NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Uniquely Represented Data Structures for Computational Geometry

Guy E. Blelloch [1] Daniel Golovin [2]
Virginia Vassilevska [3]

April 2008
CMU-CS-08-115$_2$

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present new techniques for the construction of uniquely represented data structures in a RAM, and use them to construct efficient uniquely represented data structures for orthogonal range queries, line intersection tests, point location, and 2-D dynamic convex hull. Uniquely represented data structures represent each logical state with a unique machine state. Such data structures are *strongly history-independent*. This eliminates the possibility of privacy violations caused by the leakage of information about the historical use of the data structure. Uniquely represented data structures may also simplify the debugging of complex parallel computations, by ensuring that two runs of a program that reach the same logical state reach the same physical state, even if various parallel processes executed in different orders during the two runs.

[1]Supported in part by NSF ITR grant CCR-0122581 (The Aladdin Center).
[2]Supported in part by NSF ITR grants CCR-0122581 (The Aladdin Center) and IIS-0121678.
[3]Supported in part by NSF ITR grant CCR-0122581 (The Aladdin Center) and a Computer Science Department Ph.D. Scholarship.

University Libraries
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Keywords: Unique Representation, Canonical Forms, History Independence, Oblivious Data Structures, Ordered Subsets, Range Trees, Point Location, Convex Hull

1 Introduction

Most computer applications store a significant amount of information that is hidden from the application interface—sometimes intentionally but more often not. This information might consist of data left behind in memory or disk, but can also consist of much more subtle variations in the state of a structure due to previous actions or the ordering of the actions. For example a simple and standard memory allocation scheme that allocates blocks sequentially would reveal the order in which objects were allocated, or a gap in the sequence could reveal that something was deleted even if the actual data is cleared. Such location information could not only be derived by looking at the memory, but could even be inferred by timing the interface—memory blocks in the same cache line (or disk page) have very different performance characteristics from blocks in different lines (pages). Repeated queries could be used to gather information about relative positions even if the cache is cleared ahead of time. As an example of where this could be a serious issue consider the design of a voting machine. A careless design might reveal the order of the cast votes, giving away the voters' identities.

To address the concern of releasing historical and potentially private information various notions of *history independence* have been derived along with data structures that support these notions [14, 20, 13, 7, 1]. Roughly, a data structure is history independent if someone with complete access to the memory layout of the data structure (henceforth called the "observer") can learn no more information than a legitimate user accessing the data structure via its standard interface (*e.g.*, what is visible on screen). The most stringent form of history independence, *strong history independence*, requires that the behavior of the data structure under its standard interface along with a sequence of randomly generated bits, which are revealed to the observer, uniquely determine its memory representation. We say that such structures have a *unique representation*.

The idea of unique representations has also been studied earlier [27, 28, 2] largely as a theoretical question to understand whether redundancy is required to efficiently support updates in data structures. The results were mostly negative. Anderson and Ottmann [2] showed, for example, that ordered dictionaries storing n keys require $\Theta(n^{1/3})$ time, thus separating unique representations from redundant representations (redundant representations support dictionaries in $\Theta(\log n)$ time, of course). This is the case even when the representation is unique only with respect to the pointer structure and not necessarily with respect to memory layout. The model considered, however, did not allow randomness or even the inspection of secondary labels assigned to the keys.

Recently Blelloch and Golovin [4] described a uniquely represented hash table that supports insertion, deletion and queries on a table with n items in $O(1)$ expected time per operation and using $O(n)$ space. The structure only requires $O(1)$-wise independence of the hash functions and can therefore be implemented using $O(\log n)$ random bits. The approach makes use of recent results on the independence required for linear probing [22] and is quite simple and likely practical. They also showed a perfect hashing scheme that allows for $O(1)$ worst-case queries, although it requires more random bits and is probably not practical. Using the hash tables they described efficient uniquely represented data structures for ordered dictionaries and the order maintenance problem [10]. This does not violate the Anderson and Ottmann bounds as it allows the keys to be hashed, allows random bits to be part of the input, and considers performance in expectation (or with high probability) rather than in the worst case. Very recently, Naor *et. al.* [19] developed a

second uniquely represented dynamic perfect hash table supporting deletions. Their construction is based on the *cuckoo hashing* scheme of Pagh and Rodler [23], whereas the earlier construction of [4] is based on linear probing.

In this paper we use these and other results to develop various uniquely represented structures in computational geometry. (An extended abstract version of this paper appeared in the Scandinavian Workshop on Algorithm Theory [5].) We show uniquely represented structures for the well studied dynamic versions of orthogonal range searching, horizontal point location, and orthogonal line intersection. All our bounds match the bounds achieved using fractional cascading [8], except that our bounds are in expectation instead of worst-case. In particular for all problems the structures support updates in $O(\log n \log \log n)$ expected time and queries in $O(\log n \log \log n + k)$ expected time, where n is the number of supported elements and k is the size of the output. They use $O(n \log n)$ space and use $O(1)$-wise independent hash functions. Although better redundant data structures for these problems are known [16, 17, 3] (an $O(\log \log n)$-factor improvement), our data structures are the first to be uniquely represented. Furthermore they are quite simple, arguably simpler than previous redundant structures that match our bounds.

Instead of fractional cascading our results are based on a uniquely represented data structure for the ordered subsets problem (OSP). This problem is to maintain subsets of a totally ordered set under insertions and deletions to either the set or the subsets, as well as predecessor queries on each subset. Our data structure supports updates or comparisons on the totally ordered set in expected $O(1)$ time, and updates or queries to the subsets in expected $O(\log \log m)$ time, where m is the total number of element occurrences in subsets. This structure may be of independent interest.

We also describe a uniquely represented data structure for 2-D dynamic convex hull. For n points it supports point insertions and deletions in $O(\log^2 n)$ expected time, outputs the convex hull in time linear in the size of the hull, takes expected $O(n)$ space, and uses only $O(\log n)$ random bits. Although better results for planar convex hull are known ([6]), we give the first uniquely represented data structure for this problem.

Our results are of interest for a variety of reasons. From a theoretical point of view they shed some light on whether redundancy is required to efficiently support dynamic structures in geometry. From the privacy viewpoint range searching is an important database operation for which there might be concern about revealing information about the data insertion order, or whether certain data was deleted. Unique representations also have potential applications to concurrent programming and digital signatures [4].

2 Preliminaries

Let $\mathbb{R}$ denote the real numbers, $\mathbb{Z}$ denote the integers, and $\mathbb{N}$ denote the naturals. Let $[n]$ for $n \in \mathbb{Z}$ denote $\{1, 2, \ldots, n\}$.

Unique Representation. Formally, an *abstract data type* (ADT) is a set V of logical states, a special starting state $v_0 \in V$, a set of allowable operations $\mathcal{O}$ and outputs $\mathcal{Y}$, a transition function $t : V \times \mathcal{O} \to V$, and an output function $y : V \times \mathcal{O} \to \mathcal{Y}$. The ADT is initialized to v_0,

and if operation $O \in \mathcal{O}$ is applied when the ADT is in state v, the ADT outputs $y(v, O)$ and transitions to state $t(v, O)$. A *machine model* $\mathcal{M}$ is itself an ADT, typically at a relatively low level of abstraction, endowed with a programming language. Example machine models include the *random access machine* (RAM) with a simple programming language that resembles C, the *Turing machine* (where the program corresponds to the finite state control of the machine), and various *pointer machines*. An *implementation* of an ADT $\mathcal{A}$ on a machine model $\mathcal{M}$ is a mapping f from the operations of $\mathcal{A}$ to programs over the operations of $\mathcal{M}$. Given a machine model $\mathcal{M}$, an implementation f of some ADT (V, v_0, t, y) is said be *uniquely represented* (UR) if for each $v \in V$, there is a unique machine state $\sigma(v)$ of $\mathcal{M}$ that encodes it. Thus, if we run $f(O)$ on $\mathcal{M}$ exactly when we run O on (V, v_0, t, y), then the machine is in state $\sigma(v)$ iff the ADT is in logical state v.

Model of Computation & Memory allocation. Our model of computation is a unit cost RAM with word size at least $\log |U|$, where U is the universe of objects under consideration. As in [4], we endow our machine with an infinite string of random bits. Thus, the machine representation may depend on these random bits, but our strong history independence results hold no matter what string is used. In other words, a computationally unbounded observer with access to the machine state and the random bits it uses can learn no more than if told what the current logical state is. In our performance guarantees we take probabilities and expectations over these random bits; we use randomization solely to improve performance.

Our data structures are based on the solutions of several standard problems. For some of these problems UR data structures are already known. The most basic structure that is required throughout this paper is a hash table with insert, delete and search. The most common use of hashing in this paper is for memory allocation. Traditional memory allocation depends on the history since locations are allocated based on the ordering in which they are requested. We maintain data structures as a set of *blocks*. Each block has its own unique integer label which is used to hash the block into a unique *memory cell*. It is not too hard to construct such block labels if the data structures and the basic elements stored therein have them. For example, we can label points in $\mathbb{R}^d$ using their coordinates and if a point p appears in multiple structures, we can label each copy using a combination of p's label, and the label of the data structure containing that copy. Such a representation for memory contains no traditional "pointers" but instead uses labels as pointers. For example for a tree node with label l_p, and two children with labels l_1 and l_2, we store a cell containing (l_1, l_2) at label l_p. This also allows us to focus on the construction of data structures whose *pointer structure* is UR; such structures together with this memory allocation scheme yield UR data structures in a RAM. Note that all of the tree structures we use have pointer structures that are UR. Given the memory allocation scheme above, the proofs that our data structures are UR are thus quite straightforward, and we omit them.

Trees. Throughout this paper we make significant use of tree-based data structures. We note that none of the deterministic trees (e.g., red-black, AVL, splay-trees, weight-balanced trees) have unique representations, even not accounting for memory layout. We therefore use randomized treaps [25] throughout our presentation.

Definition 2.1 (Treap) *A* treap *is a binary search tree in which each node has both a* key *and a* priority. *The nodes appear in-order by their keys (as in a standard binary search tree) and are heap-ordered by their priorities, so that the each parent has a higher priority than its children.*

We expect that one could also make use of skip lists [24] but we can leverage the elegant results on treaps with respect to limited randomness. For a tree T, let $|T|$ be the number of nodes in T, and for a node $v \in T$, let T_v denote the subtree rooted at v, and let $\mathrm{depth}(x)$ denote the length of the path from x to the root of T.

