 0, 1 or $\emptyset$. Note that all keys in D will have their bits set to 0 at b_d or all to 1 (they branch from y at this position). All other keys will be set to $\emptyset$ at b_d. Therefore only a constant number of keys have to be looked at in their entirety during the search, so all operations can be done on a constant number of words in constant time using bit-parallelism and table lookup. Deletion can be done similarly.

To handle colors (`pred`(S, i, C)) we store a bit vector of $O(\sqrt{w})$ colors with each partial sketch. All such vectors fit within a constant number of words. When finding the predecessor x_p and successor x_s of $s_X(y)$ or s' in $P_0(X)$ we can ignore all sketches that do not match a color in C. Such matching can be determined with word parallelism in constant time. This will return the predecessor just among the keys with a color in C. Adding and deleting colors just updates the color bit vectors. `report`$(S, y, y', C,)$ can be implemented by finding the successor of y, the predecessor of y' and using bit-parallelism to find the elements in B that fall between the two and match a color in C.

We now present an extension of the results by Mortensen [12, Theorem 1] to allow for predecessor queries (Mortensen just describes report queries).

LEMMA 5.2. *The colored predecessor problem on $I_m, m \leq 2^w$ with $c = O(\sqrt{w})$ colors can be supported such that* `insert`, `delete`, *and* `pred` *take $O(\log\log m)$ time,* `report` *queries take $O(\log\log m + k)$ time, and the space required is $O(cm \log m)$ bits, in addition to a $O(2^{w/\alpha})$-bit static shared table.*

Proof. The solution is based on the van Emde Boas *et. al.* data structure (EKZ) [15]. Mortensen [12] described `report` queries so we just describe `pred` queries.

We first review the EKZ structure. The structure is defined recursively. We use V_m to indicate a structure that can store a set $S \subset \{1, \ldots, m\}$. It consists of `min` and `max` slots, a *top* structure $T : V_{\sqrt{m}}$, and an array A of *bottom* structures $A_i : V_{\sqrt{m}}, 0 \leq i < \sqrt{m}$. The `min` and `max` slots are used to store the minimum and maximum elements in S, and the rest of the elements are stored in one of the bottom structures. In particular a key i is stored in $A_{i/\sqrt{m}}$ using key ($i \bmod \sqrt{m}$). The top structure stores a key j if there are any elements in A_j. For report queries the EKZ structure also stores all elements ordered in a linked list.

To insert a new element i, and assuming there are already two distinct elements in the structure occupying `min` and `max`, we check which is the median of `min`, `max` and i, call it j. We then insert $j \bmod \sqrt{m}$ into $A_{j/\sqrt{m}}$. If $A_{j/\sqrt{m}}$ is empty, then the insert returns immediately after installing $j \bmod \sqrt{m}$ as the `min` and `max` of $A_{j/\sqrt{m}}$. In this case we insert the key $j/\sqrt{m}$ into the top structure to identify that it is no longer empty. Note that at most one recursive call is made to insert, either to the bottom structure if not empty, or to the top structure if empty. The time for insertion is therefore bounded by $T(m) = T(\sqrt{m}) + O(1) = O(\log\log m)$. The delete operation is symmetrical.

To determine `pred`(V_m, i), if i is less than `min` we return $\perp$ indicating no predecessor is found, and otherwise we recursively call `pred`$(A_{i/\sqrt{m}}, i \bmod \sqrt{m})$. If this returns $\perp$ (in constant time), then let j =

$\texttt{pred}(T, i/\sqrt{m})$. If $j = \bot$ then return min from V_m else return max from the structure A_j. Note that in all cases only one non-trivial recursive call is made so again the runtime is $O(\log \log m)$.

We now consider the multicolor version. Instead of storing min and max we store arrays Min and Max, each containing c elements corresponding to the min and max of each color. Since we have $O(\sqrt{w})$ colors we can use Lemma 5.1 to store Min and Max so that we can find the predecessor (or successor) for any set of colors $C \subset \mathcal{C}$ in each array in constant time. To distinguish the predecessor on Min and Max we refer to it as predS. The $\texttt{pred}(V_m, i, C)$ operation is a version of the uncolored predecessor routine in which each lookup in Min or Max is done in "parallel" using predS. It proceeds as follows:

```
proc pred(V, i, C)
  m = |V|
  a = predS(V.Min, i, C)
   if a = ⊥ then return ⊥
   else
     b = pred(V.A_{i/√m}, i mod √m, C)
     if b ≠ ⊥ then return max(a, b)
     else
       c = pred(V.T, i/√m, C)
       if c = ⊥ then return a
       else return max(a, predS(V.Max, ⊤, C))
```

As with the uncolored version this takes $O(\log \log m)$ time since if $b = \bot$ then the recursive call on $V.A_{i/\sqrt{m}}$ took constant time, and otherwise no recursive call is made on $V.T$. Insertion and deletion for a given color c proceed as in the uncolored case treating the color independently from the others.

