Persistent Predecessor Search and Orthogonal Point Location on the Word RAM

Timothy M. Chan*

July 7, 2010

Abstract

We answer a basic data structuring question (for example, raised by Dietz and Raman back in SODA 1991): can van Emde Boas trees be made persistent, without changing their asymptotic query/update time? We present a (partially) persistent data structure that supports predecessor search in a set of integers in $\{1, \ldots, U\}$ under an arbitrary sequence of n insertions and deletions, with $O(\log \log U)$ expected query time and expected amortized update time, and $O(n)$ space. The query bound is optimal in U for linear-space structures and improves previous near-$O((\log \log U)^2)$ methods.

The same method solves a fundamental problem from computational geometry: point location in orthogonal planar subdivisions (where edges are vertical or horizontal). We obtain the first static data structure achieving $O(\log \log U)$ worst-case query time and linear space. This result is again optimal in U for linear-space structures and improves the previous $O((\log \log U)^2)$ method by de Berg, Snoeyink, and van Kreveld (1992). The same result also holds for higher-dimensional subdivisions that are orthogonal binary space partitions, and for certain nonorthogonal planar subdivisions such as triangulations without small angles. Many geometric applications follow, including improved query times for orthogonal range reporting for dimensions ≥ 3 on the RAM.

Our key technique is an interesting new van-Emde-Boas–style recursion that alternates between two strategies, both quite simple.

1 Introduction

Van Emde Boas trees [32, 33, 60] are fundamental data structures that support *predecessor searches* in $O(\log \log U)$ time on the word RAM with $O(n)$ space, when the n elements of the given set S come from a bounded integer universe $\{1, \ldots, U\}$ ($U \geq n$). In a predecessor search, we seek the largest element in S which is smaller than a given value not necessarily in S. The word size w is assumed to be at least $\log U$ so that each element fits in a word. The key assumption is not so much about input elements being integers but that they have bounded precision, as in "real life" (for example, floating-point numbers with w_1-bit mantissa and w_2-bit exponents can be mapped to integers with $U = O(2^{w_1+w_2})$ for the purpose of comparisons). In terms of U, the $O(\log \log U)$ time bound is known to be tight for predecessor search for any data structure with $O(n \operatorname{polylog} n)$ space [10, 54]. Van Emde Boas trees can also be dynamized to support insertions and deletions in

*Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca). Work supported by NSERC.

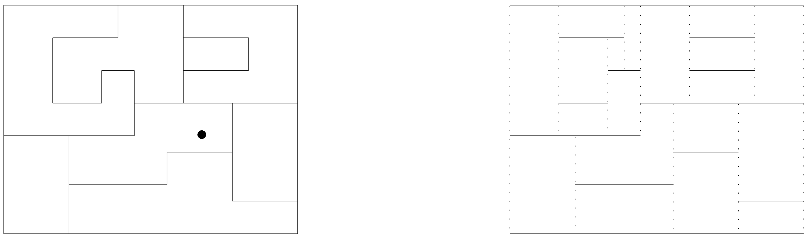


Figure 1: Left: The point location problem—or, where are we (the black dot)? Right: the vertical decomposition of a set of horizontal line segments.

$O(\log\log U)$ expected time with randomization. The importance of van Emde Boas trees is reflected, for instance, by its inclusion in the latest edition of a popular textbook [24].

In this paper, we address some basic questions about such trees. Very roughly put: is there a persistent analog of van Emde Boas trees? is there a two-dimensional equivalent of van Emde Boas trees?

Persistent predecessor search. Persistence has been a frequently studied topic in data structures since the seminal papers by Sleator and Tarjan [57] and Driscoll *et al.* [30]. A *partially persistent* data structure is one that supports updates and allows queries to be made to past versions of the data structure. (In this paper we will not consider *fully* persistent data structures, which also allow updates to be done to past versions, resulting in a tree of different versions. Despite the mildness of the name, "partial" persistence is actually sufficient for most of the interesting applications, for instance, in computational geometry.)

The archetypal example of a persistent data structure is Sleator and Tarjan's, which maintains a dynamic set S under n insertions and deletions in $O(\log n)$ time and supports predecessor search in S at any previous time value in $O(\log n)$ time. Many other results on persistent data structures have subsequently been published, for example, on union-find, deques, ... [46].

Here, we consider the possibility of making van Emde Boas trees persistent, which has remained unresolved to this day. The underlying problem is the same as in the original paper by Sleator and Tarjan, except we work in a bounded universe. Known techniques can give around $O((\log\log U)^2)$ query and update time, and the challenge is to obtain an $O(\log\log U)$ bound. This problem was explicitly mentioned, for instance, by Dietz and Raman [27] at SODA exactly two decades ago (and might have dated back even further).

We solve this open problem: we provide an optimal $O(n)$-space data structure which supports insertions and deletions in $O(\log\log U)$ expected amortized time and predecessor search at any previous time value in $O(\log\log U)$ expected time.

Orthogonal planar point location. A closely related problem is *orthogonal* (or *rectilinear*) *point location* in 2-d, which can be viewed as a geometric generalization of 1-d predecessor search: given a planar subdivision of size $O(n)$ where all the edges are vertical or horizontal, build a data structure so that we can quickly find the region containing any given query point. By constructing a *vertical decomposition* of the horizontal edges (see Figure 1), the problem reduces to point location in a collection of $O(n)$ interior-disjoint axis-aligned rectangles in the plane. This in turn is equivalent to answering *vertical ray shooting* queries among $O(n)$ horizontal line segments. By viewing x-

coordinate as time, we can further see that the geometric problem is identical to (partially) persistent predecessor search in the *offline* case where all the updates are known in advance (or where all queries occur after all updates).

Planar point location is one of the central problems in computational geometry. Many different methods with $O(\log n)$ query time and $O(n)$ space have been proposed in the literature [14, 55, 58]. On the word RAM, Chan and Pătraşcu [19] were the first to obtain sublogarithmic query time $O(\min\{\log n/\log\log n, \sqrt{\log U/\log\log U}\})$ with linear space for arbitrary, nonorthogonal planar subdivisions with coordinates from $\{1,\ldots,U\}$. For the orthogonal case, de Berg, van Kreveld, and Snoeyink [15] have earlier obtained an $O(\log\log U)^2)$ query bound with linear space.

Our result immediately implies the first optimal $O(n)$-space data structure for orthogonal planar point location achieving $O(\log\log U)$ query time. The query bound here can be made worst-case rather than expected. We also get optimal $O(n)$ preprocessing time, assuming that the given subdivision is connected.

The $(\log\log)^2$ **barrier.** There are actually two different approaches to obtaining the previous $O((\log\log U)^2)$ bound:

For orthogonal 2-d point location, one approach is to use a "two-level" data structure, where we build a van Emde Boas tree for the y-values, and nodes of the global tree store van Emde Boas trees for x-values. De Berg *et al.* [15] followed this approach. More generally, Dietz [26] has shown how any data structure, abstracted in the form of an array, can be made (fully) persistent by adding an extra level of van Emde Boas trees storing time values, with a $\log\log$ factor slow-down. (Driscoll *et al.* [30] has described another general transformation without the slow-down, but under a bounded fan-out assumption that does not hold for van Emde Boas trees.)

A second approach starts with the observation that $O(\log\log U)$ query time is easy if space or update time is not a consideration. For example, for orthogonal 2-d point location, the trivial method of dividing into $O(n)$ vertical slabs enables queries to be answered by a search in x followed by a search in y, using total space $O(n^2)$; for persistent predecessor search, the trivial method of building a new data structure after every update has $O(n)$ update time. We can then apply known techniques to lower space (or update time in the persistent problem), for example, via sampling, separators, or exponential search trees, as described by Chan and Pătraşcu [19]. With a constant number of rounds of reduction, the space usage can be brought down to $O(n^{1+\varepsilon})$. However, to get below $O(n \operatorname{polylog} n)$ space, $O(\log\log n)$ rounds are needed. As a result, the query time becomes $O(\log\log n \log\log U) \leq O((\log\log U)^2)$, as before. (It is not important to distinguish between $\log\log n$ and $\log\log U$, since for orthogonal point location at least, any query bound $t(n,U)$ can be converted to $O(\log\log U + t(n,n))$ by a simple normalization trick.)

A very small improvement is possible if we start with Beame and Fich's data structure for predecessor search [10], which for polynomial space has $O(\log\log U/\log\log\log U)$ query time. This gives an overall query bound of $O(\log\log n \log\log U/\log\log\log U)$ (see Section 3.3).

Note that if we start with the other known predecessor-search bounds $O(\log_w n)$ and $O(\sqrt{\log n/\log\log n})$ attained by *fusion trees* and *exponential search trees* [8, 9, 10, 37, 54], the repeated rounds of space reduction do *not* cause a $\log\log n$-factor blow-up in the query time, because these functions grow sufficiently rapidly in n (see Section 3.3). The van Emde Boas bound is thus the chief remaining question concerning orthogonal point location and persistent predecessor search.