Definition 2.2 (k-Wise Independence) *Let $k \in \mathbb{Z}$ and $k \geq 2$. A set of random variables is* k-wise independent *if any k-subset of them is independent. A family $\mathcal{H}$ of hash functions from set A to set B is* k-wise independent *if the random variables in $\{h(x)\}_{x \in A}$ are k-wise independent and uniform on B when h is picked at random from $\mathcal{H}$.*

Unless otherwise stated, all treaps in this paper use 8-wise independent hash functions to generate priorities. We use the following properties of treaps.

Theorem 2.3 (Selected Treap Properties [25]) *Let T be a random treap on n nodes with priorities generated by an 8-wise independent hash function from nodes to $[p]$, where $p \geq n^3$. Then for any $x \in T$,*

(1) $\mathbf{E}[\mathrm{depth}(x)] \leq 2\ln(n) + 1$, *so access and update times are expected $O(\log n)$*

(2) Given a predecessor handle, the expected insertion or deletion time is $O(1)$

(3) If the time to rotate a subtree of size k is $f(k)$ for some $f : \mathbb{N} \to \mathbb{R}_{\geq 1}$ such that $f(k)$ is polynomially bounded, then the total time due to rotations to insert or delete an element is $O\left(\frac{f(n)}{n} + \sum_{0<k<n} \frac{f(k)}{k^2}\right)$ in expectation. Thus even if the cost to rotate a subtree is linear in its size (e.g., $f(k) = \Theta(k)$), updates take expected $O(\log n)$ time.

Dynamic Ordered Dictionaries. The dynamic ordered dictionary problem is to maintain a set $S \subset U$ for a totally ordered universe $(U, <)$. In this paper we consider supporting insertion, deletion, predecessor ($\mathrm{Pred}(x, S) = \max\{e \in S : e < x\}$) and successor ($\mathrm{Succ}(x, S) = \min\{e \in S : e > x\}$). Henceforth we will often skip successor since it is a simple modification to predecessor. If the keys come from the universe of integers $U = [m]$ a simple variant of the Van Emde Boas *et. al.* structure [29] is UR and supports all operations in $O(\log \log m)$ expected time [4] and $O(|S|)$ space. Under the comparison model we can use treaps to support all operations in $O(\log |S|)$ time and expected $O(|S|)$ space. In both cases $O(1)$-wise independence of the hash functions is sufficient. We sometimes associate data with each element.

Order Maintenance. The *Order-Maintenance* problem [10] (OMP) is to maintain a total ordering L on n elements while supporting the following operations:

- Insert(x, y): insert new element y right after x in L.
- Delete(x): delete element x from L.
- Compare(x, y): determine if x precedes y in L.

In previous work [4] the first two authors described a randomized UR data structure for the problem that supports compare in $O(1)$ worst-case time and updates in $O(1)$ expected time. It is based on a three level structure. The top two levels use treaps and the bottom level uses state transitions. The bottom level contains only $O(\log\log n)$ elements per structure allowing an implementation based on table lookup. In this paper we use this order maintenance structure to support ordered subsets.

Ordered Subsets. The *Ordered-Subset* problem (OSP) is to maintain a total ordering L and a collection of subsets of L, denoted $\mathcal{S} = \{S_1, \dots, S_q\}$, while supporting the OMP operations on L and the following ordered dictionary operations on each S_k:

- $\text{Insert}(x, S_k)$: insert $x \in L$ into set S_k.
- $\text{Delete}(x, S_k)$: delete x from S_k.
- $\text{Pred}(x, S_k)$: For $x \in L$, return $\max\{e \in S_k | e < x\}$.

Dietz [11] first describes this problem in the context of fully persistent arrays, and gives a solution yielding $O(\log\log m)$ expected amortized time operations, where $m := |L| + \sum_{i=1}^{q} |S_i|$ is the total number of element occurrences in subsets. Mortensen [15] describes a solution that supports updates to the subsets in expected $O(\log\log m)$ time, and all other operations in $O(\log\log m)$ worst case time. In section 3 we describe a UR version.

3 Uniquely Represented Ordered Subsets

Here we describe a UR data structure for the ordered-subsets problem. It supports the OMP operations on L in expected $O(1)$ time and the dynamic ordered dictionary problems on the subsets in expected $O(\log\log m)$ time, where $m = |L| + \sum_{i=1}^{q} |S_i|$. We use a somewhat different approach than Mortensen [15], which relied heavily on the solution of some other problems which we do not know how to make UR. Our solution is more self-contained and is therefore of independent interest beyond the fact that it is UR. Furthermore, our results improve on Mortensen's results by supporting insertion into and deletion from L in $O(1)$ instead of $O(\log\log m)$ time.

Theorem 3.1 *Let $m := |\{(x, k) \; : \; x \in S_k\}| + |L|$. There exists a UR data structure for the ordered subsets problem that uses expected $O(m)$ space, supports all OMP operations in expected $O(1)$ time, and all other operations in expected $O(\log\log m)$ time.*

We devote the rest of this section to proving Theorem 3.1. To construct the data structure, we start with a UR *order maintenance* data structure on L, which we will denote by $L^{\leq}$ (see Section 2). Whenever we are to compare two elements, we simply use $L^{\leq}$.

We recall an approach used in constructing $L^{\leq}$ [4], ***treap partitioning***: Given a treap T and an element $x \in T$, let its *weight* $w(x, T)$ be the number of its descendants, including itself. For a parameter s, let $\mathcal{L}_s[T] = \{x \in T : w(x, T) \geq s\} \cup \{\text{root}(T)\}$ be the *weight s partition leaders* of T[1]. For every $x \in T$ let $\ell(x, T)$ be the least (deepest) ancestor of x in T that is a partition leader. Here, each node is considered an ancestor of itself. The weight s partition leaders partition the

[1]For technical reasons we include $\text{root}(T)$ in $\mathcal{L}_s[T]$ ensuring that $\mathcal{L}_s[T]$ is nonempty.

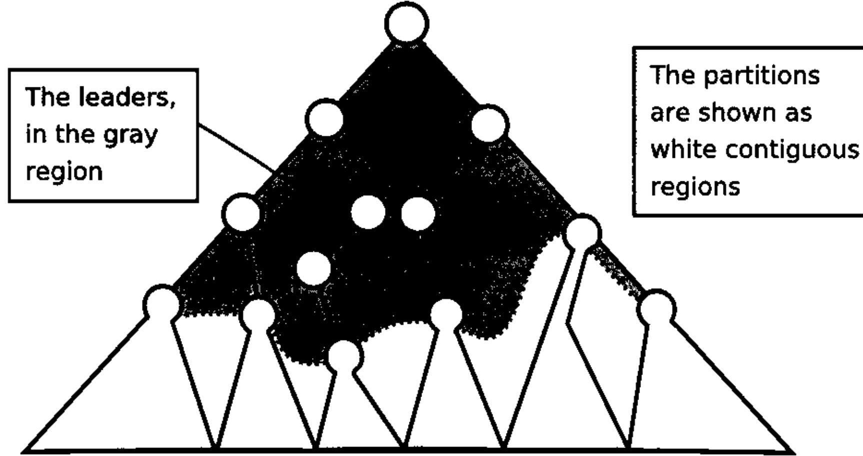

Figure 1: A depiction of the treap partitioning scheme.

treap into the sets $\{\{y \in T : \ell(y, T) = x\} : x \in \mathcal{L}_s[T]\}$, each of which is a contiguous block of keys from T. Figure 1 depicts the treap partitioning.

In the construction of $L^{\leq}$ [4] the elements of the order are treap partitioned twice, at weight $s := \Theta(\log |L|)$ and again at weight $\Theta(\log\log |L|)$. The partition sets at the finer level of granularity are then stored in UR hash tables. In the rest of the exposition we will refer to the treap on all of L as $T(L^{\leq})$. The set of weight s partition leaders of $T(L^{\leq})$ is denoted by $\mathcal{L}[T(L^{\leq})]$, and the treap on these leaders by $T(\mathcal{L}[L^{\leq}])$.

The other main structure that we use is a treap $\mathcal{T}$ containing all elements from the set $\hat{L} = \{(x, k) : x \in S_k\} \cup \{(x, 0) : x \in \mathcal{L}[T(L^{\leq})]\}$. Treap $\mathcal{T}$ is depicted in Figure 2. It is partitioned by weight $s = \Theta(\log m)$ partition leaders. Each of these leaders is labeled with the path from the root to it (0 for left, 1 for right), so that label of each v is the binary representation of the root to v path. We keep a hash table H that maps labels to nodes, so that the subtreap of $\mathcal{T}$ on $\mathcal{L}[\mathcal{T}]$ forms a trie. It is important that only the leaders are labeled since otherwise insertions and deletions would require $O(\log m)$ time. We maintain a pointer from each node of $\mathcal{T}$ to its leader. In addition, we maintain pointers from each $x \in \mathcal{L}[T(L^{\leq})]$ to $(x, 0) \in \mathcal{T}$.

We store each subset S_k in its own treap T_k, also partitioned by weight $s = \Theta(\log m)$ leaders. When searching for the predecessor in S_k of some element x, we use $\mathcal{T}$ to find the leader ℓ in T_k of the predecessor of x in S_k. Once we have ℓ, the predecessor of x can easily be found by searching in the partition of S_k associated with leader ℓ, which is either $\{\ell\}$ or is stored in an $O(\log m)$-sized subtree of T_k rooted at ℓ. The exact details appear later. To guide the search for ℓ, we store at each node v of $\mathcal{T}$ the minimum and maximum T_k-leader labels in the subtree rooted at v, if any. Since we have multiple subsets we need to find predecessors in, we actually store at each v a *mapping* from each subset S_k to the minimum and maximum leader of S_k in the subtree rooted at v. For efficiency, for each leader $v \in \mathcal{T}$ we store a hash table H_v, mapping $k \in [q]$ to the tuple $(\min\{u : u \in \mathcal{L}[T_k] \text{ and } (u, k) \in \mathcal{T}_v\}, \max\{u : u \in \mathcal{L}[T_k] \text{ and } (u, k) \in \mathcal{T}_v\})$, if it exists.

