Uniquely Represented Data Structures for Computational Geometry

Guy E. Blelloch [⋆], Daniel Golovin [⋆⋆], and Virginia Vassilevska [⋆⋆⋆]

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA
{blelloch,dgolovin,virgi}@cs.cmu.edu

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 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

[⋆] Supported in part by NSF ITR grant CCR-0122581 (The Aladdin Center).

[⋆⋆] Supported in part by NSF ITR grants CCR-0122581 (The Aladdin Center) and IIS-0121678.

[⋆⋆⋆] Supported in part by NSF ITR grant CCR-0122581 (The Aladdin Center) and a Computer Science Department Ph.D. Scholarship.

that support these notions [14, 18, 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 collection 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 had also been studied earlier [24, 25, 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 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 [20] 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 random bits to be part of the input.

In this paper we use these and other results to develop various uniquely represented structures in computational geometry. 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 bounds. 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 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 [15, 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. Due to space considerations, the details of our results on horizontal point location and dynamic planar convex hull appear in the full version of the paper [5].

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), the *Turing 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. We use randomization solely to improve performance; in our performance guarantees we take probabilities and expectations over these random bits.

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, and so the proofs that our structures are UR are quite straightforward. We omit the details due to lack of space.

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 [22] throughout our presentation. We expect that one could also make use of skip lists [21] 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 1 (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 1 (Selected Treap Properties [22]). *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) $\mathbf{Pr}[|T_x| = k] = O(1/k^2)$ *for all $1 \leq k < n$*
(3) Given a predecessor handle, the expected insertion or deletion time is $O(1)$
(4) If the time to rotate a subtree of size k is $f(k)$ for some $f : \mathbb{N} \to \mathbb{R}_{\geq 1}$, 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 [26] 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 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 $\mathcal{S} = \{S_1, \ldots, S_q\}$ with $m = |L| + \sum_{i=1}^{q} |S_i|$ while supporting the OMP operations on L and the following ordered dictionary operations on each S_k:

- $\mathrm{Insert}(x, S_k)$: insert $x \in L$ into set S_k.
- $\mathrm{Delete}(x, S_k)$: delete x from S_k.
- $\mathrm{Pred}(x, S_k)$: 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. Mortensen [16] 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, where m is the total number of element occurrences in subsets. 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 [16], 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 2. *Let $m := |\{(x,k) : x \in S_k\}| + |L|$. There exists a UR data structure for the ordered subsets problem that uses $O(m)$ 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 2. To construct the data structure, we start with a UR *order maintenance* data structure on L, which we will denote by D (see Section 2). Whenever we are to compare two elements, we simply use D.

We recall an approach used in constructing D [4], ***treap partitioning***: Given a treap T and an element $x \in T$, let its *weight* $w(x,T)$ be the number of descendants, including itself. For a parameter s, let $\mathcal{L}_s[T] = \{x \in T : w(x,T) \geq s\} \cup \{\mathrm{root}(T)\}$ 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 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.

In the construction of D [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 $T(D)$. The set of weight s partition leaders of $T(D)$ is denoted by $\mathcal{L}[T(D)]$, and the treap on these leaders by $T(\mathcal{L}[D])$.

The other main structure that we use is a treap $\mathcal{T}$ containing all elements from the set $\hat{L} = \{(x,k) : x \in S_k\} \cup \{(x,0) : x \in \mathcal{L}[T(D)]\}$. Treap $\mathcal{T}$ is partitioned by weight $\log m$ partition leaders. These leaders are labeled with the path from the root to their node (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 $x \in \mathcal{L}[T(D)]$ to $(x,0) \in \mathcal{T}$.

We store each subset S_k in its own treap T_k, also partitioned by weight $\log m$ 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 $O(\log m)$-size subtree of T_k rooted at ℓ. 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. 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.

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

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

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 1. *[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 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 $x = x'$ 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 $\mathcal{L}[T(D)]$ changes we will need to update the treaps $\{J_v : v \in \mathcal{L}[T(D)]\}$, $\mathcal{T}$, 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 1 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 $T(\mathcal{L}[D])$ if the rotation of subtrees of size k costs $k + \log m$, which is $O(\log m)$ by Theorem 1. This bound holds because $|J_v| = O(\log m)$ for any v, and any tree rotation on $T(D)$ causes at most $3s$ elements of $T(D)$ to change their weight s leader. Therefore only $O(\log m)$ elements need to be added or deleted from the treaps $\{J_v : v \in T(\mathcal{L}[D])\}$, 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 $\mathcal{L}[T(D)]$ changes, which by Lemma 1 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 $\mathcal{L}[T(D)]$). 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 $T(D)$, and let y' be the predecessor of y in $\mathcal{L}[T(D)]$. 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 $\{J_v : v \in \mathcal{L}[T(D)]\}$, and the fast dictionaries $\{I_x : x \in L\}$.