Because the two approaches fail to give anything substantially better than $O((\log\log U)^2)$, our

$O(\log\log U)$ result may come as a bit of surprise—the 2-d orthogonal point location problem has the same complexity as in 1-d! (Frankly, the author would have been satisfied with any result that beats $O((\log\log U)^2/\log\log\log U)$.) Before, $O(\log\log U)$ bounds were known in limited special cases only: Dietz and Raman [27] considered persistent predecessor search in the *insertion-only* or *deletion-only* case, with the further restriction $U = n$; Iacono and Langerman [44] studied orthogonal point location among *fat* (hyper)rectangles; and a number of researchers [7, 16] investigated *approximate* nearest neighbor search in constant dimensions, though this approximate problem turns out to be directly reducible to 1-d [17].

Although many applications and variations of van Emde Boas trees have appeared in the literature, our $\log\log$ result requires a truly new generalization. Still, our final solution to orthogonal 2-d point location is simple, to the degree of being implementable, and is based on an unexpected synergy between two different recursive strategies (see Section 2). Our solution to persistent predecessor search builds on these strategies further (see Section 3), but is more theoretical: it invokes fusion trees or atomic heaps [37, 38] at some point, and requires additional new ideas concerning *monotone list labeling*, possibly of independent interest.

Applications. Orthogonal planar point location is a very fundamental problem in computational geometry, and our new result has numerous immediate consequences (see Section 4). For example, we obtain the current-record upper bounds on the query time for *orthogonal range reporting*—a standard problem studied since the dawn of computational geometry—among data structures with $O(n \operatorname{polylog} n)$ space on the word RAM, for all constant dimensions $d \geq 3$.

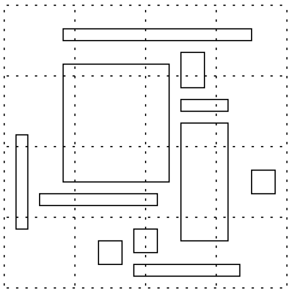
Some of the corollaries are of interest even to traditionalists not accustomed to sublogarithmic word-RAM results. For example, we obtain the first data structures with $O(\log n)$ query time and $O(n \operatorname{polylog} n)$ space, for rectangle containment queries in 2-d (for a set of possibly intersecting rectangles, finding a rectangle contained in a given query rectangle) and vertical ray shooting queries for a set of horizontal rectangles in 3-d. (Any word-RAM algorithm with $t(n, U)$ query time can be converted to a real-RAM algorithm with $O(\log n + t(n, n))$ query time in a model where we are permitted to inspect the input coordinates only through comparisons, in addition to usual $(\log n)$-bit RAM operations.)

Though important, the orthogonal case may seem a bit restrictive at first glance. However, our result actually implies $O(\log\log U)$ query time for point location in certain planar nonorthogonal subdivisions, such as subdivisions with a constant number of different edge directions, or triangulations satisfying a fatness assumption, which is reasonable in practice. We are also able to extend our point location result to higher dimensions in the case of binary space partitions (BSPs), which include k-d trees and quadtrees (possibly unbalanced) as special cases.

2 Orthogonal Planar Point Location

We first describe our technique for the orthogonal planar point location problem, which as mentioned is equivalent to persistent predecessor search in the case of offline updates. We could have chosen to explain our algorithm entirely in the persistent 1-d setting, but the 2-d setting has visual appeal and is particularly nice to work with because of the underlying symmetry of the problem (w.r.t. x- vs. y-coordinates, i.e., time vs. elements' values).

We work with the following formulation of the problem: The input is a set S of n disjoint axis-aligned rectangles with integer coordinates inside a rectangular range which, by translation, is



h/v	h	h	h/v
v	c	v	h/v
v	h	v	v
h/v	h/v	h	h/v

Figure 2: The table T ("c", "v", and "h" stand for "covered", "vertical", and "horizontal" respectively).

assumed to be $[0, W) \times [0, H)$. (Initially, $W = H = U$.) We want a data structure that can find the rectangle in S containing a query point q (if the rectangle exists).

2.1 First idea: range reduction via grids/tables

We use recursion to solve the problem. Our first recursive strategy is based on a simple idea—build a uniform grid. Although this may sound commonplace (for example, grid-based approaches have been used in some data structures for orthogonal range searching), the way we apply the idea to point location is original, to the author's knowledge.

Specifically, we use a grid with $\lceil W/K \rceil$ rows and $\lceil H/L \rceil$ columns for some parameters K and L. *Column* i refers to the range $[iK, (i+1)K) \times [0, H)$, *row* j refers to $[0, W) \times [jL, (j+1)L)$, and *grid cell* (i, j) refers to $[iK, (i+1)K) \times [jL, (j+1)L)$. We say that a line segment s *cuts through* a cell if s intersects the cell but both endpoints of s are outside the cell.

The data structure:

- Store a table T (see Figure 2), where for each grid cell (i, j),
  - if the cell is contained in some rectangle $s \in S$, then $T[i, j] =$ "covered by s";
  - else if some vertical edge in S cuts through the cell, then $T[i, j] =$ "vertical";
  - else if some horizontal edge in S cuts through the cell, then $T[i, j] =$ "horizontal";
  - else $T[i, j]$ is set to either "horizontal" or "vertical" arbitrarily.
- For each column i, take the subset of all rectangles in S that have a vertex inside the column, and recursively build a data structure for these rectangles clipped to the column.
- For each row j, take the subset of all rectangles in S that have a vertex inside the row, and recursively build a data structure for these rectangles clipped to the row.

Note that each rectangle is stored in at most 4 of these substructures (2 column and 2 row data structures). We choose $K = \lceil W/\sqrt{n} \rceil$ and $L = \lceil H/\sqrt{n} \rceil$ so that the number of grid cells, and thus the table size, are $O(n)$.

The query algorithm: Given query point $q = (x, y)$,

1. let $i = \lfloor x/K \rfloor$, $j = \lfloor y/L \rfloor$;
2. if $T[i,j] =$ "covered by s", then return s;
3. if $T[i,j] =$ "vertical", then recursively query the data structure for column i;
4. if $T[i,j] =$ "horizontal", then recursively query the data structure for row j.

Correctness is easy to see: for example, if $T[i,j] =$ "vertical", then no horizontal edges can cut through the cell (i,j), by disjointness of the rectangles, so the rectangle containing q must have a vertex inside column i.

The space and query time of the above data structure clearly satisfy the recurrences

$$\begin{aligned} S(n,W,H) &\leq \sum_i S(m_i, \lceil W/\sqrt{n} \rceil, H) \,+\, \sum_j S(n_j, W, \lceil H/\sqrt{n} \rceil) \,+\, O(n) \\ Q(n,W,H) &\leq \max\left\{ \max_i Q(m_i, \lceil W/\sqrt{n} \rceil, H),\, \max_j Q(n_j, W, \lceil H/\sqrt{n} \rceil) \right\} \,+\, O(1) \end{aligned} \qquad (1)$$

for some m_i's and n_j's with $\sum_i m_i + \sum_j n_j \leq 4n$ (where the number of terms is $O(\sqrt{n})$).

Remarks: A more careful accounting could perhaps reduce the factor-4 blow-up in space at each level of the recursion, but this will not matter at the end as revealed in Section 2.4 (besides, there will be a similar blow-up in the strategy in Section 2.2 that is not as easy to avoid).

Initially, by normalization ("rank space reduction"), we can make $W = H = n$ and the recursion allows us to reduce one of W or H greatly, to $\sqrt{n}$, in just the first iteration. However, as n gets small (relative to W and H), progress gets worse with the above recurrences. We could re-normalize every so often, to close the gap from n to W and H, but each such re-normalization requires 1-d predecessor search (van Emde Boas trees) which would cost an extra $\log\log$ factor in the query time.

In the next subsection, we propose another strategy to bring W and H closer to n in just constant time, which can be viewed as a generalization of one round of a van Emde Boas recursion.

2.2 Second idea: range reduction via hashing

We consider a different recursive strategy to reduce the range of the y-coordinates. Following van Emde Boas, we could divide the y-range $[0,H)$ into about $\sqrt{H}$ rows each of height $\sqrt{H}$, build a "top" data structure treating the nonempty rows as elements recursively, and a "bottom" data structure for the elements within each nonempty row. However, this approach does not work for our problem, because an element can "participate" in multiple rows (i.e., a rectangle can have its vertical edges cutting through multiple rows). Our new idea is to use a single bottom data structure rather than multiple ones, obtained via a "collapsing" operation. This, however, results in a recurrence that deviates somewhat from van Emde Boas'.

Specifically, we use a grid with $\lceil H/L \rceil$ rows for some parameter L, where *row j* refers to the range $[0,W) \times [jL, (j+1)L)$.