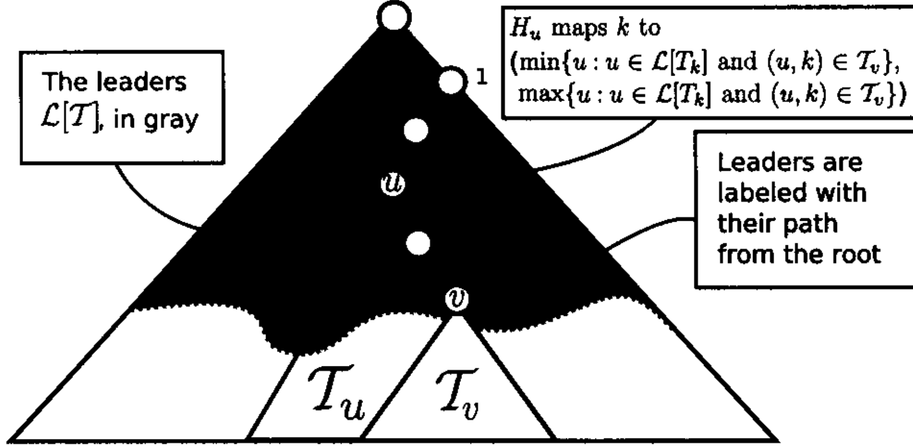

Figure 2: Treap $\mathcal{T}$ storing $\hat{L}$.

Recall $\mathcal{T}_v$ is the subtreap of $\mathcal{T}$ rooted at v. The high-level idea is to use the hash tables H_v to find the right "neighborhood" of $O(\log m)$ elements in T_k which we will have to update (in the event of an update to some S_k), or search (in the event of a predecessor or successor query). Since these neighborhoods are stored as treaps, updating and searching them takes expected $O(\log\log m)$ time. We summarize these definitions, along with some others, in Table 1.

$w(x, T)$	number of descendants of node x of treap T
$\ell(x, T)$	the partition leader of x in T
$\mathcal{L}[T]$	weight $s = \Theta(\log m)$ partition leaders of treap T
T_k	treap containing all elements of the ordered subset S_k, $k \in [q]$
$T(L^{\leq})$	the treap on L
$T(\mathcal{L}[L^{\leq}])$	the subtreap of $T(L^{\leq})$ on the weight $s = \Theta(\log m)$ leaders of $T(L^{\leq})$
$\hat{L}$	the set $\{(x,k) : x \in S_k\} \cup \{(x,0) : x \in \mathcal{L}[T(L^{\leq})]\}$
$\mathcal{T}$	a treap storing $\hat{L}$
H	hash table mapping label $i \in \{0,1\}^m$ to a pointer to the leader of $\mathcal{T}$ with label i
H_v	hash table mapping $k \in [q]$ to the tuple (if it exists) $(\min\{u \,:\, u \in \mathcal{L}[T_k] \wedge (u,k) \in \mathcal{T}_v\},\ \max\{u \,:\, u \in \mathcal{L}[T_k] \wedge (u,k) \in \mathcal{T}_v\})$
I_x	for $x \in L$, a fast ordered dictionary [4] mapping each $k \in \{i : x \in S_i\}$ to (x,k) in $\mathcal{T}$
J_x	for $x \in \mathcal{L}[T(L^{\leq})]$, a treap containing $\{u \in L : \ell(u, T(L^{\leq})) = x$ and $\exists i : u \in S_i\}$

Table 1: Some useful notation and definitions of various structures we maintain.

We use the following Lemma to bound the number of changes on partition leaders.

Lemma 3.2 *[4] Let $s \in \mathbb{Z}^+$ and let T be a treap of size at least s. Let T' be the treap induced on the weight s partition leaders in T. Then the probability that inserting a new element into T or deleting an element from T alters the structure of T' is at most c/s for some global constant c.*

Note that each partition set has size at most $O(\log m)$. The treaps T_k, J_x and $\mathcal{T}$, and the dictionaries I_x from Table 1 are stored explicitly. We also store the minimum and maximum element of each $\mathcal{L}[T_k]$ explicitly. We use a total ordering for $\hat{L}$ as follows: $(x, k) < (x', k')$ if $x < x'$ or if $x = x'$ and $k < k'$.

OMP Insert & Delete Operations: These operations remain largely the same as in the order maintenance structure of [4]. We assume that when $x \in L$ is deleted it is not in any set S_k. The main difference is that if the set $\mathcal{L}[T(L^{\leq})]$ changes we will need to update the treaps $\{J_v : v \in \mathcal{L}[T(L^{\leq})]\}$, $\mathcal{T}$, and the tables $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ appropriately.

Note that we can easily update H_v in time linear in $|\mathcal{T}_v|$ using in-order traversal of $\mathcal{T}_v$, assuming we can test if x is in $\mathcal{L}[T_k]$ in $O(1)$ time. To accomplish this, for each k we can store $\mathcal{L}[T_k]$ in a hash table. Thus using Theorem 2.3 we can see that all necessary updates to $\{H_v : v \in \mathcal{T}\}$ take expected $O(\log m)$ time. Clearly, updating $\mathcal{T}$ itself requires only expected $O(\log m)$ time. Finally, we bound the time to update the treaps J_v by the total cost to update $T(\mathcal{L}[L^{\leq}])$ if the rotation of subtrees of size k costs $k + \log m$, which is $O(\log m)$ by Theorem 2.3. This bound holds because $|J_v| = O(\log m)$ for any v, and any tree rotation on $T(L^{\leq})$ causes at most $3s$ elements of $T(L^{\leq})$ to change their weight s leader. Therefore only $O(\log m)$ elements need to be added or deleted from the treaps $\{J_v : v \in T(\mathcal{L}[L^{\leq}])\}$, and we can batch these updates in such a way that each takes expected amortized $O(1)$ time. However, we need only make these updates if $\mathcal{L}[T(L^{\leq})]$ changes, which by Lemma 3.2 occurs with probability $O(1/\log m)$. Hence the expected overall cost is $O(1)$.

Predecessor & Successor: Suppose we wish to find the predecessor of x in S_k. (Finding the successor is analogous.) If $x \in S_k$ we can test this in expected $O(\log\log m)$ time using I_x. So suppose $x \notin S_k$. We will first find the predecessor w of (x, k) in $\mathcal{T}$ as follows. (We can handle the case that w does not exist by adding a special element to L that is smaller than all other elements and is considered to be part of $\mathcal{L}[T(L^{\leq})]$). First search I_x for the predecessor k_2 of k in $\{i : x \in S_i\}$ in $O(\log\log m)$ time. If k_2 exists, then $w = (x, k_2)$. Otherwise, let y be the leader of x in $T(L^{\leq})$, and let y' be the predecessor of y in $\mathcal{L}[T(L^{\leq})]$. Then either $w \in \{(y', 0), (y, 0)\}$ or else $w = (z, k_3)$, where $z = \max\{u : u < x \text{ and } u \in J_y \cup J_{y'}\}$ and $k_3 = \max\{i : z \in S_i\}$. Thus we can find w in expected $O(\log\log m)$ time using fast finger search for y', treap search on the $O(\log m)$ sized treaps in $\{J_v : v \in \mathcal{L}[T(L^{\leq})]\}$, and the fast dictionaries $\{I_x : x \in L\}$.

We will next find the predecessor w' or the successor w'' of x in $\mathcal{L}[T_k]$. Note that if we have w' we can find w'' quickly via fast finger search, and vice versa. If $w \in \mathcal{L}[T_k] \times \{k\}$, then $w = w'$ and we can test this in constant time if we store the subtree sizes in the treaps T_k. So assume $w \notin \mathcal{L}[T_k] \times \{k\}$. We start from w and consider its leader $\ell(w)$ in $\mathcal{T}$. We first binary search on the path P from the root of $\mathcal{T}$ to $\ell(w)$ for the deepest node u' such that $\mathcal{T}_{u'}$ contains at least one node from $\mathcal{L}[T_k] \times \{k\}$. (In particular, this means that this subtreap also contains a node (u, k) where u is either the predecessor or successor of x in $\mathcal{L}[T_k]$. If no node on the path has this property, then S_k is empty.) The binary search is performed on the length of the prefix of the label of $\ell(w)$. Given a prefix α, we look up the node p with label α using H, and test whether $\mathcal{T}_p$ contains at least one node from $\mathcal{L}[T_k] \times \{k\}$ using H_p. If so, we increase the prefix length. Otherwise we decrease it.

Next use $H_{u'}$ to obtain $u_{\min} = \min\{u : u \in \mathcal{L}[T_k] \text{ and } (u,k) \in \mathcal{T}_{u'}\}$ and $u_{\max} = \max\{u : u \in \mathcal{L}[T_k] \text{ and } (u,k) \in \mathcal{T}_{u'}\}$. Note that either (1) $u_{\max} < x$, or (2) $u_{\min} > x$, or (3) $u_{\min} < x < u_{\max}$.

In the first case, we claim that $u_{\max}$ is the predecessor of x in $\mathcal{L}[T_k]$. To prove this, note that the predecessor of x in $\mathcal{L}[T_k]$ is the predecessor of w in $\mathcal{L}[T_k]$ under the extended ordering for $\hat{L}$. Now suppose for a contradiction that $z \neq u_{\max}$ is the predecessor of w in $\mathcal{L}[T_k]$. Thus $u_{\max} < z \leq w$. Clearly, $z \notin \mathcal{T}_{u'}$, for otherwise we obtain a contradiction from the definition of $u_{\max}$. However, it is easy to see that every node that is not in $\mathcal{T}_{u'}$ is either less than $u_{\max}$ or greater than w, contradicting the assumption that $u_{\max} < z \leq w$. In the second case, we claim that $u_{\min}$ is the successor of x in $\mathcal{L}[T_k]$. The proof is analogous to the first case, and we omit it.

Thus, in the first two cases we are done. Consider the third case. We first prove that $u' = \ell(w)$ in this case. Note that $u_{\min} < x < u_{\max}$ implies $u_{\min} \leq w < u_{\max}$. Since there is an element of $\mathcal{L}[T_k] \times \{k\}$ on either side of w in $\mathcal{T}_{u'}$, the child u'_c of u' that is an ancestor of w must have an element of $\mathcal{L}[T_k] \times \{k\}$ in its subtree. From the definition of u', we may infer that u'_c was not on the $\ell(w)$ to root path in $\mathcal{T}$. Thus u' is the deepest node on this path, or equivalently $u' = \ell(w)$. Given that $u' = \ell(w)$ and $u_{\min} \leq w < u_{\max}$, we next prove that at least one element of $\{u_{\min}, u_{\max}\}$ is within distance $s = \Theta(\log m)$ of w in $\mathcal{T}$. Note that because $u'_c \notin \mathcal{L}[\mathcal{T}]$, its subtree $\mathcal{T}_{u'_c}$ has size at most s. If u'_c is the left child of u', then $u_{\min}$ is in this subtree as is w. The distance between $u_{\min}$ and w is thus at most $s = \Theta(\log m)$ in $\mathcal{T}$. If u'_c is the right child of u', then we may argue analogously that $u_{\max}$ and w are both in $\mathcal{T}_{u'_c}$ and thus $u_{\max}$ is within distance s of w.

