Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science (2014) 1–20
Theoretical Computer Science

Dynamic 3-Sided Planar Range Queries with Expected Doubly-Logarithmic Time☆

Gerth Stølting Brodal[a,1], Alexis C. Kaporis[b], Apostolos N. Papadopoulos[c], Spyros Sioutas[d,*], Konstantinos Tsakalidis[e], Kostas Tsichlas[c]

[a] *MADALGO, Department of Computer Science, Aarhus University, Denmark*
[b] *Computer Engineering and Informatics Department, University of Patras, Greece*
[c] *Department of Informatics, Aristotle University of Thessaloniki, Greece*
[d] *Department of Informatics, Ionian University, Corfu, Greece*
[e] *Computer Science and Engineering Department, CUHK, Hong Kong*

Abstract

The Priority Search Tree is the classic solution for the problem of dynamic 2-dimensional searching for the orthogonal query range of the form $[a, b] \times (-\infty, c]$ (3-sided rectangle). It supports all operations in logarithmic worst case complexity in both main and external memory. In this work we show that the update and query complexities can be improved to expected doubly-logarithmic, when the input coordinates are being continuously drawn from specific probability distributions. We present three pairs of linear space solutions for the problem, i.e. a RAM and a corresponding I/O model variant:

(1) First, we improve the update complexity to doubly-logarithmic expected with high probability, under the most general assumption that both the x- and y-coordinates of the input points are continuously being drawn from a distribution whose density function is unknown but fixed.

(2) Next, we improve both the query complexity to doubly-logarithmic expected with high probability and the update complexity to doubly-logarithmic amortized expected, by assuming that only the x-coordinates are being drawn from a class of *smooth* distributions, and that the deleted points are selected uniformly at random among the currently stored points. In fact, the y-coordinates are allowed to be arbitrarily distributed.

(3) Finally, we improve both the query and the update complexity to doubly-logarithmic expected with high probability by moreover assuming the y-coordinates to be continuously drawn from a more restricted class of realistic distributions.

All data structures are deterministic and their complexity's expectation is with respect to the assumed distributions. They comprise combinations of known data structures and of two new data structures introduced here, namely the *Weight Balanced Exponential Tree* and the *External Modified Priority Search Tree*.

Keywords: 3-sided range reporting, doubly logarithmic, average case analysis, dynamic data structures, computational geometry

☆This work is based on a combination of two conference papers that appeared in the Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC), 2010, pp. 1-12 (by all authors except the third one) and the Proceedings of the 13th International Conference on Database Theory (ICDT), 2010, pp. 34-43 (by all authors except the first one).

*Corresponding author

Email addresses: gerth@madalgo.au.dk (Gerth Stølting Brodal), kaporis@ceid.upatras.gr (Alexis C. Kaporis), papadopo@csd.auth.gr (Apostolos N. Papadopoulos), sioutas@ionio.gr (Spyros Sioutas), tsakalid@cse.cuhk.edu.hk (Konstantinos Tsakalidis), tsichlas@csd.auth.gr (Kostas Tsichlas)

[1] Work supported by the Danish National Research Foundation grant DNRF84 through Center for Massive Data Algorithmics (MADALGO).

1. Introduction

Recently, a significant effort has been made towards developing worst-case efficient data structures for range searching in two dimensions [37]. In their pioneering work, Kanellakis et al. [20, 21] illustrated that the problem of indexing in new data models, such as *constraint*, *temporal* and *object* models, can be reduced to special cases of two-dimensional indexing. In particular, they identified that *3-sided range searching* is of major importance.

The *3-sided range reporting query* in the 2-dimensional space is defined by an orthogonal region of the form $R = [a, b] \times (-\infty, c]$, i.e., a rectangular region with one side "open", and returns all points contained in R. Figure 1 depicts examples of possible 3-sided queries, defined by the shaded regions. Black dots represent the points comprising the result. In many applications, only positive coordinates are used and therefore, the region defining the 3-sided query always touches one of the two axes, according to application semantics.

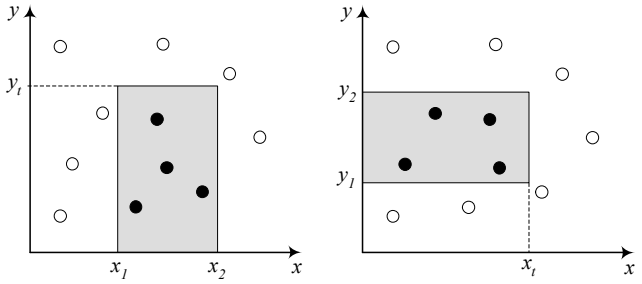

Figure 1. Examples of 3-sided queries.

We note that there is a plethora of applications that can benefit by efficient indexing schemes for 3-sided queries. Consider for example a spatio-temporal database that monitors the locations of moving objects to enable *location-based services*. The query asking for the set of objects that reached a specific point of interest during a time interval may be expressed as a 3-sided query where the time interval is represented by the values t_1 and t_2, whereas the other dimension limits the distance from the point of interest. In the sequel we provide a more detailed application example based on a *sensor network*.