The data structure:

- Create a dictionary D storing the indices of all columns that contain vertices in S, i.e., D contains $\lfloor y/L \rfloor$ for all y-coordinates y of the vertices of the rectangles. Sort D and record the rank $f[z]$ of each element z in D.

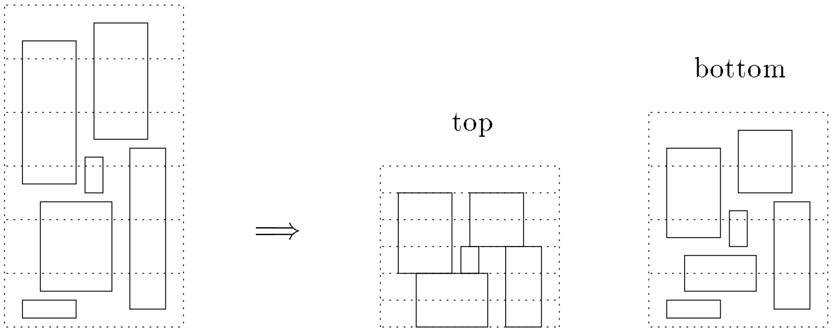


Figure 3: The top and bottom data structures.

- "Round" each y-coordinate to its column index; in other words, map each rectangle $[x_1, x_2) \times [y_1, y_2)$ in S to $[x_1, x_2) \times [\lfloor y_1/L \rfloor, \lfloor y_2/L \rfloor)$. Recursively build a *top* data structure for these mapped rectangles. (See Figure 3. Naturally, an identifier to the original rectangle is kept for each mapped rectangle.)

- "Collapse" all columns that do not contain vertices; in other words, map each rectangle $[x_1, x_2) \times [y_1, y_2)$ in S to $[x_1, x_2) \times [f[\lfloor y_1/L \rfloor]L + (y_1 \bmod L), f[\lfloor y_2/L \rfloor]L + (y_2 \bmod L))$. Recursively build a *bottom* data structure for these mapped rectangles. (See Figure 3.)

Note that the rectangular range in the top data structure has width $\lceil H/L \rceil$, whereas the range in the bottom structure has width at most nL. We choose $L = \left\lceil \sqrt{H/n} \right\rceil$ so that the widths in both structures are at most $\sqrt{nH} + n$.

The query algorithm: Given query point (x, y),

1. if $\lfloor y/L \rfloor \notin D$, then recursively query the top data structure for the point $(x, \lfloor y/L \rfloor)$;

2. if $\lfloor y/L \rfloor \in D$, then recursively query the bottom data structure for the point $(x, f[\lfloor y/L \rfloor]L + (y \bmod L))$.

Correctness is easy to see. Note that membership in D (and lookup of f) takes $O(1)$ time by perfect hashing.

The space and query time of the above data structure clearly satisfy the following recurrences:

$$\begin{aligned} S(n, W, H) &\leq 2\,S(n, W, \sqrt{nH} + n) + O(n) \\ Q(n, W, H) &\leq Q(n, W, \sqrt{nH} + n) + O(1). \end{aligned} \tag{2}$$

By a symmetric approach for x-coordinates, we can obtain the following alternate recurrences:

$$\begin{aligned} S(n, W, H) &\leq 2\,S(n, \sqrt{nW} + n, H) + O(n) \\ Q(n, W, H) &\leq Q(n, \sqrt{nW} + n, H) + O(1). \end{aligned} \tag{3}$$

Remarks: These recursions can lower W and H very rapidly, requiring only $\log\log$ iterations to converge. However, unlike in the original van Emde Boas recursion, W and H converge to $O(n)$, not $O(1)$, so this strategy by itself would not solve the problem.

2.3 Combining

Although neither of the two recursive strategies performs well enough alone, they complement each other surprisingly perfectly—the first makes good progress when n is large, whereas the second makes good progress when n is small (relative to W or H). A simple combination of the two turns out to be sufficient to yield $\log\log$ query time.

Several ways to combine are viable. We suggest the following:

- if $n > \sqrt{W}, \sqrt{H}$, adopt the first strategy, using (1);
- if $n \leq \sqrt{H}$, adopt the second strategy, using (2);
- if $n \leq \sqrt{W}$, adopt the second strategy, using (3).

The choice of thresholds comes from balancing the expressions $W/\sqrt{n}$ against $\sqrt{nW}$, and $H/\sqrt{n}$ against $\sqrt{nH}$. In each of these cases, we get

$$\begin{aligned} S(n, W, H) &\leq \sum_i S(m_i, O(W^{3/4}), H) + \sum_j S(n_j, W, O(H^{3/4})) + O(n) \\ Q(n, W, H) &\leq \max\left\{\max_i Q(m_i, O(W^{3/4}), H),\ \max_j Q(n_j, W, O(H^{3/4}))\right\} + O(1) \end{aligned} \tag{4}$$

for some m_i's and n_j's with $\sum_i m_i + \sum_j n_j \leq 4n$ (in the case $n \leq \sqrt{H}$ or $n \leq \sqrt{W}$, the number of terms degenerates to 2, and the constant factor 4 reduces to 2).

The base case, where W or H drops below a constant, can be handled directly by a 1-d data structure. Solving the recurrences, we immediately obtain $Q(n, U, U) \leq O(\log\log U)$, and $S(n, U, U) \leq O(4^{2\log_{4/3}\log U} n) \leq O(n \log^{10} U)$.

2.4 Last step: lowering space via separators

It remains to eliminate the extra polylogarithmic factor from the space bound. This step is standard (e.g., in 1-d van Emde Boas trees, it is analogous to the final transformation from "x-fast" to "y-fast tries" [60]). Chan and Pătraşcu [19] have described three general approaches to lowering space for planar point location: random sampling, planar graph separators, and an approach based on 1-d exponential search trees. De Berg *et al.* [15] have specifically employed the graph separator approach for orthogonal planar point location. For the sake of completeness, we quickly re-sketch the separator-based approach, which has the theoretical advantage of achieving good deterministic preprocessing time (the sampling-based approach makes for a more practical option and will be discussed later in Section 3.3).

It is more convenient now to consider the point location problem in the vertical decomposition $\mathcal{V}$ of a given subdivision. By applying a multiple-components version of the planar separator theorem [36] to the dual of $\mathcal{V}$, there exist r *separator* cells of $\mathcal{V}$ whose removal leaves connected components each with at most $O((n/r)^2)$ cells.[1] Build a point-location data structure for the subdivision induced by the separator cells, with $S_0(r)$ space and $Q_0(r)$ query time. For each component γ_i with n_i cells, build a point-location data structure, with $S_1(n_i)$ space and $Q_1(n_i)$ query time. This leads to a data

[1] Equivalently, $O(\sqrt{kn})$ separator cells suffice to yield components of size $O(n/k)$.

structure with the following space and query bound:

$$\begin{aligned} S(n) &\leq S_0(r) + \sum_i S_1(n_i) + O(n) \\ Q(n) &\leq Q_0(r) + \max_i Q_1(n_i) + O(1) \end{aligned} \tag{5}$$

where $\sum_i n_i \leq n$ and $n_i \leq O((n/r)^2)$.

We can use our data structure in Section 2.3 with $S_0(r) = O(r \log^{10} U)$ and $Q_0(r) = O(\log\log U)$, along with a classical point-location data structure with $S_1(n_i) = O(n_i)$ and $Q_1(n_i) = O(\log n_i)$. Choosing $r = \lfloor n/\log^{10} U \rfloor$, we obtain a solution with $S(n) = O(n)$ space, and $Q(n) = O(\log\log U)$ query time.

2.5 Preprocessing time

We analyze the time required to build the data structure.

In Sections 2.1 and 2.2, it can be checked that the preprocessing time satisfies the same recurrence as space, up to at most a logarithmic factor increase: In Section 2.1, we can easily set all the "covered" entries of the table T in $O(n)$ time (since the rectangles are disjoint and there are $O(n)$ table entries). To set the "vertical" entries, we can take each column i and find the "interval" of grid cells that are cut through by each vertical edge inside column i, and compute the union of these intervals. The total cost of solving these union-of-intervals subproblems is at most $O(n \log n)$ (or $O(n)$ if one is careful). The "horizontal" entries can be set similarly. In Section 2.2, hashing requires linear expected preprocessing time or $O(n \log n)$ deterministic time [40].

The data structure in Section 2.3 therefore has preprocessing time $T_0(n) \leq O(n \log^{11} U)$.

In Section 2.4, the separator can be constructed in linear time [4, 39]. The classical point-location data structure by Kirkpatrick [48] has $T_1(n_i) = O(n_i)$ preprocessing time. The total preprocessing time is

$$T(n) \leq T_0(r) + \sum_i T_1(n_i) + O(n). \tag{6}$$

Resetting $r = \lfloor n/\log^{11} U \rfloor$ gives $T(n) = O(n)$.