Once we have obtained a node (u_{near}, k) in $\mathcal{T}$ such that $u_{\text{near}} \in \mathcal{L}[T_k]$ and (u_{near}, k) is within distance $d = O(\log m)$ of w in $\mathcal{T}$, we find the predecessor w' and successor w'' of x in $\mathcal{L}[T_k]$ via fast finger search on T_k using u_{near}. Note that the distance between u_{near} and the nearest of $\{w', w''\}$ is at most d. This is because if $u_{\text{near}} \leq w'$, then every element e of $\mathcal{L}[T_k]$ between u_{near} and w' must have a corresponding node $(e,k) \in \mathcal{T}_{u'_c}$ between (u_{near}, k) and w, and there are at most d such nodes. Similarly, if $u_{\text{near}} \geq w''$, then every element e of $\mathcal{L}[T_k]$ between w'' and u_{near} must have a corresponding node $(e,k) \in \mathcal{T}_{u'_c}$ between w and (u_{near}, k), and there are at most d such nodes. Note that finding the predecessor and successor of x in T_k given a handle to a node in T_k at distance d from x takes $O(\log(d))$ time in expectation [25]. In this case, $d = O(\log m)$ so this step takes $O(\log\log m)$ time in expectation.

Once we have found w' and w'', the predecessor and successor of x in $\mathcal{L}[T_k]$, we simply search their associated partitions of S_k for the predecessor of x in S_k. These partitions are both of $O(\log m)$ size and can be searched in expected $O(\log\log m)$ time. The total time to find the predecessor is thus $O(\log\log m)$ in expectation.

OSP-Insert and OSP-Delete: *OSP-Delete* is analogous to *OSP-Insert*, hence we focus on *OSP-Insert*. Suppose we wish to add x to S_k. First, if x is not currently in any sets $\{S_i : i \in [q]\}$, then find the leader of x in $T(L^{\leq})$, say y, and insert x into J_y in expected $O(\log\log m)$ time. Next, insert x into T_k as follows. Find the predecessor w of x in S_k, then insert x into T_k in expected $O(1)$ time starting from w to speed up the insertion.

Find the predecessor w' of (x,k) in $\mathcal{T}$ as in the predecessor operation, and insert (x,k) into $\mathcal{T}$ using w' as a starting point. If neither $\mathcal{L}[T_k]$ nor $\mathcal{L}[\mathcal{T}]$ changes, then no modifications to $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ need to be made. If $\mathcal{L}[T_k]$ does not change but $\mathcal{L}[\mathcal{T}]$ does, as happens with probability $O(1/\log m)$, we can update $\mathcal{T}$ and $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ appropriately in expected $O(\log m)$ time

by taking time linear in the size of the subtree to do rotations. If $\mathcal{L}[T_k]$ changes, we must be careful when updating $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$. Let $\mathcal{L}[T_k]$ and $\mathcal{L}[T_k]'$ be the leaders of T_k immediately before and after the addition of x to S_k, and let $\Delta_k := (\mathcal{L}[T_k] - \mathcal{L}[T_k]') \cup (\mathcal{L}[T_k]' - \mathcal{L}[T_k])$. Then we must update $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ appropriately for all nodes $v \in \mathcal{L}[\mathcal{T}]$ that are descendants of (x, k) as before, but must also update H_v for any node $v \in \mathcal{L}[\mathcal{T}]$ that is an ancestor of some node in $\{(u, k) : u \in \Delta_k\}$. It is not hard to see that these latter updates can be done in expected $O(|\Delta_k| \log m)$ time. Moreover, $\mathbf{E}\big[|\Delta_k| \mid x \notin \mathcal{L}[T_k]'\big] \leq 1$ and $\mathbf{E}\big[|\Delta_k| \mid x \in \mathcal{L}[T_k]'\big] = O(1)$, since $|\Delta_k|$ can be bounded by $2(R+1)$, where R is the number of rotations necessary to rotate x down to a leaf node in a treap on $\mathcal{L}[T_k]'$. This is essentially because each rotation can cause $|\Delta_k|$ to increase by at most two. Since it takes $\Theta(R)$ time to delete x given a handle to it, from Theorem 2.3 we easily infer $\mathbf{E}[R] = O(1)$. Since the randomness for T_k is independent of the randomness used for $\mathcal{T}$, these expectations multiply, for a total expected time of $O(\log m)$, conditioning on the fact that $\mathcal{L}[T_k]$ changes. Since $\mathcal{L}[T_k]$ only changes with probability $O(1/\log m)$, this part of the operation takes expected $O(1)$ time. Finally, insert k into I_x in expected $O(\log \log m)$ time, with a pointer to (x, k) in $\mathcal{T}$.

Space Analysis. We now prove that our ordered subsets data structure uses $O(m)$ space in expectation, where $m = |\hat{L}|$. Table 1 lists the objects that that the data structure uses. The order maintenance structure uses $O(|L|)$ space. The hash table H, treap $\mathcal{T}$, the collection of treaps $\{T_k : k \in [q]\}$, and the collection fast dictionaries $\{I_x : x \in L\}$ each use only $O(m)$ space. The collection $\{J_x : x \in \mathcal{L}[T(L^{\leq})]\}$ uses only $O(|L|)$ space. That leaves the space required by $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$. We claim these hash tables use $O(m)$ space in expectation. To prove this, let $X_{u,k}$ be a random variable denoting the number of hash tables in $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ that map k to a tuple of the form $(u, *)$ or $(*, u)$, where $*$ denotes a wildcard that matches all nodes or null. (If H_v maps k the record (u, u), we may count that record as contributing two to $X_{u,k}$). The space required for $\{H_v : v \in \mathcal{L}[\mathcal{T}]\}$ is then linear in the total number of entries of all hash tables, which is $\sum_{u \in L} \sum_{k=1}^{q} X_{u,k}$. Clearly, if $u \notin S_k$ then $X_{u,k} = 0$. On the other hand, we claim that if $x \in S_k$ then $\mathbf{E}[X_{u,k}] = O(1)$, in which case $\mathbf{E}\big[\sum_{u \in L} \sum_{k=1}^{q} X_{u,k}\big] = O(m)$ by linearity of expectation. Assume $u \in S_k$. Note that if $u \notin \mathcal{L}[T_k]$, then $X_{u,k} = 0$, and $\mathbf{Pr}[u \in \mathcal{L}[T_k]] = O(1/\log m)$ (the probability of any node being a weight s leader is $O(1/s)$, which is an easy corollary of Theorem 2.3). Furthermore

$$\begin{aligned} \mathbf{E}[X_{u,k} \mid u \in \mathcal{L}[T_k]] &\leq \mathbf{E}[\text{depth of } (u, k) \text{ in } \mathcal{T}] \\ &= O(\log m) \end{aligned}$$

It follows that

$$\begin{aligned} \mathbf{E}[X_{u,k}] &= \mathbf{E}[X_{u,k} \mid u \in \mathcal{L}[T_k]] \cdot \mathbf{Pr}[u \in \mathcal{L}[T_k]] \\ &= O(\log m \cdot \tfrac{1}{\log m}) \\ &= O(1) \end{aligned}$$

4 Uniquely Represented Range Trees

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\mathbb{R}^d$. The well studied *orthogonal range reporting* problem is to maintain a data structure for P while supporting queries which given an axis aligned

box B in $\mathbb{R}^d$ returns the points $P \cap B$. The dynamic version allows for the insertion and deletion of points. Chazelle and Guibas [8] showed how to solve the two dimensional dynamic problem in $O(\log n \log\log n)$ update time and $O(\log n \log\log n + k)$ query time, where k is the size of the output. Their approach used fractional cascading. More recently Mortensen [17] showed how to solve it in $O(\log n)$ update time and $O(\log n + k)$ query time using a sophisticated application of Fredman and Willard's q-heaps [12]. All of these techniques can be generalized to higher dimensions at the cost of replacing the first $\log n$ term with a $\log^{d-1} n$ term [9].

Here we present a uniquely represented solution to the problem. It matches the bounds of the Chazelle and Guibas version, except ours are in expectation instead of worst-case bounds. Our solution does not use fractional cascading and is instead based on ordered subsets. One could probably derive a UR version based on fractional cascading, but making dynamic fractional cascading UR would require significant work[2] and is unlikely to improve the bounds. Our solution is simple and avoids any explicit discussion of weight balanced trees (the required properties fall directly out of known properties of treaps).

Theorem 4.1 *Let P be a set of n points in $\mathbb{R}^d$. There exists a UR data structure for the orthogonal range query problem that uses expected $O(n \log^{d-1} n)$ space and $O(d \log n)$ random bits, supports point insertions or deletions in expected $O(\log^{d-1} n \cdot \log\log n)$ time, and queries in expected $O(\log^{d-1} n \cdot \log\log n + k)$ time, where k is the size of the output.*

If $d = 1$, simply use the dynamic ordered dictionaries solution [4] and have each element store a pointer to its successor for fast reporting. For simplicity we first describe the two dimensional case. The remaining cases with $d \geq 3$ can be implemented using standard techniques [9] if treaps are used for the underlying hierarchical decomposition trees, as we describe below.

We will assume that the points have distinct coordinate values; thus, if $(x_1, x_2), (y_1, y_2) \in P$, then $x_i \neq y_i$ for all i. (There are various ways to remove this assumption, e.g., the composite-numbers scheme or symbolic perturbations [9].) We store P in a random treap T using the ordering on the first coordinate as our BST ordering. We additionally store P in a second random treap T' using the ordering on the second coordinate as our BST ordering, and also store P in an ordered subsets instance D using this same ordering. We cross link these and use T' to find the position of any point we are given in D. The subsets of D are $\{T_v : v \in T\}$, where T_v is the subtree of T rooted at v. We assign each T_v a unique integer label k using the coordinates of v, so that T_v is S_k in D. The structure is UR as long as all of its components (the treap and ordered subsets) are uniquely represented.

To insert a point p, we first insert it by the second coordinate in T' and using the predecessor of p in T' insert a new element into the ordered subsets instance D. This takes $O(\log n)$ expected time. We then insert p into T in the usual way using its x coordinate. That is, search for where p would be located in T were it a leaf, then rotate it up to its proper position given its priority. As we rotate it up, we can reconstruct the ordered subset for a node v from scratch in time $O(|T_v| \log\log n)$. Using Theorem 2.3, the overall time is $O(\log n \log\log n)$ in expectation. Finally, we must insert p into the subsets $\{T_v : v \in T$ and v is an ancestor of $p\}$. This requires expected $O(\log\log n)$ time per ancestor, and there are only $O(\log n)$ of them in expectation. Since these expectations

[2]We expect a variant of Sen's approach [26] could work.

are computed over independent random bits, they multiply, for an overall time bound of $O(\log n \cdot \log\log n)$ in expectation. Deletion is similar.