Consider a time evolving database storing measurements collected from a sensor network. Assume that each measurement is represented by a multi-attribute tuple of the form $<id, a_1, a_2, ..., a_d, time>$, where *id* is the sensor identifier that produced the measurement, d is the total number of attributes, each a_i, $1 \leq i \leq d$, denotes the value of the specific attribute and finally *time* records the timestamp of the specific measurement. These values may relate to measurements regarding temperature, pressure, humidity, and so on. Therefore, each tuple may be seen as a *point* in $\mathbb{R}^d$ space. Let $F(p)$ be a real-valued *scoring function* that scores each point p based on the values of (a subset of) the attributes. Usually, the scoring function F is monotone and, without loss of generality, we assume that the lower the score the "better" the measurement (the other case is symmetric). Popular scoring functions are the aggregates SUM, MIN, AVG or other more complex combinations of the attributes. Now consider the query: "search for all measurements taken between the time instances t_1 and t_2 such that the score is below s". Notice that this is essentially a 2-dimensional 3-sided query with *time* as the x-axis and *score* as the y-axis, which may be expressed in SQL as follows:

SELECT *id*, *score*, *time*
FROM SENSOR_DATA
WHERE *time*>=t_1 AND *time*<=t_2 AND *score*<=s;

To support the previous application we need to provide efficient insertions and queries. Consider now the case where the measurements of interest belong to a *sliding window* [8] such that new measurements enter the window and old ones are evicted. This is a typical scenario of a stream-based application, where usually we need to process the data as they arrive since we can only perform online processing. In this case, deletions must be also handled efficiently, meaning that the data structure that is used to store the points contained in the sliding window must be a fully dynamic data structure. These techniques are ubiquitous in settings that require continuous query processing (e.g., [8], [31]). Search efficiency directly impacts query response time as well as the general system performance,

whereas update efficiency guarantees that incoming data are stored and organized quickly, thus, preventing delays due to excessive resource consumption.

Another important issue in such data intensive applications is memory consumption. The best practice is to keep data in main memory if this is possible. However, external memory solutions must also be available to cope with large data volumes and enable the operation of large databases. For this reason, in this work we study both cases, offering efficient solutions both in the RAM and I/O models of computation.

Related Work. The usefulness of 3-sided queries has been underlined many times in the literature [11, 21]. Apart from the significance of this query in multi-dimensional data intensive applications [12, 21], 3-sided queries appear in probabilistic threshold queries in uncertain databases. Such queries are studied in a recent work of Cheng et. al. [11]. The problem has been studied both in main memory (RAM model) and secondary storage (I/O model). In the internal memory, the most commonly used dynamic data structure for supporting 3-sided queries is the *Priority Search Tree* of McCreight [29]. It supports queries in $O(\log n + t)$ worst case time, insertions and deletions of points in $O(\log n)$ worst case time and uses linear $O(n)$ space, where n is the number of points and t the size of the output of a query. It is a hybrid of a binary heap for the y-coordinates and of a balanced search tree for the x-coordinates.

In the static case, when points have x-coordinates in the set of integers $\{0, \ldots, n-1\}$, the problem can be solved in $O(n)$ space and preprocessing time with $O(t+1)$ query time [2], using a *range minimum query* data structure [18] (see also Sec. 2).

The only dynamic sublogarithmic bounds for this problem in the RAM model are due to Willard [38] who attains $O(\log n/\log\log n)$ worst case or $O(\sqrt{\log n})$ randomized update time and $O(\log n/\log\log n + t)$ query time using linear space. This solution poses no assumptions on the input distribution.

Many external data structures such as grid files, various quad-trees, z-orders and other space filling curves, $k-d-B$-trees, hB-trees and various R-trees have been proposed. A recent survey can be found in [16]. Often these data structures are used in applications, because they are relatively simple, require linear space and perform well in practice most of the time. However, they all have highly sub-optimal worst case performance, whereas their expected performance is usually not guaranteed by theoretical bounds, since they are based on heuristic rules for the construction and update operations.

Moreover, several attempts have been made to externalize Priority Search Trees, including [10, 19, 21, 33, 35], but none of them was optimal. The worst case optimal external memory solution (External Priority Search Tree) was presented in [6]. It consumes $O(n/B)$ disk blocks, supports 3-sided range queries in $O(\log_B n + t/B)$ worst case I/Os and supports updates in $O(\log_B n)$ I/Os amortized, where B denotes the block size. This solution requires no assumptions on the input distribution. Also, there exist a static linear space external data structures that support 3-sided queries in $O(1 + t/B)$ worst case I/Os [27].