We have assumed that a vertical decomposition of the subdivision is given. If not, it can be constructed in linear time by Chazelle's algorithm [22] if the subdivision is connected, and $O(n \log\log U)$ time otherwise by a sweep-line algorithm using 1-d dynamic van Emde Boas trees. The $O(n \log\log U)$ term can be reduced to $O(n \log\log n)$, by using a word-RAM sorting algorithm [41] first to normalize coordinates to $\{1, \ldots, n\}$.

Theorem 2.1 *Given an orthogonal subdivision of size n whose coordinates are from $\{1, \ldots, U\}$, we can build an $O(n)$-space data structure so that point location queries can be answered in $O(\log\log U)$ time. The preprocessing time is $O(n)$ if the subdivision is connected, and $O(n \log\log n)$ otherwise.*

Remark: Our query algorithm requires only standard word operations, namely, additions, multiplications, and shifts. In fact, we can bypass multiplications in the first strategy by rounding the parameters K and L to make them powers of 2, although we cannot avoid non-AC^0 operations in the second strategy due to hashing.

3 Persistent Predecessor Search

In this section, we extend our method to solve the (partially) persistent predecessor search problem. Following Section 2, our description will be in notation and terminology from the 2-d geometric setting. It is not difficult to see that persistent 1-d predecessor search can be reduced to the following problem: We want a data structure to maintain a set S of at most n disjoint rectangles with integer coordinates from the range $[0, W) \times ([0, H) \cup \{\infty\})$. In a query, we want to find the rectangle in S containing a given point, just as before. In addition, the data structure should support the following restricted update operations:

- *open*: insert a rectangle s of the form $[t, \infty) \times [y_1, y_2)$ to S, under the assumption that t is at least as large as all finite x-coordinates currently in S;
- *close*: replace a rectangle s of the form $[t', \infty) \times [y_1, y_2)$ with $[t', t] \times [y_1, y_2)$ in S, again under the assumption that t is at least as large as all finite x-coordinates currently in S.

Note that over the entire sequence of open/close operations, the value of t must be monotone increasing and thus can be thought of as time. Let t^* denote the current time, i.e., the value of t from the most recent update operation in S.

To see how persistent 1-d predecessor search reduces to the above, when an element y is inserted (resp. deleted), we can use a standard (ephemeral) dynamic van Emde Boas tree to find its current predecessor y^- and successor y^+ in $O(\log\log U)$ time, and then close (resp. open) a rectangle with y-range $[y^-, y^+)$ and open (resp. close) two rectangles with y-ranges $[y^-, y)$ and $[y, y^+)$ at current time.

We assume that n is known in advance, but this assumption can be eliminated by a standard doubling trick: when the number of rectangles exceeds n, we double n and rebuild the data structure by re-processing the entire update sequence from scratch. The total update time increases by a constant factor: more precisely, if the original total update time is $T(n)$, the new total update time is $O(\sum_i T(n_i))$ for some n_i's with $\sum_i n_i = O(n)$ by a geometric series.

3.1 First idea, modified

The grid strategy as described in Section 2.1 does not directly support open/close operations on-line. One issue lies in the column data structures—when t^* passes through iK, we do not know in advance which rectangles will be closed with their right vertical edges in column i. We change the definition of column data structures, which in turn leads to a complete change in the definition of the table:

- The data structure for column i now stores only the rectangles in S whose left vertical edges lie in the column. (The row data structures remain the same.)
- The table T now stores two pieces of information per entry. For each cell (i, j) with $iK \leq t^*$,
  - let $T[i, j].s$ be the rectangle that contains the cell's left side, if such a rectangle exists;
  - if some horizontal edge in S cuts through the cell, then set $T[i, j].x = \infty$;
  - else set $T[i, j].x$ to be the largest x-coordinate among the right endpoints of the horizontal edges in S that intersect the cell's left side ($-\infty$ if no such edge exists).

Here is the modified query algorithm: given point $q = (x, y)$,

1. reset $x \leftarrow \min\{x, t^*\}$ and let $i = \lfloor x/K \rfloor$, $j = \lfloor y/L \rfloor$;

2. if q is inside $T[i,j].s$, then return $T[i,j].s$;

3. if $x \le T[i,j].x$, then recursively query the data structure for row j;

4. if $x > T[i,j].x$, then recursively query the data structure for column i.

To show correctness of the algorithm, let s be the rectangle containing q. Suppose s does not contain the left side of cell (i,j). If $x \le T[i,j].x$, then the left vertical edge of s cannot cut through the cell, by disjointness of the rectangles, so s must have a horizontal edge inside row j. If $x > T[i,j].x$, then the horizontal edges of s cannot intersect the left side of the cell, by maximality in the definition of $T[i,j].x$, so s must have its left vertical edge inside column i.

We now explain how the above data structure can be maintained under open and close operations. Opening/closing a rectangle in S requires recursively opening/closing it in at most 1 column and 2 row data structures. In addition, the table can be maintained as follows:

Consider the first moment when t^* passes through iK. We can compute $T[i,j].s$ at that time, for example, in $O(\log n)$ time for each j by a standard binary search tree. (The value of $T[i,j].s$ does not change afterwards.) Create a counter $C[i,j]$, defined as the number of horizontal edges in S that cut through cell (i,j). We can compute the value of $C[i,j]$ at that time, again in $O(\log n)$ time for each j by a binary search tree. If $C[i,j]$ is initially nonzero, then initialize $T[i,j].x$ to ∞; otherwise, set $T[i,j].x = -\infty$.

Each time we close a rectangle, at most 2 counters may need to be decremented. At the first moment when $C[i,j]$ drops to 0, set $T[i,j].x$ to current time t^*. (The value of $T[i,j].x$ will not be changed again.)

Thus, we can bound the total update time by the same recurrence as space in (1), except for at most a logarithmic factor increase. Note that we do not know the values of the m_i's and n_j's in advance, but can apply the doubling trick, which still guarantees that $\sum_i m_i + \sum_j n_j \le O(n)$ in the recurrence.

3.2 Second idea, with monotone list labeling

The strategy from Section 2.2 also has difficulties in supporting the open operation. Although we can insert to the dictionary D by hashing, the ranks $f[\cdot]$ of many elements in D can change in a single update.

Fortunately, we do not require $f[\cdot]$ to be exact ranks. The algorithm is correct so long as f is a *monotone labeling*, i.e., for every $y_1, y_2 \in D$ with $y_1 < y_2$, $f[y_1] < f[y_2]$ at all times. Monotone list labeling is a well-studied problem. For example, there are simple algorithms [11, 25, 28, 45] that can maintain a monotone labeling of D, under insertions to D, where the labels are integers bounded by $n^{O(1)}$, and each insertion requires only $O(\log n)$ amortized number of label changes. (The $n^{O(1)}$ bound can be reduced to $n^{1+\varepsilon}$ for any fixed $\varepsilon > 0$; it can even be improved to $O(n)$ by using more complicated algorithms, if the amortized number of label changes increases to $O(\log^2 n)$.) Monotone list labeling has been used before in (fully) persistent data structures, e.g., in Dietz's paper [26] (though there we label time values, i.e., x-coordinates, but here we label elements' values, i.e., y-coordinates).

Unfortunately, amortized bounds on the number of label changes are not adequate for our application—we need a worst-case bound on the number of label changes *per element* (not per insertion). Previous algorithms do not seem applicable directly. The author can think of a simple monotone labeling algorithm that guarantees each element changes labels at most $O(\log n)$ times, using labels bounded by $n^{O(\log n)}$, but this bound is too big.

Fortunately, we observe that a relaxed variant of the monotone list labeling problem is sufficient for our purposes. Here, we are allowed to use multiple (but a small number of) monotone labelings, and for each element y, we require that y changes labels a small number of times in just one of the labelings, within each of a certain set of time periods. More precisely:

Lemma 3.1 *For a multiset D that undergoes n insertions, we can maintain $\ell = O(\log n)$ monotone labelings $f_1, \ldots, f_\ell : D \to \{1, \ldots, n^c\}$ for some constant c, as well as a mapping $\chi : D \to \{1, \ldots, \ell\}$, in $O(n \log^2 n)$ total time, such that*

(a) *the total number of label changes in each f_i is $O(n \log n)$;*

(b) *for each $y \in D$, the number of changes to $\chi[y]$ is $O(\log n)$, and the number of changes to $f_i[y]$ during the time period when $\chi[y] = i$ is $O(\log n)$.*

The lemma will be proved later in Section 3.4, but let us clarify the meaning of the lemma in the case of multisets. In the definition of a monotone labeling f, we require that for any two copies y_1, y_2 of the same element, $f[y_1] = f[y_2]$. In (a) above, multiple copies of an element contribute multiple times in the total (i.e., multiplicities act as weights).