LEMMA 5.3. *The colored list problem with $O(n)$ elements and $c = O(w^\epsilon)$ colors, for some $0 < \epsilon < 1$, can be supported such that* compare *takes $O(1)$ time,* color*,* uncolor*, and* pred *take $O(\log \log n)$ time,* report *takes $O(\log \log n + k)$ time, and* insert *and* delete *take $O(c \log^2 n \log \log n)$ time, with $O(cn \log n)$ bits.*

Proof. We use the list numbering scheme of Willard [16], which maintains a list with insertions and deletions at any position such that all elements are assigned an integer in the range $[1, O(n)]$ in order of the list. Every update modifies at most $O(\log^2 n)$ elements and takes $O(\log^2 n)$ time. We use the integers assigned by this algorithm in the color predecessor results of Lemma 5.2. Each insertion or deletion therefore requires $O(\log^2 n)$ updates each taking $O(\log \log n)$ time per color for a total of $O(c \log^2 n \log \log n)$ time. The color and uncolor do not need to update the list, but just add/remove a color. The compare operation uses the integers directly.

LEMMA 5.4. *The colored list problem on list L with $n = |L| = O(w^c)$ elements and $O(w/\log w)$ colors, for any constant c can be supported with $O(||L|| \log n)$ bits such that* insert*,* delete*,* color*,* uncolor *and* pred *take $O(\log n)$ time,* compare *takes $O(1)$ time,* report *takes $O(\log n + k)$ time,* reportAll *takes $O(k)$ time, and* filter *on m total elements takes $O(m)$ time, where k is the size of the output.*

Proof. The structure is hierarchical with a tree at the top level and each leaf a *group* at the lower level. For external reference we maintain a *home cell* for each element in a fixed location, and link these in a doubly-linked list. The data for an element e, stored elsewhere, consists of a pointer back to the home cell and an array of color fields, one for each color it is assigned. For each color in the array we store the color identification, and pointers to the cell for the previous and next element with that color thus creating a linked list for each color. We store this all as an adjacent block of bits. This uses $O(||e|| \log n)$ bits which fits within a constant number of words. We partition the elements into groups of weight $\Theta(w/\log n)$ based on their ordering within the colored list (except the last group which can be smaller). We pack the data for each element in a group one after the other within $O(1)$ words. The home cell points to the group in which the element belongs.

We can use word parallelism and table lookup to execute color, uncolor, insert, delete, and pred on a group in $O(1)$ time, and report in $O(k)$ time. As usual the table lookup can be done in $O(\alpha)$ time with a static shared table of size $O(2^{w/\alpha})$. For color and uncolor we also need to splice the color into (or out of) the linked list for that color. Insertions and deletions can cause the group to overflow or underflow. The group can then be split and/or joined with the groups on either side to maintain $\Theta(w/\log n)$ elements per group. Again this can all be done in $O(1)$ worst-case time.

The groups are organized in a balanced binary tree with each group as a leaf and ordered in the tree (left to right). Each internal node maintains a bit-array with a flag for each color indicating whether that color is present in the subtree. The bit-array for a node can fit in $O(1)$ words and logical operations can be used to test if there is any element from a color subset C' in the array and hence in the subtree. Insertion or deletion into this tree is required when a group overflows or underflows and takes $O(\log n)$ time using standard balanced tree implementations.

When executing $\texttt{pred}(x, C')$ we first search in the group. If nothing is found in the group we traverse the tree from the group in which x belongs up toward the root until a left parent is found which contains one of the colors in C'. We can then traverse down

starting at the left child of that parent searching for the rightmost group that contains one of the colors. Finally we search within that group for the greatest element with a color from C'. This all takes $O(\log n)$ time. Implementing `delete`, `color`, `uncolor`, and report, are all straightforward using the bit masks. We can use an order maintenance structure [8] on the groups to support `compare` in constant time. We can use the linked lists for each color to answer a `reportAll` query.