Our Contributions. In this work we present new dynamic linear space data structures for the RAM and the I/O models that support 3-sided range reporting queries and insertions and deletions of points an expected logarithmic factor more efficiently than previous worst case efficient solutions, when the coordinates of the input points are being drawn independently from various probabilistic distributions. All our structures are deterministic and they improve upon either the expected update (or even query) complexity of previous results. The expectation is only with respect to the input distribution. In particular, we propose three multi-level solutions, each with a main memory and an external memory variant. For the first solution, we assume that both the x- and y-coordinates of the input points are being continuously drawn from an unknown *μ-random* distribution. Regarding deletions, we make the standard assumption that the points to be deleted are selected uniformly at random among the points that are currently stored in the data structure [26]. The internal memory variant (Int1) achieves $O(\log n + t)$ worst case query time and $O(\log\log n)$ expected update time with high probability using $O(n)$ space. The corresponding external solution (Ext1) achieves $O(\log_B n + t/B)$ worst case query I/Os and $O(\log_B \log n)$ amortized expected update I/Os with high probability using $O(n/B)$ blocks of space. They are two-level constructions, where the upper level consists of a single (External) Priority Search Tree [29, 6] that indexes the structures of the lower level, which are (External) Priority Search Trees as well.

The second and the third solution attempt to further improve the query time. However, stricter assumptions have to be posed on the input distribution, in order to preserve the efficient expected update complexity. In particular, for the second solution, we assume that x-coordinates of points to be inserted are being drawn continuously from a $(f(n), g(n))$-*smooth* probabilistic distribution, where the functions f and g depend on the model. This assumption is

RAM Model	Query Time	Update Time
McCreight [29]	$\log n + t$	$\log n$
Willard [38]	$\frac{\log n}{\log\log n} + t$	$\frac{\log n}{\log\log n}$, [a] $\sqrt{\log n}$
Int1[b]	$\log n + t$	[c] $\log\log n$
Int2[d]	[c] $\log\log n + t$	[e] $\log\log n$
Int3[f]	[c] $\log\log n + t$	[c] $\log\log n$

I/O Model	Query I/Os	Update I/Os
Arge et al. [6]	$\log_B n + t/B$	[g] $\log_B n$
Ext1[b]	$\log_B n + t/B$	[h] $\log_B \log n$
Ext2[d]	[c] $\log\log_B n + t/B$	[e] $\log_B \log n$
Ext3[f]	[c] $\log_B \log n + t/B$	[h] $\log_B \log n$

Table 1. Asymptotic bounds for dynamic 3-sided planar range reporting in internal and external memory. The number of points currently stored in the structure is n, the size of the query output is t and the size of the block is B.

[a] Randomized algorithm and expected time bound.

[b] x- and y-coordinates are drawn from an unknown μ-random distribution, the μ function never changes, deletions are uniformly random over the inserted points.

[c] Expected with high probability.

[d] The x-coordinates are smoothly distributed, the y-coordinates are arbitrarily distributed and deletions are uniformly random over the inserted points

[e] Amortized expected.

[f] The x-coordinate distribution is $(f(n), g(n))$-*smooth*, for appropriate functions f and g depending on the model, and the y-coordinate distribution belongs to a more restricted class of distributions.

[g] Amortized.

[h] Amortized expected with high probability.

broad enough to include distributions used in practice, such as uniform, regular and classes of non-uniform ones [4, 23]. We pose no assumption on the distribution of the y-coordinates, which may in fact be selected adversarially. The internal variant (Int2) achieves $O(\log\log n + t)$ expected query time with high probability and $O(\log\log n)$ expected amortized update time using $O(n)$ space. The corresponding external solution (Ext2) achieves $O(\log\log_B n + t/B)$ expected query I/Os with high probability and $O(\log_B \log n)$ amortized expected update I/Os using $O(n/B)$ blocks. They combine (External) Interpolation Search Trees [4, 30, 22, 24] with (External) Weight Balanced Exponential Trees to handle the x- and y-coordinates, respectively. The latter structures achieve $O(1)$ expected amortized update complexity by combining techniques of Exponential Search Trees [3, 36, 5] and Weight Balanced B-Trees [7] and are also a contribution of this paper.

With the third solution we show that we can achieve similar or better results than the previous by further assuming that the distribution of the y-coordinates belongs to a more *restricted* class of distributions. These are non-smooth distributions, that are often found in practice, distributions like Zipfian and Power-Law in general. Specifically, now the update complexity holds with high probability and the the query I/O-complexity is improved by a $O(\log B)$ factor. In particular, the internal variant (Int3) achieves $O(\log\log n + t)$ expected query time with high probability and $O(\log\log n)$ expected amortized update time with high probability using $O(n)$ space. The corresponding external solution (Ext3) achieves $O(\log_B \log n + t/B)$ expected query I/Os with high probability and $O(\log_B \log n)$ amortized expected update I/Os with high probability using $O(n/B)$ blocks. These are multi-level constructions, where the top layer consists of linear space Modified Priority Search Trees [34]. For the external variant we introduce External Modified Priority Search Trees that support 3-sided queries in $O(t/B)$ expected I/Os with high probability using linear space. We employ them here, despite the result of [27], since they don't degrade the overall efficiency of our construction. Refer to Table 1 for a concrete comparison of our solutions with previous results.