We now apply the above lemma to modify our data structure from Section 2.2. First, D is now a multiset, where each occurrence of a y-coordinate y in S induces a new copy of $\lfloor y/L \rfloor$ in D. We now use $\ell = O(\log n)$ bottom data structures, one for each f_i. For each rectangle $[x_1, \infty) \times [y_1, y_2)$ in S, each time $f_i[\lfloor y_1/L \rfloor]$ or $f_i[\lfloor y_2/L \rfloor]$ changes, we close $[x_1, \infty) \times [f_i[\lfloor y_1/L \rfloor]L + (y_1 \bmod L), f_i[\lfloor y_2/L \rfloor]L + (y_2 \bmod L))$ in the i-th bottom data structure at current time t^*, and open $[t^*, \infty) \times [f_i[\lfloor y_1/L \rfloor]L + (y_1 \bmod L), f_i[\lfloor y_2/L \rfloor]L + (y_2 \bmod L))$ with the new f_i value.

Opening/closing a rectangle in S requires opening/closing a rectangle in the top data structure and each of the bottom data structures. By (a), the total number of open/close operations in each bottom data structure is $O(n \log n)$.

Below is the new query algorithm: given point (x, y),

1. if $\lfloor y/L \rfloor \notin D$, then recursively query the top data structure for the point $(x, \lfloor y/L \rfloor)$;

2. if $\lfloor y/L \rfloor \in D$, then let i be the value of $\chi[\lfloor y/L \rfloor]$ at time x, let j be the value of $f_i[\lfloor y/L \rfloor]$ at time x, and recursively query the i-th bottom data structure for the point $(x, jL + (y \bmod L))$.

Membership in D can be decided in $O(1)$ time by dynamic perfect hashing [29]; each update in D takes $O(1)$ expected amortized time. By (b), we can compute i and j in line 2 by performing predecessor search over at most $n' = O(\log n)$ time values. This takes $O(1 + \log_w n') = O(1)$ time using fusion trees [37]. These trees can be updated naively in $(n')^{O(1)} = O(\log^{O(1)} n)$ time per change in χ and f_i (the update time can be reduced to $O(1)$ if we use atomic heaps [38]).

Replacing (2), the recurrences for space and query time become

$$\begin{aligned} S(n, W, H) &\leq O(\log n)\, S(O(n \log n), W, \sqrt{n^c H} + n) + O(n \log^2 n) \\ Q(n, W, H) &\leq Q(O(n \log n), W, \sqrt{n^c H} + n) + O(1). \end{aligned} \tag{7}$$

The total expected update time obeys the same recurrence as space, up to polylogarithmic factors.

The alternate recurrences (3) do not require modification: the elements of D here are x-values, and they are already inserted in monotone increasing order, so ranks do not need to be changed.

In Section 2.3, we can use the first strategy when $n > W^{1/(c+1)}, H^{1/(c+1)}$, and the second strategy otherwise. We get a recurrence similar to (4) with 3/4 changed to $1 - 1/(2(c+1))$, but now $\sum_i m_i + \sum_i n_j \leq O(n \log^2 n)$ (because of (7)). Solving the recurrences, we obtain the same query time $O(\log\log U)$ but with space and total update time $O(n(\log n)^{O(\log\log U)})$ (in expectation).

3.3 Last step, via sampling

It remains to lower the space and update time bounds. The separator-based approach from Section 2.4 does not adapt well with on-line updates, and we decide to switch to the sampling-based approach, which is the simplest:

It is more convenient now to consider the point location problem in the vertical decomposition $\mathcal{V}(S)$ of a set S of horizontal line segments. We want a data structure to support the open/close operations of segments, defined analogously (think of segments as degenerate rectangles). Take a random sample $R \subset S$ of size r. Build a data structure for the vertical decomposition $\mathcal{V}(R)$ of R, with $T_0(r)$ total (expected) update time, $S_0(r)$ space, and $Q_0(r)$ (expected) query time. For each cell γ_i in $\mathcal{V}(R)$, build a data structure for the subset of all n_i rectangles in S that intersect γ_i (i.e., the *conflict list* of γ_i), with $T_1(n_i)$ total update time, $S_1(n_i)$ space, and $Q_1(n_i)$ query time. By a standard analysis of Clarkson and Shor [23], $\sum_i n_i \leq O(n)$ and $n_i \leq O((n/r)\log r)$ with probability exceeding a positive constant. If these conditions fail, we rebuild from scratch; the expected number of trials is $O(1)$.

With this approach, we can get the same space and (expected) query bound as in (5):

$$\begin{aligned} S(n) &\leq S_0(r) + \sum_i S_1(n_i) + O(n) \\ Q(n) &\leq Q_0(r) + \max_i Q_1(n_i) + O(1), \end{aligned} \tag{8}$$

but now $\sum_i n_i \leq O(n)$ and $n_i \leq O((n/r)\log r)$. (Note that we do not know the exact values of the n_i's in advance, but can again apply the doubling trick.)

To support updates, it is best to use Bernoulli sampling, i.e., we put each segment in R independently with probability r/n. Clarkson and Shor's analysis still holds (and we can ensure in addition that $|R| \leq O(r)$ with good probability).

Opening/closing a segment $s \notin R$ requires opening/closing in the data structure for one cell γ_i, which can be found by a point location query in $\mathcal{V}(R)$. Opening (resp. closing) a rectangle $s \in R$ requires closing 1 cell and opening 2 cells (resp. closing 2 cells and opening 1 new cell) in $\mathcal{V}(R)$. The initial conflict list of the newly opened cells can be generated by scanning through the conflict list of the closed cells; these scans take total time $O(\sum_i n_i) = O(n)$. We open the segments from the new conflict lists in the new cells. Replacing (6), the total expected update time becomes

$$T(n) \leq T_0(r) + \sum_i T_1(n_i) + O(nQ_0(r)). \tag{9}$$

To illustrate the power of the above approach, we first show how to obtain bounds analogous to those of 1-d (ephemeral) exponential search trees [8, 9]. We can start with a naive method which stores r static predecessor search structures [10, 37, 54], to get $T_0(r), S_0(r) = r^{O(1)}$, and $Q_0(r) = O(\min\{\log\log U/\log\log\log U,\ 1 + \log_w r,\ \sqrt{\log r/\log\log r}\})$. By recursion, we can replace

$T_1(n_i)$, $S_1(n_i)$, and $Q_1(n_i)$ with $T(n_i)$, $S(n_i)$, and $Q(n_i)$. Choosing $r = n^\varepsilon$ for a sufficiently small constant $\varepsilon > 0$, we obtain a recurrence with $O(\log\log n)$ levels, yielding $T(n), S(n) = O(n \log^{O(1)} n)$, and $Q(n) = O(t(n, U))$ where

$$t(n,U) \;:=\; \min\left\{\frac{\log\log n \log\log U}{\log\log\log U},\ \log\log n + \log_w n,\ \sqrt{\frac{\log n}{\log\log n}}\right\}.$$

We can reduce space and update time further by bootstrapping. This time, we set $T_0(r), S_0(r) = O(r \log^{O(1)} r)$ and $Q_0(r) = O(t(r, U))$, and use a traditional 1-d persistent predecessor search data structure to get $T_1(n_i) = O(n_i \log n_i)$, $S_1(n_i) = O(n_i)$, and $Q_1(n_i) = O(\log n_i)$. Choosing $r = n/\log^{c'} n$ for a sufficiently large constant c' gives $T(n) = O(n\, t(n, U))$, $S(n) = O(n)$, and $Q(n) = O(t(n, U))$.

Now, we can demonstrate how to reduce the space and update time of our data structure from the previous subsection. We set $T_0(r), S_0(r) = O(r(\log r)^{O(\log\log U)})$ and $Q_0(r) = O(\log\log U)$, and $T_1(n_i) = O(n_i t(n_i, U)), S_1(n_i) = O(n_i)$ and $Q_1(n_i) = O(t(n_i, U))$. Choosing $r = \left\lfloor n/(\log n)^{c'' \log\log U}\right\rfloor$ for a sufficiently large constant c'', and noting that for $n_i \le (\log n)^{O(\log\log U)}$, $t(n_i, U) \le o(\log\log U)$, we obtain our final solution with $T(n) = O(n \log\log U)$, $S(n) = O(n)$, and $Q(n) = O(\log\log U)$.

Theorem 3.2 *We can build a partially persistent data structure with $O(n)$ space to maintain a set of integers in $\{1, \ldots, U\}$ under n updates in $O(\log\log U)$ amortized expected time, so that predecessor search can be performed in $O(\log\log U)$ expected time.*

3.4 Proof of the monotone labeling lemma

We still need to prove Lemma 3.1. Our proof below uses a standard monotone labeling algorithm as a black box. We proceed in three steps, the first of which is an extension of the standard labeling result to the multiset (i.e., weighted) setting:

Lemma 3.3 *For a multiset D that undergoes n insertions, we can maintain a monotone labeling $f : D \to \{1, \ldots, n^c\}$ for some constant c, in $O(n \log n)$ total time, such that the total number of label changes is $O(n \log n)$.*

Proof: Apply one of the known monotone labeling algorithms [11, 25, 28, 45] to a set $\widehat{D}$ containing D, where the copies of an element in D are treated as distinct elements in $\widehat{D}$. We can arbitrarily totally ordered the different copies of an element. Let $\widehat{f}$ be the resulting labeling for $\widehat{D}$, which undergoes $O(n \log n)$ label changes.