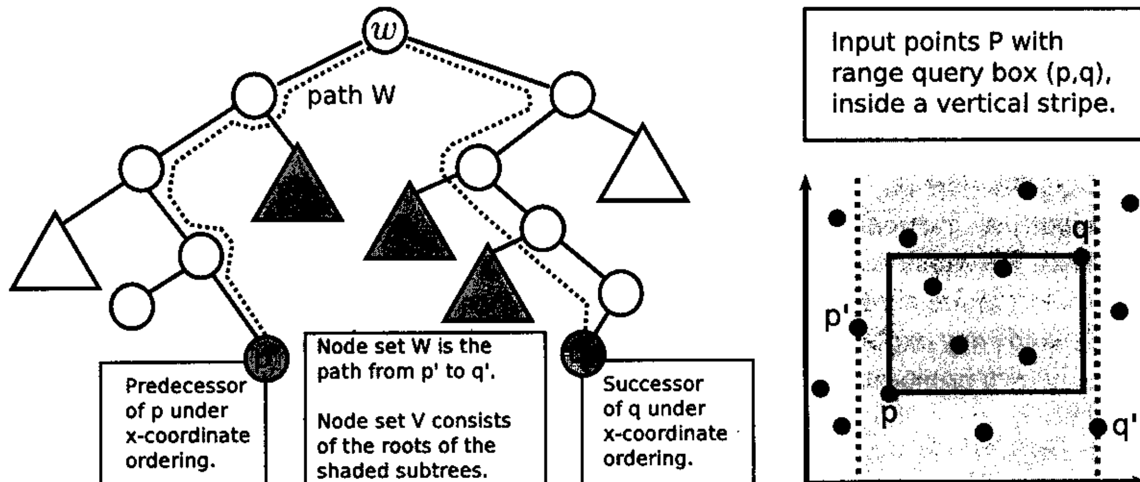

Figure 3: Answering 2-D range query (p, q). Path W together with the shaded subtrees contain the nodes in the vertical stripe between p' and q'.

To answer a query $(p, q) \in \mathbb{R}^2 \times \mathbb{R}^2$, where $p = (p_1, p_2)$ is the lower left and $q = (q_1, q_2)$ is the upper right corner of the box B in question, we first search for the predecessor p' of p and the successor q' of q in T (i.e., with respect to the first coordinate). Please refer to Figure 3. We also find the predecessor p'' of p and successor q'' of q in T' (i.e., with respect to the second coordinate). Let w be the least common ancestor of p' and q' in T, and let $A_{p'}$ and $A_{q'}$ be the paths from p' and q' (inclusive) to w (exclusive), respectively. Let V be the union of right children of nodes in $A_{p'}$ and left children of nodes in $A_{q'}$, and let $\mathcal{S} = \{T_v : v \in V\}$. It is not hard to see that $|V| = O(\log n)$ in expectation, that the sets in $\mathcal{S}$ are disjoint, and that all points in B are either in $W := A_{p'} \cup \{w\} \cup A_{q'}$ or in $\cup_{S \in \mathcal{S}} S$. Compute W's contribution to the answer, $W \cap B$, in $O(|W|)$ time by testing each point in turn. Since $\mathbf{E}[|W|] = O(\log n)$, this requires $O(\log n)$ time in expectation. For each subset $S \in \mathcal{S}$, find $S \cap B$ by searching for the successor of p'' in S, and doing an in-order traversal of the treap in D storing S until reaching a point larger than q''. This takes $O(\log\log n + |S \cap B|)$ time in expectation for each $S \in \mathcal{S}$, for a total of $O(\log n \cdot \log\log n + k)$ expected time.

To bound the space required, note that each point p is stored in $\text{depth}(p) + 1$ subsets in the ordered subsets instance D, where $\text{depth}(p)$ is the depth of p in treap T. Since $\mathbf{E}[\text{depth}(p)] = O(\log n)$, we infer that $\mathbf{E}[\sum_v |T_v|] = O(n \log n)$, and so D uses $O(n \log n)$ space in expectation. Since the storage space is dominated by the requirements of D, this is also the space requirement for the whole data structure.

Extending to Higher Dimensions. We show how to support orthogonal range queries for $d \geq 3$ dimensions by reducing the d-dimensional case to the $(d-1)$-dimensional case. That is, we complete the proof Theorem 4.1 via induction on d, where the base cases $d \in \{1, 2\}$ are proven above. So assume Theorem 4.1 is true in the $(d-1)$-dimensional case. Let $P \subset \mathbb{R}^d$ be the input set of n points as before, and suppose the dimensions are labeled $\{1, 2, \dots, d\}$. Store P in a random treap T using the dth coordinate of the points to determine the BST ordering, and a fresh 8-wise independent hash function to generate priorities. (Using fresh random bits implies that certain random variables are independent, and hence the expectation of their product is the product of their expectations. This fact aids our analysis considerably.) For each node $v \in T$, maintain a $(d-1)$-dimensional uniquely represented range tree data structure R_v on the points in T_v, where the point $(x_1, x_2, \dots, x_d)$ is treated as the $(d-1)$-dimensional point $(x_1, x_2, \dots, x_{d-1})$. Inserting a point p then involves inserting p into T and modifying $\{R_v : v \in T\}$ appropriately. Deleting a point is analogous. We can bound the running time for these operations as follows.

Inserting p into T takes $O(\log n)$ time. We will also need to update R_v for each v that is an ancestor of p in T, by inserting p into it. Note that p has expected $O(\log n)$ depth by Theorem 2.3, and we can insert p into any R_v in expected $O(\log^{d-2} n \cdot \log\log n)$ time by the induction hypothesis, Thus we can update $\{R_v : v \text{ is an ancestor of } p \text{ in } T\}$ in expected $O(\log^{d-1} n \cdot \log\log n)$ time. (Here we have used the fact that the depth of p and the time to insert p into some fixed R_v are independent random variables.) Finally, inserting p into T will in general have involved rotations, thus requiring significant changes to some of the structures R_v for nodes v that are descendants of p in T. However, it is relatively easy to see that we can rotate along an edge $\{u, v\}$ and update R_u and R_v in expected time

$$O\left((|T_u| + |T_v|) \log^{d-2} n \cdot \log\log n\right)$$

using the induction hypothesis. Using Theorem 2.3 with rotation cost $f(k) = O(k \log^{d-2} n \cdot \log\log n)$ then implies that these rotations take a total of $O(\log^{d-1} n \cdot \log\log n)$ expected time. (Here we rely on the fact that for any fixed u and v, $|T_u|$ and the time to update R_v are independent.) This yields an overall running time bound for insertions of expected $O(\log^{d-1} n \cdot \log\log n)$ time. The same argument applies to deletions as well.

Queries in higher dimensions resemble queries in two dimensions. Given a d-dimensional box query (p, q), we find the predecessor p' of p in T and the successor q' of q in T. Let w be the least common ancestor of p' and q' and let $A_{p'}$ and $A_{q'}$ be the paths from p' and q' (inclusive) to w (exclusive), respectively. Let V be the union of right children of nodes in $A_{p'}$ and left children of nodes in $A_{q'}$. For each $v \in A_{p'} \cup \{w\} \cup A_{q'}$, test if v is in box (p, q). This takes $O(d \log n)$ time in expectation, which is $O(\log^{d-1} n)$ for $d \geq 2$. Finally, issue a query $(\bar{p}, \bar{q})$ to each R_v for each $v \in V$, where $\bar{x}$ is the projection of x onto the first $(d-1)$ dimensions, so that if $x = (x_1, \dots, x_d)$ then $\bar{x} = (x_1, \dots, x_{d-1})$. The results of these queries are disjoint, each takes $O(\log^{d-2} n \cdot \log\log n + k)$ time in expectation by the induction hypothesis, and and there are $O(\log n)$ of them in expectation. Since the query times (conditioned on the set of points stored in each structure R_v) and the number of queries made are independent random variables, the total running time is $O(\log^{d-1} n \cdot \log\log n + k)$ in expectation, where k is the size of the output.

We now show that the space usage of our data structure is $O(n \log^{d-1} n)$ in expectation. As before, we proceed by induction on d. Assume that the space usage is $O(n \log^{d-2} n)$ in expectation

for a $(d-1)$-dimensional point set. The space usage is dominated by the structures $\{R_v : v \in T\}$, which by the induction hypothesis and linearity of expectation require

$$O\left(\sum_{v\in T} |T_v| \log^{d-2} |T_v|\right) \quad \text{which is} \quad O\left(\sum_{v\in T} |T_v| \log^{d-2} n\right)$$

space in expectation, where T is a random treap. Computing the expectation over the choice of random treap, the space usage is thus bounded by

$$\mathbf{E}\left[\sum_{v\in T} |T_v| \log^{d-2} n\right] = \mathbf{E}\left[\sum_{v\in T} |T_v|\right] \cdot \log^{d-2} n$$

However treaps have expected logarithmic subtree size [25], so $\mathbf{E}\left[\sum_{v\in T} |T_v|\right] = \sum_{v\in T} \mathbf{E}[|T_v|] = \sum_{v\in T} O(\log n) = O(n \log n)$. The total space required is therefore $O(n \log^{d-1} n)$.

5 Horizontal Point Location & Orthogonal Segment Intersection

Let $S = \{(x_i, x'_i, y_i) : i \in [n]\}$ be a set of n horizontal line segments. In the *horizontal point location problem* we are given a point $(\hat{x}, \hat{y})$ and must find $(x, x', y) \in S$ maximizing y subject to the constraints $x \le \hat{x} \le x'$ and $y < \hat{y}$. In the related *orthogonal segment intersection problem* we are given a vertical line segment $s = (x, y, y')$, and must report all segments in S intersecting it, namely $\{(x_i, x'_i, y_i) : x_i \le x \le x'_i \text{ and } y \le y_i \le y'\}$. In the dynamic version we must additionally support updates to S. As with the orthogonal range reporting problem, both of these problems can be solved using fractional cascading and in the same time bounds [8] ($k = 1$ for point location and is the number of lines reported for segment intersection). Mortensen [16] improved orthogonal segment intersection to $O(\log n)$ updates and $O(\log n + k)$ queries.

We extend our ordered subsets approach to obtain the following results for horizontal point location and range reporting.