Roadmap. The rest of the article is organized as follows. In Section 2, we discuss preliminary concepts, define the classes of probability distributions used and present the data structures that constitute the building blocks of our constructions. Among them, we introduce the *Weight Balanced Exponential Tree* and the *External Modified Priority Search Tree*. In Section 3 we present the two theorems that ensure the expected running times of the first and the third solution. The first solution is presented in Section 4, whereas our second and third solutions are discussed in Sections 5 and 6 respectively. Finally, Section 7 concludes the work and discusses briefly future research in the area.

2. Data Structures and Probability Distributions

For the main memory solutions we consider the RAM model of computation. We denote by n the number of elements that reside in the data structure and by t the size of the query. The universe of elements is denoted by S. When we mention that a data structure supports an update operation in a complexity that is *amortized expected with high probability*, we mean that the bound holds with high probability, under a worst case sequence of insertions and deletions of points.

For the external memory solutions we consider the I/O model of computation [1, 37]. That means that the input resides in external memory in a blocked fashion, in blocks that contain B elements. Whenever a computation needs to be performed to a particular element, the block that contains it is transferred into main memory, which can hold at most M elements. Every computation that is performed in main memory is free, since the block transfer is orders of magnitude more time consuming. Unneeded blocks that reside in the main memory are evicted by a LRU replacement algorithm. Naturally, the number of block transfers (*I/O operations* or *I/Os*) comprises the complexity measure.

Furthermore, we will consider that the points to be inserted are continuously drawn from specific distributions, presented in the sequel. The term *continuously* implies that the distribution from which we draw the points remains unchanged throughout the lifetime of the data structure. Since our constructions are dynamic, the asymptotic bounds are given with respect to the current size of the data structure. Finally, deletions of the elements of the data structures are assumed to be uniformly random. That is, every element present in the data structure is equally likely to be deleted.

2.1. Probability Distributions

In this section we overview the probabilistic distributions that will be used in the remainder of the paper. We will consider that the x- and y-coordinates are distinct elements of these distributions and will choose the appropriate distribution according to the assumptions of our solutions.

A probability distribution is μ-*random* if the elements are drawn randomly with respect to a density function denoted by μ. For this paper, we assume that μ is unknown.

Informally, a distribution defined over an interval I is *smooth* if the probability density over any subinterval of I does not exceed a specific bound, however small this subinterval is (i.e., the distribution does not contain sharp peaks). Given two functions f_1 and f_2, a density function $\mu = \mu[a, b](x)$ is (f_1, f_2)-*smooth* [30, 4] if there exists a constant β, such that for all c_1, c_2, c_3, $a \leq c_1 < c_2 < c_3 \leq b$, and all integers n, it holds that:

$$\int_{c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mu[c_1, c_3](x)dx \leq \frac{\beta \cdot f_2(n)}{n}$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu[c_1, c_3](x) = \mu(x)/p$ for $c_1 \leq x \leq c_3$ where $p = \int_{c_1}^{c_3} \mu(x)dx$. Intuitively, function f_1 partitions an arbitrary subinterval $[c_1, c_3] \subseteq [a, b]$ into f_1 equal parts, each of length $\frac{c_3 - c_1}{f_1} = O(\frac{1}{f_1})$; that is, f_1 measures how fine is the partitioning of an arbitrary subinterval. Function f_2 guarantees that no part, of the f_1 possible ones, gets more probability mass than $\frac{\beta \cdot f_2}{n}$; that is, f_2 measures the sparseness of any subinterval $[c_2 - \frac{c_3 - c_1}{f_1}, c_2] \subseteq [c_1, c_3]$. The class of (f_1, f_2)-smooth distributions (for appropriate choices of f_1 and f_2) is a superset of both regular and uniform classes of distributions, as well as of several non-uniform classes [4, 23]. Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β.

The *grid distribution* assumes that the elements are integers that belong to a specific range $\{1, \ldots, M\}$.

We define the class of *restricted distributions* to contain distributions used in practice, such as Zipfian and Power-Law. Their common attribute is that they exhibit sharp peaks and a long tail, as opposed to the smooth distribution.

The *Zipfian* distribution is a distribution of probabilities of occurrence that follows Zipf's law. Let N be the number of elements, k be their rank and s be the value of the exponent characterizing the distribution. Then Zipf's law is defined as the function $f(k; s, N) = \frac{1/k^s}{\Sigma_{k=1}^{N} 1/n^s}$. Intuitively, few elements occur very often, while many elements occur rarely.

The *Power-Law* distribution is a distribution over probabilities that satisfy $Pr[X \geq x] = cx^{-b}$ for constants $c, b > 0$.

2.2. Data Structures

In this section we describe the data structures that we will combine in order to achieve the desired complexities.

2.2.1. Priority Search Trees.

The classic *priority search tree* (PST) [29] stores points in 2-dimensional space. One of the most important operations that the PST supports is the *3-sided query*. The 3-sided query consists of a rectangle $[a, b] \times (-\infty, c]$ with one unbounded side and asks for all points that lie inside this area. Note that by rotaton we can unbound any edge of the rectangle. The PST supports this operation in $O(\log n + t)$ worst case time, where n is the number of stored points and t is the number of the reported points. Moreover, it supports the operations of inserting a new point, deleting of an existing point and searching for the x-coordinate of a point in $O(\log n)$ worst case time.