We can charge the space for each internal node of the tree to the leaves of the tree. The space for the leaves of the tree is bounded by $O(||L|| \log n)$ bits since each element uses $O(||e|| \log n)$ bits.

The $\texttt{filter}(\mathcal{L}, f, l)$ operation works as follows. When processing a particular list $L \in \mathcal{L}$ we scan the elements using the linked list on the home cells copying the ones we keep (that pass the predicate f) into a new colored list structure L'. As we scan we maintain a *predecessor array* with an entry for every available color each containing a pointer to the home cell in L' of the previous kept element that contained that color. This array fits within one word. When we get to an element e for which the predicate f passes we extract its colors, use g to generate new colors, and find the previous kept element for each new color using the predecessor array (all using word parallelism in constant time). We can also update the predecessor array with pointers to the home cell of e. When L' is full we build the tree and do a back pass to create color links in the other direction. We then start with a new colored list structure. Building the tree takes $O(n)$ time for n total elements. Maintaining the colors in "parallel" is important in avoiding spending time that is proportional to the total weight of the elements instead of the number of elements.

6 Applications

We now describe bounds on horizontal point location, segment intersection and range reporting that are based on the sublist tree bounds described in Section 3. For simplicity we assume all points are in general position (do not share an x or y coordinate) except for those at the endpoints of a segment. In all the structures we use a *root tree*, which is a balanced search tree ordered by y coordinate containing pointers to the elements in L_r of a sublist tree.

THEOREM 6.1. *The dynamic horizontal point location problem can be solved using $O(n)$ words ($O(n \log n)$ bits) so that insertions and deletions take $O(\log n)$ amortized time and queries take $O(\log n)$ worst-case time.*

Proof. Let S be a set of n horizontal segments $s_i = (x_i, x'_i, y_i), x_i < x'_i$ and let E be the set of $2n$ endpoints of the segments, where each (x_i, y_i) is a left endpoint and (x'_i, y_i) is a right endpoint. The set E is doubly ordered based on the x and y coordinates and assuming $(x_i, y_i) <_y (x'_i, y_i)$. Let $(T, \mathcal{L})$ be an ordered sublist tree that supports E.

We use the following coloring scheme. For a node u with children $c_1, c_2, \ldots, c_d$ assign *left color* l_{c_j} to each left endpoint of a segment (x, x', y) for which $(x, y) \in L_{c_j} \wedge (x', y) \notin L_u$ (*i.e.*, the left endpoint is in child j of u and the right endpoint is not in u), assign *right color* r_{c_j} to each right endpoint of a segment (x, x', y) for which $(x', y) \in L_{c_j} \wedge (x, y) \notin L_u$, and assign a *cross color* a_{c_j} to each left endpoint of a segment (x, x', y) for which $(x, y) \in L_{c_i} \wedge (x', y) \in L_{c_k} \wedge i < j < k$ (*i.e.*, the left endpoint is in child i and the right endpoint is in child k and child j is between them). We say a segment with an endpoint in u covers a child c_i if its left endpoint is to the left of all points in c_i and its right endpoint is to the right. For each child c_i we define the *cover colors* $C_i = \{l_{c_j}, j < i\} \bigcup \{r_{c_j}, j > i\} \bigcup \{a_{c_i}\}$. A segment with an endpoint in u covers c_i iff one of its endpoints in u has a color in C_i.

Based on the coloring scheme described, a point $p = (x_p, y_p)$ can be located by a simple search from the root to a leaf and then returning to the root to identify the segment. At the root we locate the handle $h = \max_y\{(x, y) \in L_u | y \leq y_p\}$ using a binary search on the root tree. At each internal node u given a handle h we identify the child c_i in which x_p belongs, point locate p with respect to segments that cover c_i using $h' = \texttt{pred}(h, C_i)$, locate the handle h in child c_i using $\texttt{down}(\mathcal{T}, h, c_i)$, and move to child c_i. When returning to the root, if the child returns h'' the node returns $\max(h', h'')$ (using `compare`). Once we reach the root we identify the desired segment. Each level of the search takes $O(\log \log n)$ time for a total of $O(\log n)$ time.