Theorem 5.1 *Let S be a set of n horizontal line segments in $\mathbb{R}^2$. There exists a uniquely represented data structure for the point location and orthogonal segment intersection problems that uses $O(n \log n)$ space, supports segment insertions and deletions in expected $O(\log n \cdot \log\log n)$ time, and supports queries in expected $O(\log n \cdot \log\log n + k)$ time, where k is the size of the output. The data structure uses $O(\log n)$ random bits.*

5.1 The Data Structures

We will first obtain a hierarchical decomposition $\mathcal{D}$ of the plane into vertical slabs (in a manner akin to segment trees) using a random treap T on the endpoints E of segments in S. The treap T uses the natural ordering on the first coordinate to determine the BST ordering. For $a, b \in \mathbb{R}^2$ with $a = (a_x, a_y)$ and $b = (b_x, b_y)$, we let $[a, b]$ denote the vertical slab $\{(x, y) : a_x \le x \le b_x, y \in \mathbb{R}\}$.

The decomposition has as its root the whole plane; for concreteness we may imagine it as the vertical slab $[(-\infty, 0), (+\infty, 0)]$. A node $[a, b]$ in $\mathcal{D}$ has children $[a, c]$ and $[c, b]$ if $c = (c_x, c_y) \in E$ is the highest priority node in T such that $a_x < c_x < b_x$. Note that the decomposition tree $\mathcal{D}$ has nearly the same structure as T. To obtain the structure of $\mathcal{D}$ from T, it suffices to add nodes to T so that the root has degree two, and every other original node in T has degree three. It will be useful to associate each node $v \in T$ with a node $\bar{v} \in \mathcal{D}$, as follows: label the nodes of T and $\mathcal{D}$ as in a trie, and for $u \in T$ and $w \in \mathcal{D}$ let $w = \bar{u}$ iff u and w have the same label.

Each $[a, b] \in \mathcal{D}$ also has an associated subset of line segments in S, which we denote by $S_{[a,b]}$. In particular, line segment $(x, x', y) \in S$ is *associated* with $[a, b]$ if $[a, b] \subseteq [x, x']$ and for all ancestors $[a', b']$ of $[a, b]$, $[a', b'] \nsubseteq [x, x']$. Note that $s \in S$ may be associated with as many as $O(\log n)$ nodes in $\mathcal{D}$, in expectation. We store the sets $\{S_{[a,b]} : [a, b] \in \mathcal{D}\}$ in an ordered subsets structure, using the natural order on the second coordinate as our total order. As with range searching we also keep a treap T' on S ordered by the second coordinate which is used to insert new elements into the ground set L of the ordered subset structure.

To answer a point location query on a point $p = (x, y)$, first search for the narrowest slab $[a, b] \in \mathcal{D}$ with $a \leq x \leq b$. Let P be the path from this node to the root of $\mathcal{D}$. Insert y into L of the OSP instance (using T') and for each $[a, b] \in P$, search $S_{[a,b]}$ for the predecessor of y using ordered subsets. Of these $|P|$ segments, return the highest one.

To answer a segment intersection query on a vertical segment (x, y, y'), find the path P as in a point location query for (x, y). For each $[a, b] \in P$, search $S_{[a,b]}$ for the successor s_i of y, and report in order all segments in $S_{[a,b]}$ from s_i to the greatest segment at height at most y'.

To insert a segment (x, x', y), insert (x, y) and (x', y) into the treap T. As (x, y) and (x', y) are rotated up to their correct positions, modify $\mathcal{D}$ accordingly and construct the sets $S_{[a,b]}$ of newly created slabs $[a, b]$ from scratch. We construct a set $S_{[a,b]}$ as follows. For each descendant e of a or b in T, determine if the segment with endpoint e is associated with $[a, b]$ in constant time. If so, insert the segment into $S_{[a,b]}$. In order to guarantee that this procedure correctly constructs $S_{[a,b]}$, we show the following: Every segment associated with $[a, b]$ in $\mathcal{D}$ has an endpoint that is a descendant of either a or b in T. See Claim 5.2 for the proof.

Finally, we may need to insert (x, x', y) into sets $S_{[a,b]}$ for slabs $[a, b]$ that were not affected by the insertions of x and x' into T and the corresponding modifications to $\mathcal{D}$. To find the sets to modify, find the path P from (x, y) to (x', y) in T, and consider $S_{[a,b]}$ for each $[a, b]$ that is a *child* of some node in $\{\bar{v} : v \in P\}$. For each, test in constant time if the new segment should be added to it, and add it accordingly. Deletion is similar.

5.2 The Analysis

We start with the running time for queries. Note that the decomposition tree $\mathcal{D}$ has nearly the same structure as T. To obtain the structure of $\mathcal{D}$ from T, it suffices to add nodes to T so that the root has degree two, and every other original node in T has degree three. Thus the path P has expected logarithmic length and can be found in logarithmic time. Performing the $O(|P|)$ predecessor queries takes expected $O(|P| \log \log n)$ time. In a point location query, finding the maximum height result takes $O(|P|)$ time for a total of expected $O(\log n \cdot \log \log n)$ time. For a segment intersection query, if there are $k_{[a,b]}$ segments in $S_{[a,b]}$ intersecting the query segment, we

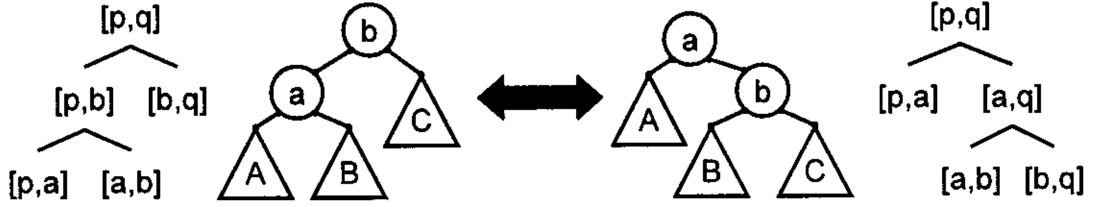

Figure 4: Rotation of the decomposition $\mathcal{D}$ and treap T about edge $\{a, b\}$, starting with slab $[p, q]$.

can report them in expected time $O(\log\log n + k_{[a,b]})$ by either performing fast finger searches in the treap $T_{[a,b]}$ which stores $S_{[a,b]}$ (thus finding the successor of a node in expected $O(1)$ time), or by storing pointers at each treap node $v \in T_{[a,b]}$ to its successor. The total expected time is thus $O(|P|\log\log n + \sum_{[a,b]\in P} k_{[a,b]})$. Since each segment in the answer appears in exactly one set $S_{[a,b]}$, this total is $O(\log n \cdot \log\log n + k)$.

We now analyze the running time for insertions. We first bound the cost to insert (x, y) and (x', y) into T, do rotations in $\mathcal{D}$, and create $S_{[a,b]}$ for newly created slabs. Note that this costs the same as updating T, up to constant factors, if the cost to rotate a subtree of size z is $z \log\log n$. Thus, by Theorem 2.3 the update time per insertion is $O(\log n \cdot \log\log n)$ in expectation. Next we bound the cost to update $S_{[a,b]}$ for preexisting slabs $[a, b]$. Let P be the path from (x, y) to (x', y) in T. It is easy to prove using Theorem 2.3 that the expected length of P is $O(\log n)$. Thus the total time to make these updates is again $O(\log n \cdot \log\log n)$ in expectation. The analysis for deletions is similar.

Finally we consider the space usage. Using Claim 5.2 and the definition of segment association, it is not difficult to prove that a segment with endpoints e and e' can be associated with at most $|P_{e,e'}|$ slabs, where $P_{a,b}$ is the path from a to b in T. Since this is logarithmic in expectation, we conclude that each segment is stored at most $O(\log n)$ times in expectation. Since treaps and our ordered subset structure take linear space in expectation, the total space usage is thus $O(n \log n)$ in expectation.

Claim 5.2 *In the data structure of section 5.1, every segment associated with* $[a, b]$ *in* $\mathcal{D}$ *has an endpoint that is a descendant of either* a *or* b *in* T.

Proof: Fix a segment $s = (x, x', y)$ with endpoints $e = (x, y), e' = (x', y)$. Let P be the treap path from e to e', and let T' be the subtree of T containing P and all descendants of nodes in P. Suppose for a contradiction that s is associated with $[a, b]$ but neither of its endpoints is a descendant of a or b. Thus $[a, b] \subseteq [x, x']$ and for all ancestors $[a', b']$ of $[a, b]$, $[a', b'] \nsubseteq [x, x']$. Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$. Since s is associated with $[a, b]$, this implies $x < a_x \le b_x < x'$. Note that $[a, b] \in \mathcal{D}$ implies that one of $\{a, b\}$ is a descendant of the other in T. Suppose b is a descendant of a (the other case is symmetric). We consider two cases: $a \in T'$ and $a \notin T'$.

In the first case, clearly $a \notin P$, so a must be a descendant of some node $v \in P$. Then if $c = (c_x, c_y)$ is the parent of a, then either $c_x < a_x$ or $c_x > b_x$, since otherwise $[a, b]$ would not be in

$\mathcal{D}$. However, $c \in T'$, thus $x < c_x < x'$, and so either $[a, c]$ or $[c, b]$ contains $[a, b]$ and is contained in $[x, x']$, a contradiction.

In the second case, $a \notin T'$. By assumption, neither e nor e' is a descendant of a, and $x < a_x < x'$, so there must be a treap node d with $x < d_x < a_x$ with higher priority than both a and e, and a node d' with $a_x < d'_x < x'$ with higher priority than both a and e'. However, a must be a descendant of at least one node in $\{d, d'\}$, and P must pass through ancestors of both d and d' (where each node is included among its own ancestors), contradicting the case assumption that $a \notin T'$. ■

6 Uniquely Represented 2-D Dynamic Convex Hull

In this section we obtain a uniquely represented data structure for maintaining the convex hull of a dynamic set of points $S \subset \mathbb{R}^2$. The *convex hull* of S is defined as the minimal convex set containing S. The *vertices* of the convex hull of S are those points which cannot be written as convex combinations of the other points in S. For our purposes, the convex hull is represented by an ordered set consisting of the vertices of the convex hull. (For concreteness we may define the ordering as starting with the point with minimum x-coordinate and proceeding in clockwise order about the centroid of S.) To ease exposition, we will refer to this representation as the convex hull, and refer to the minimal convex set containing S as the *interior* of the convex hull.

Our approach builds upon the work of Overmars & Van Leeuwen [21]. Overmars & Van Leeuwen use a standard balanced BST T storing S to partition points along one axis, and likewise store the convex hull of T_v for each $v \in T$ in a balanced BST. In contrast, we use treaps in both cases, together with the hash table in [4] for memory allocation. Our main contribution is then to analyze the running times and space usage of this new uniquely represented version, and to show that even using only $O(\log n)$ random bits to hash and generate treap priorities, the expected time and space bounds match that of the original version up to constant factors. Specifically, we prove the following.