More specifically, the PST is a combination of a search tree and a priority queue. The search tree (an (a, b)-tree suffices) allows the efficient support of searches, insertions and deletions with respect to the x-coordinates, while the priority queue allows for efficient accessing of points with respect to their y-coordinate. In particular, the leaves of the PST contain all the points sorted by x-coordinate. The internal nodes of the tree contain auxiliary x- and y-values. The former is the median of the x-coordinates of the points stored in the subtree of the node, and is used to aid searching with respect to x-coordinate. The latter is the minimum y-coordinate of the points stored in the subtree. The y-values of the PST comprise a tournament among the y-coordinates of all the points stored in the PST, and they are used to aid efficient reporting of 3-sided queries.

Regarding the I/O model, after several attempts, a worst case optimal solution was presented by Arge et al. in [6]. The proposed indexing scheme consumes $O(n/B)$ blocks, supports updates in $O(\log_B n)$ amortized I/Os and answers 3-sided range queries in $O(\log_B n + t/B)$ worst case I/Os. We will refer to this indexing scheme as the *external priority search tree (EPST)*.

2.2.2. Interpolation Search Trees.

In [24] a dynamic data structure based on interpolation search (IS-Tree) was presented, which consumes linear space and can be updated in $O(1)$ worst case time, after identify the position of the change in the tree. Furthermore, the elements can be searched in $O(\log \log n)$ expected time with high probability, given that they are drawn from a (n^α, n^β)-smooth distribution, for any arbitrary constants $0 < \alpha, \beta < 1$.

The externalization of this data structure, called interpolation search B-tree (ISB-tree), was introduced in [22]. It supports update operations in $O(1)$ worst case I/Os provided that the update position is given and search operations in $O(\log_B \log n)$ expected I/Os with high probability, when the elements are drawn by a $(n/(\log \log n)^{1+\epsilon}, n^{1/B})$-smooth distribution, for constant $\epsilon > 0$. The worst case search bound is $O(\log_B n)$ I/Os.

2.2.3. Weight Balanced Exponential Trees.

The *exponential search tree* is a technique for converting static polynomial space search structures for ordered sets into fully-dynamic linear space data structures. It was introduced in [3, 36, 5] for searching and updating a dynamic set U of n integer keys in linear space and optimal $O(\sqrt{\log n / \log\log n})$ time in the RAM model. Effectively, to solve the dictionary problem, a doubly-logarithmic height search tree is employed that stores static local search structures of size polynomial to the degree of the nodes.

Here we describe a variant of the exponential search tree that we dynamize using a rebalancing scheme relative to that of the *weight balanced search trees* [7]. In particular, a *weight balanced exponential tree* T on n points is a leaf-oriented rooted search tree where the degrees of the nodes increase double exponentially on a leaf-to-root path. All leaves have the same depth and reside on the lowest level of the tree (level zero). The *weight* of a subtree T_u rooted at node u is defined to be the number of its leaves. If u lies at level $i \geq 1$, the weight of T_u ranges within $\left[\frac{1}{2} \cdot w_i + 1, 2 \cdot w_i - 1\right]$, for a *weight parameter* $w_i = c_1^{c_2^i}$ and constants $c_2 > 1$ and $c_1 \geq 2^{3/(c_2-1)}$ (see Lem. 2.1). Note that $w_{i+1} = w_i^{c_2}$. The root does not need to satisfy the lower bound of this range. The tree has height $\Theta(\log_{c_2} \log_{c_1} n)$.

The insertion of a new leaf to the tree increases the weight of the nodes on the leaf-to-root path by one. This might cause some weights to exceed their range constraints ("overflow"). We *rebalance* the tree in order to revalidate the constraints by a leaf-to-root traversal, where we "split" each node that overflowed. An overflown node u at level i has weight $2w_i$. A split is performed by creating a new node v that is a sibling of u and redistributing the children of u among u and v such that each node acquires a weight within the allowed range. In particular, we scan the children of u, accumulating their weights until we exceed the value w_i, say at child x. Node u gets the scanned children and v gets the rest. Node x is assigned as a child to the node with the smaller weight. Processing the overflown nodes u bottom up guarantees that, during the split of u, its children already satisfy their weight constraints.