We cannot use $\widehat{f}$ directly, because our definition of monotone labeling requires different copies of the same element be mapped to the same label. We overcome this problem as follows. Let $y_1, \ldots, y_h$ denotes all current copies of an element y, in order. We maintain the invariant that $f[y_1] = \cdots = f[y_h]$ lies between $\widehat{f}[y_1]$ and $\widehat{f}[y_h]$; this guarantees monotonicity of f. At the moment that the invariant is violated, we restore it by resetting $f[y_1] = \cdots = f[y_h]$ to the current value of $\widehat{f}[y_{\lceil h/2\rceil}]$: denote the value by z. This requires h label changes for f, but ensures that the invariant will not be violated until at least $\lceil h/2\rceil$ label changes in $\widehat{f}[y_1], \ldots, \widehat{f}[y_h]$: for $\widehat{f}[y_1]$ to go above z (resp. $\widehat{f}[y_h]$ to go below z), $\widehat{f}[y_1], \ldots, \widehat{f}[y_{\lceil h/2\rceil}]$ have to go above z (resp. $\widehat{f}[y_{\lceil h/2\rceil}], \ldots, \widehat{f}[y_h]$ have to go below z). Thus, the total number of label changes in f is at most a constant times the total number of label changes in $\widehat{f}$. $\square$

Lemma 3.4 *For a multiset D that undergoes n insertions and a subset $A \subseteq D$ that undergoes at most n' insertions, we can maintain a monotone labeling $f : D \to \{1, \ldots, n^c\}$ for some constant c, in $O(n \log n)$ total time, such that the total number of label changes in D is $O(n \log n)$, and the total number of label changes in A is $O(n' \log n)$.*

Proof: Just apply Lemma 3.3 to a multiset $\widetilde{D}$ which contains D plus $\lceil n/n' \rceil$ copies of each element in A. The multiset $\widetilde{D}$ undergoes $O(n + (n/n')n') = O(n)$ insertions. Since the total number of label changes in $\widetilde{D}$ is $O(n \log n)$, the total number of label changes in A is $O(\frac{n \log n}{n/n'}) = O(n' \log n)$. □

Proof of Lemma 3.1: We maintain $\ell = \lfloor \log_2 n \rfloor + 1$ sets $A_1 \supseteq A_2 \supseteq \cdots \supseteq A_\ell$, where each A_i will undergo insertions only and have at most $n/2^{i-1}$ elements. We apply Lemma 3.4 to D and A_i to maintain a monotone labeling f_i.

The sets $A_1, \ldots, A_k$ are formed as follows. When we insert y to D, we insert it to A_1 (if it is not in A_1) and set $\chi[y] = 1$. For each $y \in A_i$, at the first moment that $f_i[y]$ changes more than $c' \log n$ times for a sufficiently large constant c', we insert y to A_{i+1} and set $\chi[y] = i + 1$ (we do not remove y from A_i and $f_i[y]$ can continue to change).

Assume inductively that $|A_i| \leq n/2^{i-1}$. Since the total number of changes to $f_i[y]$ over all $y \in A_i$ is at most $c_0(n/2^{i-1}) \log n$ for some absolute constant c_0, the number of elements $y \in A_i$ that changes more than $c' \log n$ is at most $\frac{c_0(n/2^{i-1}) \log n}{c' \log n} \leq n/2^i$ by choosing $c' = 2c_0$. So, indeed $|A_{i+1}| \leq n/2^i$.

Properties (a) and (b) follow immediately. □

4 Applications and Extensions

4.1 Immediate consequences

Orthogonal 2-d point location has many applications. We briefly mention some of them below. None of the geometric applications requires Section 3.

By Afshani's work [1] on dominance and orthogonal range reporting, which improved on Nekrich's earlier work [50], Theorem 2.1 immediately implies:

Corollary 4.1 *For a set of n points in $\{1, \ldots, U\}^d$ for a constant d, we can build a data structure for dominance reporting with*

- *$O(n)$ space and $O(\log \log U + k)$ query time for $d = 3$;*
- *$O(n \log^{d-3+\varepsilon} n)$ space and and $O((\log n / \log \log n)^{d-3} \log \log n + k)$ query time for $d \geq 4$.*

We can build a data structure for orthogonal range reporting with

- *$O(n \log^3 n)$ space and $O(\log \log U + k)$ query time for $d = 3$;*
- *$O(n \log^{d+\varepsilon} n)$ space and and $O((\log n / \log \log n)^{d-3} \log \log n + k)$ query time for $d \geq 4$.*

Here, $\varepsilon > 0$ is an arbitrarily small constant, and k denotes the output size of a query (we can set $k = 1$ for deciding whether the range is empty or for reporting one element in the range).

These improve previous query bounds by a $\log\log$ factor and are the best query bounds known among data structures with $O(n \operatorname{polylog} n)$ space on the RAM (results stated in Afshani, Arge, and Larsen's recent papers [2, 3] are for the pointer machine model or I/O model). Our query bound for the $d = 4$ case—$O(\log n + k)$—looks especially attractive and is optimal in a comparison model that permits $(\log n)$-bit RAM operations (although on the word RAM there remains a small gap from Pătraşcu's lower bound for 4-d orthogonal range reporting, with $\Omega(\log n/\log\log n)$ query time for $O(n \operatorname{polylog} n)$ space [53]).

For orthogonal range reporting, the polylogarithmic factors in space stated in the preceding corollary are not the best possible, and remain the subject of current investigation. For example, naively combining our point location result with a recent method of Karpinski and Nekrich [47] already reduces space by a $\log^{2-\varepsilon} n$ factor, though at the expense of increasing query time by a $\log\log n$ factor.

As one application of Corollary 4.1, we can build an $O(n \log^{1+\varepsilon} n)$-space data structure for n (possibly intersecting) rectangles in 2-d, so that all rectangles contained inside a query rectangle can be reported in $O(\log n + k)$ time. This follows by mapping each rectangle to a point in 4-d and reducing to 4-d dominance reporting.

Below are some further implications of Theorem 2.1 based on observations from de Berg, van Kreveld, and Snoeyink's work [15]:

Corollary 4.2 *Assume that input coordinates are from* $\{1, \ldots, U\}$.

(a) *Given n horizontal line segments in the plane, we can build an $O(n)$-space data structure so that the first k segments hit by a query vertical ray can be found (in sorted order) in $O(\log\log U + k)$ time.*

(b) *Given a 3-d orthogonal subdivision consisting n interior-disjoint boxes covering the entire space, we can build an $O(n \log\log U)$-space data structure so that point location queries can be answered in $O(\log\log U + (\log\log n)^2)$ time.*

(c) *Given a planar subdivision where each edge is parallel to one of a constant number of fixed directions, we can build an $O(n)$-space data structure so that point location queries can be answered in $O(\log\log U)$ time.*

(d) *Given n interior-disjoint* fat *triangles where all angles are greater than a positive constant, we can build an $O(n)$-space data structure so that point location queries can be answered in $O(\log\log U)$ time.*

(e) *Given n horizontal axis-aligned rectangles in 3-d, we can build an $O(n \log^{1+\varepsilon} n)$-space data structure so that vertical ray shooting queries can be answered in $O(\log n)$ time.*

(f) *Given n horizontal fat triangles in 3-d, we can build an $O(n \log^{1+\varepsilon} n)$-space data structure so that vertical ray shooting queries can be answered in $O(\log n)$ time.*

Part (a) follows from Chazelle's hive graph [21]; the 1-d persistent analog will be addressed later in Section 4.3.

In (b), de Berg *et al.* obtained $O((\log\log U)^3)$ query time in 3-d using the 2-d case as a subroutine; our improvement thus gives $O((\log\log U)^2)$, which can be rewritten as $O(\log\log U + (\log\log n)^2)$ by normalization. It is important that the boxes fill the entire space. A special case where the query time can be improved to $O(\log\log U)$ will be discussed in Section 4.2.

Part (c) follows by constructing the vertical decomposition of the edges of each direction separately, using a shear transformation. Part (d) follows from (c). Note that the query bound is much better than Chan and Pătraşcu's bound $O(\min\{\sqrt{\log U/\log\log U}, \log n/\log\log n\})$ for general planar point location [19]. The triangle fatness assumption is common in the literature.