Theorem 6.1 *Let $n = |S|$. There exists a uniquely represented data structure for 2-D dynamic convex hull that supports point insertions and deletions in $O(\log^2 n)$ expected time, outputs the convex hull in $O(k)$ time, where k is the number of points in the convex hull, reports if a query point is in the convex hull or in its interior in $O(\log k)$ expected time, finds the tangents to the convex hull from an exterior query point in $O(\log k)$ expected time, and finds the intersection of the convex hull with a query line in $O(\log k)$ expected time. Furthermore, the data structure uses $O(n)$ space in expectation and requires only $O(\log n)$ random bits.*

Our Approach. We will discuss how to maintain only the *upper* convex hull, the lower convex hull is kept analogously. Let $U \subseteq \mathbb{R}^2$ be the universe of possible points, S be our set of points, and N be an upper bound on the number of points to be stored. We maintain a top level random treap T on the points, using an 11-wise independent hash function $h : U \to [N^3]$ to generate priorities, and using the natural ordering on the x-coordinates of the points as the key-ordering. That is, $(x, y) < (x', y')$ in the key ordering iff $x < x'$. (For simplicity, we will assume no two

points have the same x-coordinate, and that no three points are collinear.) Let $V[T_v]$ denote the points in T_v. Each node v stores a point $p \in S$ as well as the *convex hull* of $V[T_v]$. This convex hull is itself stored in a modified treap on $V[T_v]$, which we call H_v. Each treap in $\{H_v : v \in T\}$ obtains key priorities from the same 8-wise independent hash function $g : U \to [N^3]$, and they all use the same key-ordering as T. We will also maintain with each node u in each H_v pointers to its predecessor and successor in H_v according to the key ordering. Abusing notation slightly, we will call these pointers $\text{pred}(u)$ and $\text{succ}(u)$. Maintaining these pointers during updates is relatively straightforward, so we omit the details.

Insertions. Suppose we are currently storing point set S, and insert point p. First we identify the leaf position l that p would occupy in T if it had priority $-\infty$, and then rotate it up to its proper position (given its priority $h(p)$). We then must recompute H_v for all v in the path P from l to the root of T. For $v \in P$ that are ancestors of p, we need only add p to H_v as described below. For each $v \in P$ that is either p and one of its descendants, we must merge the convex hulls of v's children, and then add v to the result.

Adding a Point. We first consider adding a point u to H_v, assuming that u is not already in H_v. First, we can determine if u is in the upper convex hull of $V[T_v]$ in expected $O(\log |H_v|)$ as follows. Find the nodes $a := \max\{w : w < u\}$ and $b := \min\{w : w > u\}$, which takes expected $O(\log |H_v|)$ time. Then do a line side test to see if u is above or on line segment (a, b). Point u is in the hull if and only if u is above or on (a, b), otherwise not. If u is not in the hull, we leave H_v unchanged. If u is in the hull, we must find the points x and y such that the upper hull is $\{w \in H_v : w \leq x\} \cup \{u\} \cup \{w \in H_v : w \geq y\}$. Once these are found, we can split H_v at x and y, join the appropriate pieces, and add u to get the upper hull in $O(\log |H_v|)$ time.

We now discuss how to find x. Finding y is analogous. Let $\text{line}(p, q)$ denote the line containing points p and q. Given a point $w < u$, we can conclude that $x \leq w$ if u is above or on $\text{line}(w, \text{succ}(w))$. Additionally, we can conclude that $x \geq w$ if u is below $\text{line}(\text{pred}(w), w)$. Thus we can do a binary search for x by traversing the path from the root to x in H_v. Since we have the pointers to find $\text{succ}(\cdot)$ and $\text{pred}(\cdot)$ in constant time, it follows that we can find x in $O(\text{depth}(x))$ time, where $\text{depth}(x)$ is the depth of x in H_v. By Theorem 2.3, $\mathbf{E}[\text{depth}(x)] \leq 2\ln(|H_v|) + 1$, so adding a point takes expected $O(\log |H_v|)$ time.

The total time spent in adding points is $O(\log^2 n)$ in expectation. To see this, note that each addition takes $O(\log n)$ time in expectation, and there are at most the depth of l in T of them. Theorem 2.3 states that $\text{depth}(l)$ is $O(\log n)$ in expectation. Finally, $\text{depth}(l)$ is independent of the time taken for any point additions, since the former depends on h and the latter on g, so the expectation of their product is the product of their expectations.

Merging Two Upper Hulls. When rotating up the newly inserted point p in T, we must recompute H_v for each v involved in a rotation. We do this by *merging* the hulls of the children of v, say u and w, and then add v as described above. We can do this so that the expected time for all merges when adding a point to the top-level treap is $O(\log n)$. Our approach mimics that of [21].

Suppose we want to merge the hulls of the children of v, say u and w, and then add v as described above. We initially begin with H_u and H_w such that all of the points in the former are

smaller than all the points in the latter. We must find the *bridge* between them, that is, the pair of points (x, y) such that the upper hull of $V[T_u] \cup V[T_w]$ is $\{q \in H_u : q \leq x\} \cup \{q \in H_w : q \geq y\}$. Once we find x and y, two splits and a join immediately gives us the treap representation of the desired upper hull in $O(\log |V[T_v]|)$ expected time.

To find the bridge (x, y), we start with a guess $(x_0, y_0) = (\text{root}(H_u), \text{root}(H_w))$. We iteratively develop improved guesses (x_t, y_t), maintaining the invariant that x_t is an ancestor of x and y_t is an ancestor of y. In each step, we replace at least one of $\{x_t, y_t\}$ with one of its children to get (x_{t+1}, y_{t+1}), being careful to maintain the invariant. Clearly, after $\text{depth}(x) + \text{depth}(y)$ steps, we find the bridge.

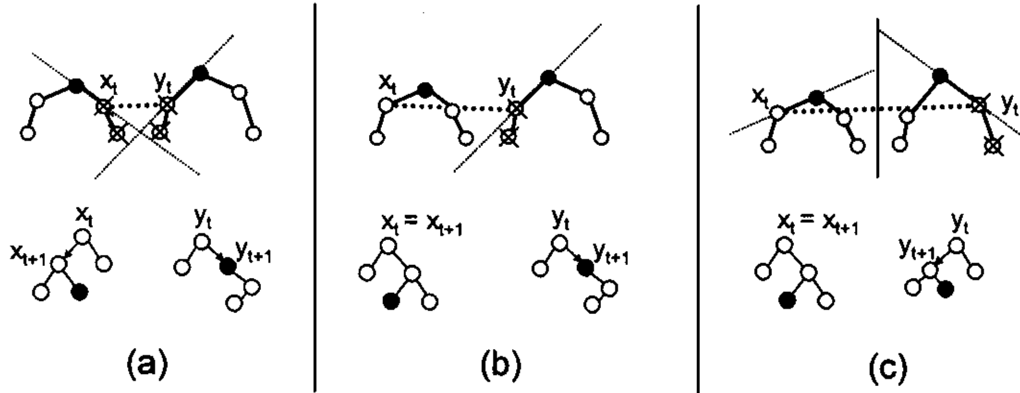

Figure 5: Finding the bridge. Both the convex hull and treaps storing them are displayed, with the bridge nodes in black. To test the current guess for the bridge (thick dotted line), we employ line-side tests. The thin dotted lines denote lines of interest in each of the three cases. Nodes marked "X" are discovered not to be in the bridge.

To compute (x_{t+1}, y_{t+1}) from (x_t, y_t), we first find $\text{pred}(x_t)$, $\text{pred}(y_t)$, $\text{succ}(x_t)$, and $\text{succ}(x_t)$. Then, line-side tests can be used to find an improving move. Figure 5 shows three of four cases, where the forth case is identical to case (b) if we swap the role of x_t and y_t. More specifically, we apply the following tests in order.

1. Test if $\text{line}(\text{pred}(x_t), x_t)$ is below y_t (or equivalently, if $\text{line}(\text{pred}(x_t), y_t)$ is above x_t). If so, we may infer that $x < x_t$ in the BST order, and thus we set x_{t+1} to be the left child of x_t. This is the case in Figure 5(a).

2. Test if $\text{line}(y_t, \text{succ}(y_t))$ is below x_t (or equivalently, if $\text{line}(x_t, \text{succ}(y_t))$ is above y_t). If so, we may infer that $y > y_t$ in the BST order, and thus we set y_{t+1} to be the right child of y_t. This is the case in Figures 5(a) and 5(b).

3. If neither of the above tests allowed us to make progress (i.e., $\text{line}(\text{pred}(x_t), x_t)$ is above y_t and $\text{line}(y_t, \text{succ}(y_t))$ is above x_t), then test if $\text{line}(x_t, y_t)$ is below either $\text{succ}(x_t)$ or $\text{pred}(y_t)$. If so, arbitrarily select a vertical line l such that all points in H_u are to the left of l and all points in H_w are to the right of it. Let z_0 be the intersection of $\text{line}(x_t, \text{succ}(x_t))$ and l, and let z_1 be the intersection of $\text{line}(\text{pred}(y_t), y_t)$ and l. If z_0 is above z_1, then we may

infer $x > x_t$ and set x_{t+1} to be the right child of x_t. Otherwise, we may infer $y < y_t$ and set y_{t+1} to be the left child of y_t, as is the case in Figure 5(c).

If no three points are collinear and none of these tests make any progress, then we may infer that we have found the bridge, i.e., $x = x_t$ and $y = y_t$. Though we omit the proof of this fact, Figure 6 illustrates some of the relevant cases, and is suggestive of how the proof goes. See [21] for further details.