Insertion of a new segmented is implemented by inserting both of its endpoints in the balanced sublist tree and the root tree. The coloring scheme is efficient since it assigns at most $O(\log n / \log \log n)$ colors to each element ($O(\log n / \log \log n)$ along each path to an endpoint and $O(\log^\epsilon n)$ at the least common ancestor of the paths), and it is not hard to verify that a split causes at most $O(1)$ color changes per element in neighbors (although it can cause up to $O(\log^\epsilon n)$ color deletions within the node itself), and is local. By Theorem 4.1 the space is bounded by $O(n)$ words and insertion and deletion take $O(\log n)$ amortized time.

THEOREM 6.2. *The dynamic segment intersection problem can be solved using $O(n)$ words ($O(n \log n)$ bits) so that insertions and deletions take $O(\log n)$ amortized time and intersection queries take $O(\log n + k \log n / \log \log n)$ worst-case time, where k is the size of the result.*

Proof. For segment intersection we maintain the exact same data structure as for point location, but the query is different. For a intersection query for a segment (x, y, y') we locate the successor of y and the predecessor of y' in L_r using the root tree returning two handles h_l and h_r. We pass both handles down in the search and at each node instead of doing a predecessor search $\texttt{pred}(h, C_i)$ we do a report query $\texttt{report}(h_l, h_r, C_i)$. This will return a set of handles and to extract the segments we need to return to the root for each handle using `up` operations. Therefore each reported segment will require $O(\log n/ \log\log n)$ time. Since traversing the tree takes $O(\log n)$ time the total time is $O(\log n + k \log n/ \log\log n)$.

Theorem 6.3. *The dynamic range reporting problem can be solved using $O(n)$ words so that insertions and deletions take $O(\log n)$ amortized time and range reporting queries take $O(\log n + k \log n/ \log\log n)$ worst-case time, where k is the size of the result.*

Proof. The set of points P is doubly ordered based on the x and y coordinates. Let $(T, \mathcal{L})$ be an ordered sublist tree that supports P. We need no colors beyond the colors o_{c_i} so the coloring scheme is efficient. Therefore by Theorem 4.1 the space is bounded by $O(n)$ words and insertion and deletion can be implemented in $O(\log n)$ amortized time. For a range query with a box (x, x', y, y') let P and P' be the search paths to x and x' in T, v be the node at which they separate, P_v be the part of P below v and P'_v the part of P' below v (not inclusive). Let x and y be in child c_i and child c_k of v, respectively. In v we do a report query on colors $C = \{o_{c_j}, i < j < k\}$, for each $u \in P_v$ where x is in child i we do a report query on colors $C = \{o_{c_j}, i < j\}$, and for each $u \in P'_v$ where x' is in child k we do a report query on colors $C = \{o_{c_j} | j < k\}$. As with segment intersection reporting each element requires returning to the root which takes $O(\log n/ \log\log n)$ time. Since traversing down the tree along two paths to start the report queries takes $O(\log n)$ time, the total time is $O(\log n + k \log n/ \log\log n)$.

7 Conclusion

In the paper we described various structures that achieve asymptotically optimal space bounds and optimal time bounds in the term that is not output sensitive. This was mainly achieved by using local data structures and short pointers. It remains open whether the output-sensitive cost can be reduced. For the static range-reporting problem Chazelle showed $O(n)$ space (in words) and $O(\log n + k \log^{\epsilon}(2n/k))$ time queries for a constant $\epsilon > 0$ [4]. These are tighter than our dynamic bounds in the output sensitive costs. One might also try to remove the amortization.

References

[1] M. Ajtai, M. L. Fredman, and J. Komlos. Hash functions for priority queues. *Information and Control*, 63(3):217–225, Dec. 1986.

[2] L. Arge, G. Brodal, and L. Georgiadis. Improved dynamic planar point location. In *Proc. 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2006.

[3] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 560–569, 1996.

[4] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17:427–462, 1988.

[5] B. Chazelle and L. Guibas. Fractional cascading. *Algorithmica*, 1:133–196, 1986.

[6] E. D. Demaine, J. Iacono, and S. Langerman. Retroactive data structures. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 274–283, 2004.

[7] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd Annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 78–88, 1991.

[8] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[9] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, June 1994.

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

[11] C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 618–627, 2003.

[12] C. W. Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35(6):1494–1525, 2006.

[13] Y. Nekrich. Space efficient dynamic orthogonal range reporting. In *Proc. 21st Annual Symposium on Computational geometry*, pages 306–313, 2005.

[14] M. H. Overmars. *Design of Dynamic Data Structures*. Springer-Verlag, New York, 1987.

[15] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[16] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.