For (e), de Berg *et al.* stated an $O(n\log n)$ space bound and an $O(\log n(\log\log n)^2)$ query bound by using a segment tree on top of orthogonal 2-d point location. Our improvement eliminates one $\log\log$ factor. To make the query bound even more attractive, we can trade off the other $\log\log$ factor if we increase space by $\log^{\varepsilon} n$, using a higher-degree segment tree. Part (f) follows similarly from another method by de Berg *et al.*

On another front, Nekrich [51] recently considered orthogonal planar point location in the I/O model and gave a structure with $O(n)$ space and $O((\log_2\log_B U)^2)$ query cost. It is straightforward to adapt our method to get an improved $O(\log_2\log_B U)$ query cost. In fact, our method works in the cache-oblivious model (in Section 2.4, we can use a bound $Q_1(n) = O(\log_B n)$ from [12]).

4.2 Point location in higher-dimensional orthogonal BSPs

We now show that our planar point location method from Section 2 can be extended to higher dimensions in one important special case: when the subdivision is an orthogonal BSP (we place no restrictions on the height of the BSP tree). Previously, Schwarz, Smid, and Snoeyink [56] had considered point location in this case and described an $O(\log n)$ query algorithm. The BSP case is important as it includes point location for k-d trees and quadtrees, which are useful in approximate nearest neighbor search; see [13] for other applications. Any set of disjoint boxes, or any orthogonal subdivision, can be refined into a BSP, although the size can increase considerably in the worst case; see [31, 43, 52, 59] for combinatorial bounds on orthogonal BSPs.

Let S be the n interior-disjoint leaf boxes in an orthogonal BSP in $\mathbb{R}^d$ where d is a constant. We seek a data structure that can find the box in S containing a query point (if the box exists).

In the first strategy from Section 2.1, we use a d-dimensional grid and change the definition of the table T as follows:

- if the cell $(i_1, \ldots, i_d)$ is contained in some box $s \in S$, then set $T[i_1, \ldots, i_d] =$ "covered by s";
- else if no edges parallel to some coordinate axis δ cut through the cell, then set $T[i_1, \ldots, i_d] = \delta$ (more than one choice of δ may be applicable).

It is easy to see that a direction δ must exist for orthogonal BSPs: take the first hyperplane cut h through the cell (if h does not exist, the cell would be covered by a box in S); the direction orthogonal to h satisfies the property.

During a query, if the query point q lies in cell $(i_1, \ldots, i_d)$ and $T[i_1, \ldots, i_d] = \delta$, then we can recurse in the data structure for the slab containing q between two grid hyperplanes orthogonal to δ.

We get a recurrence similar to (1), with $\sqrt{n}$ replaced by $n^{1/d}$.

The second strategy from Section 2.2 can be adapted easily. (Note that after applying either mapping, we still have a BSP.) We get recurrences similar to (2) and (3). Balancing the two recurrences as in Section 2.3 then gives a method with $T_0(n), S_0(n) = O(n\log^{O(1)} U)$ preprocessing time and space and $Q_0(n) = O(\log\log U)$ query time.

For the last step in Section 2.4 to lower space and preprocessing time, we use known tree separators [35]: in any binary tree with $O(n)$ nodes, there exist r edges whose removal leaves connected components (subtrees) of size $O(n/r)$, where each subtree is adjacent to $O(1)$ other subtrees; the subtrees can be formed in $O(n)$ time. We apply this fact to the underlying BSP tree. Build a point-location data structure for the subdivision $\mathcal{R}$ formed by the boundary of the boxes associated with roots of the r subtrees, with $T_0(O(r))$ preprocessing time, $S_0(O(r))$ space, and $Q_0(O(r))$ query time. Although the subdivision $\mathcal{R}$ may not be a BSP, we can further subdivide each of its cells $O(1)$ times to turn $\mathcal{R}$ into a BSP of size $O(r)$, since each cell in $\mathcal{R}$ is the set difference of an outer box with at most $O(1)$ inner boxes. For each subtree γ_i of size n_i, build a point-location data structure for the subdivision formed by the boxes in the subtrees, with $T_1(n_i)$ preprocessing time, $S_1(n_i)$ space, and $Q_1(n_i)$ query time. We then get the same bounds as in (5) and (6), but with $\sum_i n_i \leq n$ and $n_i \leq O(n/r)$.

To illustrate the power of the tree separator approach, we can start with setting r to be a sufficiently large constant and by recursion replace $T_1(n_i)$, $S_1(n_i)$, and $Q_1(n_i)$ with $T(n_i)$, $S(n_i)$, and $Q(n_i)$. This yields $T(n), S(n) = O(n \log n)$ and $Q(n) = O(\log n)$.

Next, we set $T_0(r), S_0(r) = O(r \log r)$ and $Q_0(r) = O(\log r)$, and trivially $T_1(n_i), S_1(n_i), Q_1(n_i) = O(n_i)$. Choosing $r = \lceil n/\log n \rceil$ gives $T(n), S(n) = O(n)$ and $Q(n) = O(\log n)$.

Finally, we reduce the space and preprocessing time of our data structure. We set $T_0(r), S_0(r) = O(r \log^{O(1)} U)$ and $Q_0(r) = O(\log\log U)$, and $T_1(n_i), S_1(n_i) = O(n_i)$ and $Q(n_i) = O(\log n_i)$. Choosing $r = \lceil n/\log^c U \rceil$ for a sufficiently large constant c gives $T(n), S(n) = O(n)$ and $Q(n) = O(\log\log U)$.

Theorem 4.3 *Given an orthogonal BSP in a constant dimension d with n leaf boxes whose coordinates are from $\{1, \ldots, U\}$, we can build an $O(n)$-space data structure in $O(n)$ time so that point location queries can be answered in $O(\log\log U)$ time.*

It can be checked that the hidden constant factors depend on d only polynomially.

4.3 k predecessors

In this subsection, we describe an extension of the data structure from Section 3 to find the first k predecessors of a query element at a given time in $O(\log\log U + k)$ expected time. In the geometric 2-d setting, this is equivalent to finding the first k segments hit by a downward query ray (in reverse y-sorted order), for a set S of horizontal segments, subject to open/close operations. Corollary 4.2(a) solves the same query problem using Chazelle's hive graphs, but this approach does not support on-line updates. We suggest following the sampling-based approach from Section 3.3, which is simpler.

We choose sample size $r = \lceil n/\log\log U \rceil$ and use our data structure with $T_0(r) = O(r \log\log U)$, $S_0(r) = O(r)$, and $Q_0(r) = O(\log\log U)$. The data structure of each cell γ_i is simply the conflict list of γ_i, sorted by y-coordinates, which can be maintained in a dynamic van Emde Boas tree, with $T_1(n_i) = O(n_i \log\log U)$ and $S_1(n_i) = O(n_i)$. By (8) and (9), we get total expected update time $T(n) = O(n \log\log U)$ and space $S(n) = O(n)$.

To answer a query for a ray from point q, we first locate the cell γ of $\mathcal{V}(R)$ containing q in $Q_0(r)$ time. We then find all segments (in reverse y-sorted order) hit by the ray inside γ, by scanning through the conflict list of γ. If the number of segments found is less than k, we locate the cell immediately below γ and repeat.

Assume that q is independent of the random choices made inside the data structure. Let $\overline{q}$ be the vertical segment from q to the k-th intersection along the ray. The expected number of cells of $\mathcal{V}(R)$ intersected by $\overline{q}$ is clearly $O(1 + kr/n)$. By Clarkson and Shor's technique [23], the expected value of $\sum\{n_i : \gamma_i \text{ intersects } \overline{q}\}$ is $O((1 + kr/n)(n/r))$. It follows that the expected query time is $O((1 + kr/n)Q_0(r) + (1 + kr/n)(n/r))$, which for our choice of r is $O(\log\log U + k)$.

Theorem 4.4 *We can build a partially persistent data structure with the same space and expected amortized update time as in Theorem 3.2, so that the first k predecessors of an element at a given time can be found (in sorted order) in $O(\log\log U + k)$ expected time.*

Remark: The same approach also yields $O(\log\log U + k)$ query time for persistent 1-d range search, i.e., reporting all elements inside a query interval at a given time, where the output size k is not given in advance.

5 Open Problems

We conclude with open problems. The following questions are natural to ask in light of our results, but might not necessarily have positive answers:

- Can the persistent predecessor search data structure in Section 3 be derandomized? Hashing is the main issue. Even for standard (ephemeral) dynamic predecessor search, the known deterministic upper bound is worse ($O(\log\log n \log\log U / \log\log\log U)$ query/update time [9]).
- Can our predecessor data structure be made fully persistent, with $O(\log\log U)$ update and query time? We could of course try applying monotone list labeling to not just y- but also x-coordinates (time), but additional ideas seem to be required.
- Can our orthogonal 2-d point location data structure in Section 2 for disjoint rectangles be made fully dynamic, to support insertions and deletions of rectangles in $O(\text{polylog}\, U)$ update time? An $O(n^\varepsilon)$ update time is straightforward (by using a smaller $n^{\varepsilon/2} \times n^{\varepsilon/2}$ table). For vertical ray shooting among horizontal line segments subject to insertions/deletions of segments, there is an $\Omega(\log n/\log\log n)$ query lower bound for $O(\text{polylog}\, n)$ update time, even when $U = n$, by a reduction from the marked ancestor problem [6].