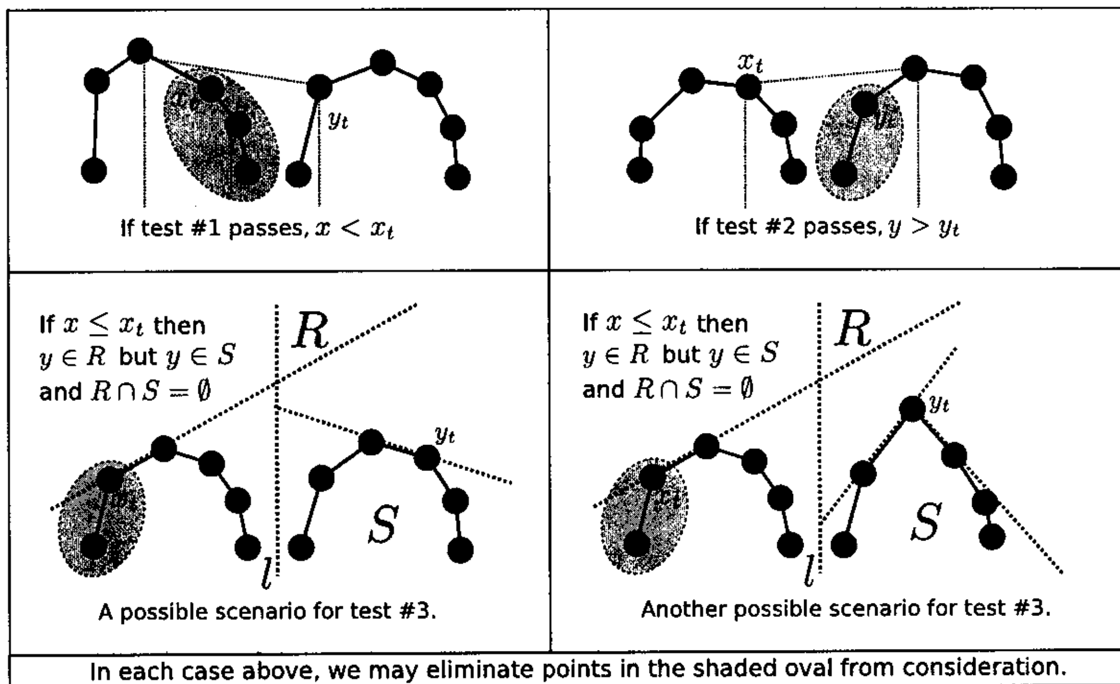

Figure 6: Illustrations of some of the cases encountered, with proof sketches.

In expectation, there are $\mathbf{E}[\text{depth}(x) + \text{depth}(y)] = O(\log |V[T_v]|)$ steps to find the bridge, and each step takes $O(1)$ time using the pointers for pred$(\cdot)$ and succ$(\cdot)$. Since treaps provide expected logarithmic time insertions even if tree rotations take linear time, and the cost to merge is independent of the top level treap's structure, the expected time for all merges when adding a point to the top-level treap is $O(\log n)$.

Deletions. To delete a point p, we rotate p down to its correct leaf position l (by treating it as though its priority were $-\infty$), then cut the leaf. We must then update H_v and H'_v for each v on the path from l to the root. Working our way up, and recomputing H_v and H'_v by merging their children's corresponding treaps, and then adding v itself, we can update all the necessary entries in $O(\log^2 n)$ expected time. The running time argument is similar to that for insertions, and we omit it here.

The Convex Hull Queries. To obtain the convex hull we simply do an in-order traversal of H_r, where r is the root of T. This clearly takes time linear in the size of the hull.

To determine if a query point p is in the convex hull, we simply search H_r for it, where r is the root of T. To determine if p is in the interior of the convex hull, find the predecessor u_{up} and successor v_{up} of p in the upper hull. Then find the predecessor u_{low} and successor v_{low} of p in the lower hull. Then p is in the interior of the convex hull if and only if $\text{line}(u_{\text{up}}, v_{\text{up}})$ is above p and $\text{line}(u_{\text{low}}, v_{\text{low}})$ is below p. Given random treaps containing the upper and lower convex hulls, both of these queries clearly take expected $O(\log k)$ time, where k is the number of points in the convex hull.

Finding the tangents to the convex hull from an exterior query point p involves searching the treap storing the convex hull, where lines side tests involving a candidate node u and its predecessor and successors allow one to determine whether a tangent point is in the left or right subtree of u in constant time. Determining the intersection of the convex hull with a query line is similar. Both queries can thus be done in expected $O(\log k)$ time. We omit the details on how to guide the searches, and instead refer the reader to [21].

Space Usage. The top level treap T clearly requires only $n-1$ pointers. Most of the space required is in storing $\{H_v : v \in T\}$. We store these treaps functionally, so storing H_v requires only $O(\log |V[T_v]|)$ pointers in expectation. Specifically, suppose v has children u and w. Given H_u and H_w, storing H_v requires storing a split path in each of H_u and H_w, and the insertion path for v. Each of these requires $O(\log |V[T_v]|)$ pointers in expectation [25]. Thus the total space is $O(\sum_v \log |V[T_v]|)$ in expectation, and we need to bound $\mathbf{E}[\log |V[T_v]|]$ for each v. By Lemma 6.2 $\mathbf{E}[\log |V[T_v]|] = O(1)$ for each v, so the total space usage is $O(n)$ in expectation.

Lemma 6.2 *A treap T on n nodes with priorities drawn from an 11-wise independent hash function $h : U \to \{0, 1, \ldots, r\}$ with $r \geq n^3$ satisfies $\mathbf{E}[\log |T_v|] = O(1)$. Furthermore, $\mathbf{Pr}[|T_v| = k] = O(1/k^2)$ for all $1 \leq k < n$ and $\mathbf{Pr}[|T_v| = n] = O(1/n)$.*

Proof: It is easy to prove that probability of a collision on priorities is at most $\binom{n}{2}/r < 1/2n$. In this case, we use the trivial bound $|T_v| \leq n$ to show that

$$\mathbf{E}[\log |T_v|] \leq \log(n)/2n + \mathbf{E}[\log(|T_v|) \mid \text{no collisions}].$$

So assume there are no collisions on priorities. Let S be the set of points in T. A useful lemma due to Mulmuley [18] implies that h satisfies the d-max property (see definition 6.3) if h is $(3d+2)$-wise independent, and thus h satisfies the d-max property for $d \leq 3$.

Now consider the probability that $|T_v| = k$ for some k. Relabel the nodes $\{1, 2, \ldots, n\}$ by their key order. Note that $|T_v| = k$ if and only if the maximal interval I containing v such that v has the maximum priority in I has size $|I| = k$. If $k = n$, then by the 1-max property $\mathbf{Pr}[|T_v| = n] = O(1/n)$. So suppose $k < n$, and further suppose $I = [a : b] := \{a, a+1, \ldots, b\}$. Then either $a - 1$ has priority higher than v, or $a - 1$ does not exist, and similarly with $b + 1$. If $a - 1$ does not exist and $|I| = k$, then $h(k+1) > h(v) > \max\{h(i) : i \in I \text{ and } i \neq v\}$, which occurs with probability $O(1/k^2)$ by the 2-max property. The case that $b + 1$ does not exist is analogous. If both $a - 1$ and $b + 1$ exist, then $\min\{h(a-1), h(b+1)\} > h(v) > \max\{h(i) | i \in I \text{ and } i \neq v\}$. By the 3-max property, this occurs with probably $O(1/k^3)$.

Taking a union bound over all intervals $I \subset [1:n]$ of length k that contain v, we obtain

$$\mathbf{Pr}[|T_v| = k] = O(1/k^2)$$

Thus $\mathbf{E}[\log |T_v|] = \sum_{k=1}^{n} \mathbf{Pr}[|T_v| = k] \cdot \log(k) = \sum_{k=1}^{n-1} O(\frac{\log k}{k^2}) + O(\frac{\log n}{n}) = O(1)$ ■

Definition 6.3 (d-max property) *A finite collection $\mathcal{X}$ of random variables has the d-max property iff there is some constant c such that for any subset of $\mathcal{X}$ and for any enumeration $X_1, X_2, \ldots, X_m$ of the elements of that subset, we have*

$$\mathbf{Pr}[X_1 > X_2 > \cdots > X_d > \max\{X_i \mid i > d\}] \leq c/m^{\underline{d}}$$

where $m^{\underline{d}} := \prod_{i=0}^{d-1}(m-i)$.

7 Conclusions

We have introduced uniquely represented data structures for a variety of problems in computational geometry. Such data structures represent every logical state by a unique machine state and reveal no history of previous operations, thus protecting the privacy of their users. For example, our uniquely represented range tree allows for efficient orthogonal range queries on a database containing sensitive information (e.g., viral load in the blood of hospital patients) without revealing any information about what order the current points were inserted into the database, whether points were previously deleted, or what queries were previously executed. Uniquely represented data structures have other benefits as well. They make equality testing particularly easy. They may also simplify the debugging of parallel processes by eliminating the conventional dependencies upon the specific sequence of operations that led to a particular logical state.

References

[1] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proc. SODA*, pages 531–540, 2004.

[2] A. Andersson and T. Ottmann. New tight bounds on uniquely represented dictionaries. *SIAM Journal of Computing*, 24(5):1091–1103, 1995.

[3] G. E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proc. SODA*, 2008.

[4] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *48th Annual IEEE Symposium on Foundations of Computer Science*, pages 272–282. IEEE, October 2007.

[5] G. E. Blelloch, D. Golovin, and V. Vassilevska. Uniquely represented data structures for computational geometry. In *Algorithm Theory - SWAT 2008, 11th Scandinavian Workshop on Algorithm Theory, Gothenburg, Sweden, July 2-4, 2008, Proceedings.*, Lecture Notes in Computer Science. Springer, 2008. to appear.

[6] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. FOCS*, pages 617–626, 2002.

[7] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In *Proc. CRYPTO*, pages 445–462, 2003.

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

[9] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry: algorithms and applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[10] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. STOC*, pages 365–372, 1987.

[11] P. F. Dietz. Fully persistent arrays. In *Proc. WADS*, pages 67–74, 1989.

[12] 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, 1994.

[13] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[14] D. Micciancio. Oblivious data structures: applications to cryptography. In *Proc. STOC*, pages 456–464, 1997.

[15] C. W. Mortensen. Fully-dynamic orthogonal range reporting on RAM. In *Technical report TR-2003-22 in IT University Technical Report Series*, 2003.

[16] C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proc. SODA*, pages 618–627, 2003.

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

[18] K. Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms.* Prentice-Hall, Englewood Cliffs, 1994.

[19] M. Naor, G. Segev, and U. Wieder. History-independent cuckoo hashing. In *ICALP*, page to appear, 2008.

[20] M. Naor and V. Teague. Anti-presistence: history independent data structures. In *Proc. STOC*, pages 492–501, 2001.

[21] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.

[22] A. Pagh, R. Pagh, and M. Ruzic. Linear probing with constant independence. In *Proc. STOC*, pages 318–327, 2007.

[23] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[24] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[25] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[26] S. Sen. Fractional cascading revisited. *Journal of Algorithms*, 19(2):161–172, 1995.

[27] L. Snyder. On uniquely representable data structures. In *Proc. FOCS*, pages 142–146, 1977.

[28] R. Sundar and R. E. Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In *Proc. STOC*, pages 18–25, 1990.

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