Once we have found the predecessor w of (x, k) in $\mathcal{T}$, we search for the predecessor w' of x in $\mathcal{L}[T_k]$. (If w' does not exist, we simply use $\min\{u \in \mathcal{L}[T_k]\}$). To find w', we first use w to search for a node u', defined as the leader (x, k) would have had in $\mathcal{T}$, had it been given a priority of $-\infty$. Note that with priority $-\infty$, (x, k) would be the leftmost leaf of the right subtree of w in $\mathcal{T}$. Hence its leader would either be the leader of w, or the deepest leader on the leftmost path starting from the right child of w. Hence u' can be found in expected $O(\log \log m)$ time, by binary searching on its label (i.e., if the label of w is α, then find the maximum k such that $\alpha \cdot 1 \cdot 0^k$ is an label in H).

Let P be the path from u' to the root of $\mathcal{T}$. We use the label of u' and H to binary search on P for the deepest node $v \in P$ for which $\min\{u : u \in \mathcal{L}[T_k] \text{ and } (u, k) \in \mathcal{T}_v\} < x$. This takes $O(\log |P|) = O(\log \log m)$ time in expectation. If $v \neq u'$, then u' is in the right subtree of v in $\mathcal{T}$, and (w', k) is in the left subtree of v. So let v_l be the left child of v and note that $w' = \max\{u \ : \ u \in \mathcal{L}[T_k] \text{ and } (u, k) \in \mathcal{T}_{v_l}\}$, which we can look up in $O(1)$ time after finding v by using H_v. Otherwise $v = u'$. In this case, lookup $a := \min\{u : u \in \mathcal{L}[T_k] \text{ and } (u, k) \in \mathcal{T}_v\}$ and $b := \max\{u : u \in \mathcal{L}[T_k] \text{ and } (u, k) \in \mathcal{T}_v\}$, find the least common ancestor c of $\{a, b\}$ in T_k, and starting from c search T_k for w'. Since a and b are both descendants of u', their distance (i.e., one plus the number of nodes between them in the order) in $\hat{L}$ is at most $s = \Theta(\log m)$, and thus their distance in T_k is at most $O(\log m)$. However, in random treaps the expected length of a path between nodes at distance d is $O(\log(d))$, even if priorities are generated using only 8-wise independent hash functions [22]. Thus we can find c in expected $O(\log \log m)$ time. Note c has at most $O(\log^2 m)$ descendants between a and b in T_k, since there are at most $O(\log m)$ partition leaders between a and b and each has at most $O(\log m)$ "followers" in its partition set, and we can find w' in expected $O(\log \log m)$ time starting from c. Once we have found w', the predecessor of x in $\mathcal{L}[T_k]$, we can simply find the successor of w' in $\mathcal{L}[T_k]$, say w'', via fast finger search, and then search the subtreaps rooted at w' and w'' for the actual predecessor of x in S_k in expected $O(\log \log m)$ time.

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 $T(D)$, 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. 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 $O(|\Delta_k| \log m)$ time. Moreover, $\mathbf{E}\left[|\Delta_k| \mid x \in \mathcal{L}[T_k]'\right] = O(1)$, 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 $\mathcal{L}[T_k]'$. Since it takes $\Theta(R)$ time to delete x given a handle to it, from Theorem 1 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 $\mathcal{T}$.

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 3. *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 $O(n \log^{d-1} n)$ 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 describe

[2] We expect a variant of Sen's approach [23] could work.

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. The description will be deferred to the full paper. 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 1, 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 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.

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). 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.

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 $x \leq \hat{x} \leq x'$ and $y < \hat{y}$. In the related *orthog-*

onal segment intersection problem we are given a vertical line segment $s = (x, y, y')$, and must report all segments in S intersecting it, namely $\{(x_i, x'_i, y_i) : x_i \leq x \leq x'_i$ and $y \leq y_i \leq y'\}$. 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 [15] 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 4. *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.*

6 Uniquely Represented 2-D Dynamic Convex Hull

Using similar techniques we obtain a uniquely represented data structure for maintaining the convex hull of a dynamic set of points $S \subset \mathbb{R}^2$. Our approach builds upon the work of Overmars & Van Leeuwen [19]. 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 5. *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 size of the convex hull, requires $O(n)$ space in expectation, and uses only $O(\log n)$ random bits.*

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 *Proc. FOCS*, pages 272–282, 2007.
5. G. E. Blelloch, D. Golovin, and V. Vassilevska. Uniquely represented data structures for computational geometry. Technical Report CMU-CS-08-115, Carnegie Mellon University, April 2008.
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. Paul 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 two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proc. SODA*, pages 618–627, 2003.
16. C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Technical report TR-2003-22 in IT University Technical Report Series*, 2003.
17. C. W. Mortensen. Fully dynamic orthogonal range reporting on a RAM. *SIAM J. Comput.*, 35(6):1494–1525, 2006.
18. M. Naor and V. Teague. Anti-presistence: history independent data structures. In *Proc. STOC*, pages 492–501, 2001.
19. M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.
20. A. Pagh, R. Pagh, and M. Ruzic. Linear probing with constant independence. In *Proc. STOC*, pages 318–327, 2007.
21. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Proc. WADS*, pages 437–449, 1989.
22. R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.
23. S. Sen. Fractional cascading revisited. *Journal of Algorithms*, 19(2):161–172, 1995.
24. L. Snyder. On uniquely representable data structures. In *Proc. FOCS*, pages 142–146, 1977.
25. 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.
26. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.