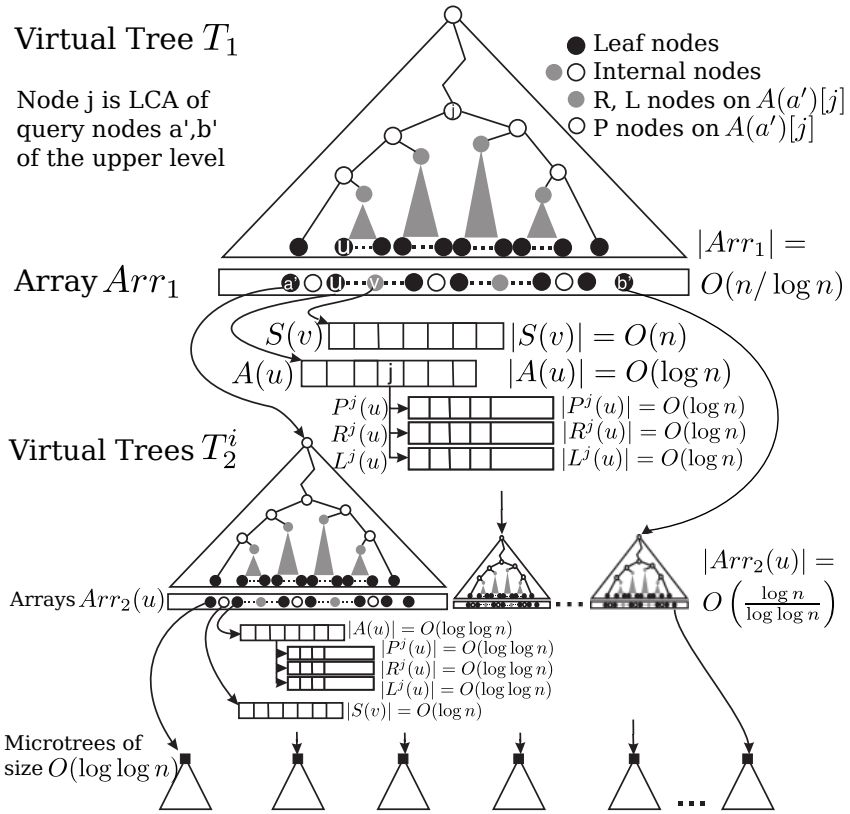

Figure 2. The linear space MPST stores (a) arrays Arr_1 and $Arr_2(u)$ that imply the structure of the virtual trees T_1 and $T_2^i, i \in \{1, \ldots, \log n\}$ by an inorder traversal, (b) for every array Arr the corresponding arrays S and A that index arrays P^j, R^j and L^j that in turn store pointers to corresponding internal nodes of the respective virtual trees, and (c) a global look-up table and the corresponding microtrees.

The deletion of a leaf might cause the nodes on the leaf-to-root path to "underflow", i.e. a node u at level i reaches weight $\frac{1}{2}w_i$. By an upwards traversal of the path, we discover the underflown nodes. In order to revalidate their node constraints, each underflown node chooses a sibling node v to "merge" with. That is, we assign the children of u to v and delete u. It may happen that the resulting node v overflows after the merge, i.e. its weight exceeds $\frac{3}{2}w_i$. In this case, we "split" it again as described above. This combined "merge" and "split" operation is called a "share". In either case, the traversal continues upwards, which guarantees that the children of the underflown nodes already satisfy their weight constraints. The following lemma, which is similar to [7, Lem. 9], holds.

Lemma 2.1. *After rebalancing a node u at level i, $\Omega(w_i)$ insertions or deletions need to be performed on T_u, for u to overflow or underflow again.*

Proof 2.2. *A split, a merge or a share on a node u on level i yield nodes with weight in $\left[\frac{3}{4}w_i - w_{i-1}, \frac{3}{2}w_i + w_{i-1}\right]$. If we set $w_{i-1} \leq \frac{1}{8}w_i$, which always holds for $c_1 \geq 2^{3/(c_2-1)}$, this interval is always contained in $[\frac{5}{8}w_i, \frac{14}{8}w_i]$.* □

2.2.4. Range Minimum Queries.

The *range minimum query* (RMQ) problem asks to preprocess an array of size n such that, given an index range, one can report the position of the minimum element in the range. In [18] the RMQ problem is solved in $O(1)$ worst case time using $O(n)$ space and preprocessing time. A succinct implementation that uses $2n + O(n)$ bits of space and supports the queries in $O(1)$ worst case time appears in [13].

2.2.5. Dynamic External Memory 3-sided Range Queries for $O(B^2)$ Points.

For external memory, Arge et al. [6] present the following lemma for handling a set of at most B^2 points.

Lemma 2.3. *A set of $K \leq B^2$ points can be stored in $O(K/B)$ blocks, such that 3-sided queries are supported in $O(t/B + 1)$ worst case I/Os and updates in $O(1)$ worst case I/Os, for output size t.*

Proof 2.4. *See Lemma 1 presented in [6].*

2.2.6. Modified Priority Search Trees.

A *modified priority search tree* (MPST) is a static data structure that stores points on the plane and supports 3-sided queries. It is stored as an array (*Arr*) in memory, yet it can be visualized as a complete binary tree. Although it has been presented in [25, 34] we sketch it here again, in order to introduce its external variant.