In view of Corollary 4.2 and Theorem 4.3, can the 3-d orthogonal point location for arbitrary box subdivisions of size n (or, more generally, a collection of n disjoint boxes) be solved in $O(\log\log U)$ time with $O(n\, \text{polylog}\, U)$ space?

A minor open question, implicit in Section 3.2, is whether there is a monotone list labeling algorithm that supports insertions, where labels are bounded by $n^{O(1)}$ and each element changes labels at most a polylogarithmic number of times.

An interesting direction is to explore *offline* (or *batched*) query problems, in the spirit of [18, 20]. The version of 1-d persistent predecessor search where the updates and queries are all given in advance is equivalent to the version of orthogonal 2-d point location where n query points are given in advance. This in turn is equivalent to a basic geometric problem: construct the vertical decomposition of n horizontal line segments (since we can add a degenerate segment for each query point). An $O(n\log\log n)$ upper bound is obvious, by van Emde Boas trees. Can the problem be solved in the same time as integer sorting (currently $O(n\sqrt{\log\log n})$ with randomization [42])? Or, more challengingly, if the coordinates have been pre-sorted (i.e., normalized to $\{1, \ldots, n\}$), can the problem be solved in $O(n)$ time?

Note that an easier problem of constructing the lower envelope of n horizontal pre-sorted segments is known to be linear-time solvable [34]. (This problem corresponds to offline priority queues.)

Another interesting offline problem is the version of 1-d range search where the updates and queries are all given in advance. Note that in the on-line setting, 1-d range emptiness queries are known to have lower complexity than predecessor search ($O(1)$ query time in the static case [5], and $O(\log\log\log U)$ in the dynamic case with polylogarithmic update time [49]). The offline problem is equivalent to another basic geometric problem, *orthogonal 2-d segment intersection*: report the k intersections among n horizontal and vertical line segments. More simply, we can consider the emptiness problem: detect whether an intersection exists. Can the problem be solved faster than integer sorting? Or, if the coordinates have been pre-sorted, can the problem be solved in $O(n)$ time? (The author only knows of a linear-time solution in an asymmetric special case when there are a lot fewer horizontal segments than vertical segments, or vice versa.)

References

[1] P. Afshani. On dominance reporting in 3D. In *Proc. 16th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 5198, Springer-Verlag, pages 41–51, 2008.

[2] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting in three and higher dimensions. In *Proc. 50th IEEE Sympos. Found. Comput. Sci.*, pages 149–158, 2009.

[3] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher dimensional improvements. In *Proc. 26th ACM Sympos. Comput. Geom.*, pages 240–246, 2010.

[4] L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM J. Discrete Math.*, 9:129–150, 1996.

[5] S. Alstrup, G. Brodal, and T. Rauhe. Optimal static range reporting in one dimension. In *Proc. 33rd ACM Sympos. Theory Comput.*, pages 476–483, 2001.

[6] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th IEEE Sympos. Found. Comput. Sci.*, pages 534–543, 1998.

[7] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient regular data structures and algorithms for dilation, location, and proximity problems. *Algorithmica*, 30:164–187, 2001.

[8] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. 37th IEEE Sympos. Found. Comput. Sci.*, pages 135–141, 1996.

[9] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.

[10] P. Beame and F. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Sys. Sci.*, 65:38–72, 2002.

[11] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. 10th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 2461, Springer-Verlag, pages 152–164, 2002.

[12] M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proc. 29th Int. Colloq. Automata, Languages, and Programming*, Lect. Notes Comput. Sci., vol. 2380, Springer-Verlag, pages 195–207, 2002.

[13] M. de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica*, 28:353–366, 2000.

[14] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2000.

[15] M. de Berg, M. van Kreveld, and J. Snoeyink. Two-dimensional and three-dimensional point location in rectangular subdivisions. *J. Algorithms*, 18:256–277, 1995.

[16] M. Cary. Towards optimal ϵ-approximate nearest neighbor algorithms. *J. Algorithms*, 41:417–428, 2001.

[17] T. M. Chan. Closest-point problems simplified on the RAM. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 472–473, 2002.

[18] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, II: Offline search. In preparation. Preliminary version appeared in *Proc. 39th ACM Sympos. Theory Comput.*, pages 31–39, 2007.

[19] T. M. Chan and M. Pătraşcu. Transdichotomous results in computational geometry, I: Point location in sublogarithmic time. *SIAM J. Comput.*, 39:703–729, 2009.

[20] T. M. Chan and M. Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proc. 21st ACM-SIAM Sympos. Discrete Algorithms*, pages 161–173, 2010.

[21] B. Chazelle. Filtering search: a new approach to query-answering. *SIAM J. Comput.*, 15:703–724, 1986.

[22] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.

[23] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd ed., 2009.

[25] P. F. Dietz. Maintaining order in a linked list. In *Proc. 14th ACM Sympos. Theory Comput.*, pages 122–127, 1982.

[26] P. F. Dietz. Fully persistent arrays. In *Proc. 1st Workshop Algorithms Data Struct.*, Lect. Notes Comput. Sci., vol. 382, Springer-Verlag, pages 67–74, 1989.

[27] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proc. 2nd ACM-SIAM Sympos. Discrete Algorithms*, pages 78–88, 1991. Full version in Tech. Report 353, Computer Science Department, University of Rochester, 1990.

[28] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Sympos. Theory Comput.*, pages 365–372, 1987.

[29] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, 23:738–761, 1994.

[30] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Sys. Sci.*, 38:86–124, 1989.

[31] A. Dumitrescu, J. S. B. Mitchell, and M. Sharir. Binary space partitions for axis-parallel segments, rectangles, and hyperrectangles. *Discrete Comput. Geom.*, 31:207–227, 2004.

[32] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6:80–82, 1977.

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

[34] D. Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 827–835, 2001.

[35] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.

[36] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16:1004–1022, 1987.

[37] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Sys. Sci.*, 47:424–436, 1993.

[38] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Sys. Sci.*, 48:533–551, 1994.

[39] M. T. Goodrich. Planar separators and parallel polygon triangulation. *J. Comput. Sys. Sci.*, 51:374–389, 1995.

[40] T. Hagerup, P. B. Miltersen and R. Pagh. Deterministic dictionaries. *J. Algorithms*, 41:69–85, 2001.

[41] Y. Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. *J. Algorithms*, 50:96–105, 2004.

[42] Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space. In *Proc. 43rd IEEE Sympos. Found. Comput. Sci.*, pages 135–144, 2002.

[43] J. Hershberger, S. Suri, and C. D. Tóth. Binary space partitions of orthogonal subdivisions. *SIAM J. Comput.*, 34:1380–1397, 2005.

[44] J. Iacono and S. Langerman. Dynamic point location in fat hyperrectangles with integer coordinates. In *Proc. 12th Canad. Conf. Comput. Geom.*, pages 181–186, 2000.

[45] A. Itai, A. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. 8th Int. Colloq. Automata, Languages, and Programming*, Lect. Notes Comput. Sci., vol. 115, Springer-Verlag, pages 417–431, 1981.

[46] H. Kaplan. Persistent data structures. In *Handbook on Data Structures and Applications* (D. Mehta and S. Sahni, eds.), CRC Press, 2005.

[47] M. Karpinski and Y. Nekrich. Space efficient multi-dimensional range reporting. In *Proc. 15th Int. Conf. Comput. Comb.*, Lect. Notes Comput. Sci., vol. 5609, Springer-Verlag, pages 215–224, 2009.

[48] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[49] C. W. Mortensen, R. Pagh, and M. Pătraşcu. On dynamic range reporting in one dimension. In *Proc. 37th ACM Sympos. Theory Comput.*, pages 104–111, 2005.

[50] Y. Nekrich. A data structure for multi-dimensional range reporting. In *Proc. 23rd ACM Sympos. Comput. Geom.*, pages 344–353, 2007.

[51] Y. Nekrich. I/O-efficient point location in a set of rectangles. In *Proc. 8th Latin American Sympos. Theoretical Informatics*, Lect. Notes Comput. Sci., vol. 4957, Springer-Verlag, pages 687–698, 2008.

[52] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.

[53] M. Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, submitted. Also in *Proc. 49th IEEE Sympos. Found. Comput. Sci.*, pages 434–443, 2008.

[54] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th ACM Sympos. Theory Comput.*, pages 232–240, 2006.

[55] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[56] C. Schwarz, M. H. M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12:18–29, 1994.

[57] D. D. Sleator and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[58] J. Snoeyink. Point location. In *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, eds.), CRC Press, Boca Raton, FL, pages 559–574, 1997.

[59] G. Tóth. Binary space partitions for axis-aligned fat rectangles. *SIAM J. Comput.*, 38:429–447, 2008.

[60] D. E. Willard. Log-logarithmic worst case range queries are possible in space $\Theta(n)$. *Inform. Process. Lett.*, 17:81–84, 1983.