Let T be an MPST [34] which stores n points of S (see figure 2). We denote by T_v the subtree of T with root the internal node v. Like in an ordinary PST, v of the MPST is assigned with the point with minimum y-coordinate among the points in T_v. Let u be a leaf of the tree and let P_u be the root-to-leaf path for u. For every leaf u, we sort the points assigned in the nodes of P_u by their y-coordinate. We denote by P_u^j the subpath of P_u with nodes of depth bigger or equal to j (the depth of the root is 0). Similarly L_u^j (respectively R_u^j) denotes the set of nodes that are left (resp. right) children of nodes of P_u^j and do not belong to P_u^j . The tree structure T has the following properties:

- Each point of S is stored in a leaf of T and the points are in sorted x-order from left to right.
- Each internal node v is equipped with a secondary list $S(v)$ that contains the points stored in the leaves of T_v in non-decreasing y-coordinate.
- A leaf u also the lists $A(u)$, $P^j(u)$, $L^j(u)$ and $R^j(u)$, for $0 \leq j \leq \log n$. In particular, the lists $P^j(u)$, $L^j(u)$ and $R^j(u)$ store pointers to the respective internal nodes in non-decreasing y-coordinate, while $A(u)$ is an array that indexes j.

Note that the first element of the list $S(v)$ is the point of the subtree T_v with minimum y-coordinate. Also note that $0 \leq j \leq \log n$, so there are $\log n$ such sets P_u^j, L_u^j, R_u^j for each leaf u. Thus the size of A is $\log n$ and for a given j, any list $P^j(u)$, $L^j(u)$ or $R^j(u)$ can be accessed in constant time. By storing the nodes of the tree T according to their inorder traversal in an array *Arr* of size $O(n)$, we can imply the structure of tree T. Also each element of *Arr* contains a binary label that corresponds to the inorder position of the respective node of T, in order to facilitate constant time lowest common ancestor (LCA) queries.

To answer a query with the range $[a, b]\times(-\infty, c]$ we find the two leaves u, w of *Arr* that contain a and b respectively. If we assume that the leaves that contain a, b are given, we can access them in constant time. Then, since *Arr* contains an appropriate binary label, we use a simple LCA (Lowest Common Ancestor) algorithm [17, 18] to compute the depth j of the nearest common ancestor of u and w in $O(1)$ time. That is done by performing the XOR operation between the binary labels of the leaves u and w and finding the position of the first set bit, assuming that the left-most bit is placed in position 0. Afterwards, we traverse $P^j(u)$ until the scanned y-coordinate is not bigger than c. Next, we traverse $R^j(u)$, $L^j(w)$ in order to find the nodes whose stored points have y-coordinate not bigger than c. For each such node v we traverse the list $S(v)$ in order to report the points of *Arr* that satisfy the query constraints. Since we only access points that lie in the query range, the total query time is $O(1 + t)$, where t is the answer size.

The total size of the lists $S(u)$ for each level of T is $O(n)$. Each of the $O(n)$ leaves stores $\log n$ lists P_j, L_j and R_j, each of which consumes $O(\log n)$ space. Thus the space for these lists becomes $O(n \log^2 n)$. However, since P_{j+1} contains only extra element more than P_j, we can employ a partially persistent sorted list [9] to reduce the space of all lists P_j, similarly L_j and R_j to $O(n \log n)$, and thus also the total space for T.

We can further reduce the space of the structure by use of pruning, as in [14, 32]. However, pruning alone does not reduce the space to linear, even when it applied recursively. Thus, to obtain an optimal space bound we will use a combination of recursive pruning and table lookup. In particular, the pruning method is applied as follows: Consider the nodes of T that have height $\log \log n$. These nodes are roots of subtrees of T of size $O(\log n)$ and there are $O(n/\log n)$ such nodes. Let T_1 be the tree with these nodes as leaves and let T_2^i be the subtrees of these leaves for $1 \leq i \leq O(n/\log n)$. We call T_1 the first layer of the structure and the subtrees T_2^i the second layer. T_1 and each subtree

T_2^i is by itself a Modified Priority Search Tree. Note that T_1 has size $O(n/\log n)$, namely $o(n)$. Thus only the space of the second layer is $O(n \log n)$. We repeat the pruning at all the trees of the second layer, which results in a third layer that consists of $O(n/\log\log n)$ modified priority search trees each of size $O(\log\log n)$. Ignoring the third layer, now each subtree T_2^i of the second layer has $O(\log n/\log\log n)$ leaves and height $O(\log\log n)$. This means that now also the second layer takes $o(n)$, while the $O(n \log n)$ space bottleneck is charged only to the third level. To reduce the total space to $O(n)$, we can use the technique of table lookup [15] for the third level. Namely, since the size of every subtree T_3^i is $O(\log\log n)$, all the possible queries over all possible permutations of as many points are $O(n)$ in total, and thus fit in a linear space global look-up table. We implement the subtrees T_3^i as *microtrees*. Refer to [34, Section 3.2.3] for the details.

In order to answer a query on the three-layered structure we access the microtrees that contain a and b and extract in $O(1)$ time the part of the answer that they contain. Then we locate the subtrees T_2^i, T_2^j that contain the representative leaves of the accessed microtrees and extract the part of the answer that is contained in them by executing the query algorithm of the MPST. The roots of these subtrees are in turn leaves of T_1. Thus we execute again the MPST query algorithm on T_1 with these leaves as arguments. Once we reach the node with y-coordinate larger than c, we continue in the same manner top down. This may lead us to subtrees of the second layer that contain part of the answer and have not been accessed yet. That means that for each accessed tree of the second layer, we execute the MPST query algorithm, where instead of a and b, we set as arguments the minimum and the maximum x-coordinates of all the points stored in the queried tree. The argument c remains, of course, unchanged. Correspondingly, in that way we access the microtrees of the third layer that contain part of the answer. We execute the top down part of the algorithm on them, in order to report the final part of the answer.

Lemma 2.5. *Given a set of n points on the plane we can store them in a static data structure with $O(n)$ space that allows 3-sided range queries to be answered in $O(t)$ worst case, where t is the answer size.*

Proof 2.6. *See [34, Theorem 7].*

The *external modified priority search tree* (EMPST) is similar to the MPST, yet we store the lists in a blocked fashion. In order to attain linear space in external memory we prune the structure k times, instead of two times. The pruning terminates when $\log^{(k)} n = O(B)$. We assume that $k = O(1)$. Since computation within a block is free, we do not need the additional layer of microtrees. By that way we achieve $O(n/B)$ space.

Assume that the query algorithm accesses first the two leaves u and v of the k-th layer of the EMPST, which contain a and b respectively. If they belong to different EMPSTs of that layer, we recursively take the roots of these EMPSTs until the roots r_u and r_v belong to the same EMPST, w.l.o.g. the one on the top layer. That is done in $O(k) = O(1)$ I/Os. Then, in $O(1)$ I/Os we access the j-th entry of $A(r_u)$ and $A(r_v)$, where j is the depth of $LCA(r_u, r_v)$, thus also the corresponding sublists $P^j(r_u), R^j(r_u), L^j(r_u)$ and $P^j(r_v), R^j(r_v), L^j(r_v)$. Since these sublists are y-ordered, by scanning them in t_1/B I/Os we get all the t_1 pointers to the S-lists that contain part of the answer. We access the S-lists in t_1 I/Os and scan them as well in order to extract the part of the answer (let's say t_2) they contain. We proceed recursively to the $O(t_2)$ S-lists of the layer below.. In total, we consume $t_1/B + t_1 \cdot t_2/B + ... + t_{i-1} \cdot t_i/B + ... + t_{k-1} \cdot t_k/B$ I/Os. Let p_i the probability that $t_i = t^{p_i}$ where t is the total size of the answer and $\sum_{i=1}^{k} p_i = 1$. Thus, we need $t^{p_1}/B + \sum_{i=1}^{k-1} \frac{t^{p_i}}{B} \cdot t^{p_{i+1}}$ I/Os or $t^{p_1}/B + \sum_{i=1}^{k-1} \frac{t^{(p_i+p_{i+1})}}{B}$ I/Os. Assuming with high probability An equally likely distribution of answer amongst the k layers, we need $t^{\frac{1}{k}}/B + \sum_{i=1}^{k-1} \frac{t^{\frac{1}{k}+\frac{1}{k}}}{B}$ expected number of I/Os or $t^{\frac{1}{k}}/B + \sum_{i=1}^{k-1} \frac{t^{\frac{2}{k}}}{B}$. Since $k >> 2$, we need totally $O(t/B)$ expected with high probability I/Os. We conclude that:

Lemma 2.7. *Given a set of n points on the plane we can store them in a static external data structure with $O(n/B)$ space that supports 3-sided range queries in $O(t/B)$ expected with high probability I/Os, where t is the size of the answer.*

3. Expected First Order Statistic of Unknown Distributions

In this section we prove two theorems that will ensure the expected running times of our solutions in Sections 4 and 6.

They are multilevel data structures, where for each pair of consecutive levels, the upper level indexes representative elements of the lower level. We consider the elements to be points $p = (x, y)$ on the plane, where the x-coordinates are being drawn independently from the y-coordinates. We consider every lower level structure to be a *bucket* of points that occur consecutively by x-coordinate, and the *representative element* of the bucket to be the smallest y-coordinate in the bucket. An insertion or deletion of a point to a lower level bucket is a *violation*, when it causes the representative of the bucket to change, and thus triggers an update to the upper level. We consider *epochs* of consecutive update operations, during which we update the dynamic structures of the lower levels in an online fashion, and we buffer the violations to the upper level. The two theorems presented in this section ensure that the epochs are on the one hand short enough to allow for a few violations to affect the upper level, and on the other hand long enough to allow for amortizing its expensive update cost over the next epoch.

In particular, Theorem 3.5 allows epochs of logarithmic size that suffice to update efficiently the solution of Section 4, since updating its upper level structure needs logarithmic time. The proof requires the following propositions that assume that the x- and y-coordinates are being drawn independently from an unknown probability density function $\mu = \mathcal{F}$ over the continuous range $[a, b] \subseteq \mathfrak{R}$.

Proposition 3.1. *Consider an epoch of $O(\log n)$ updates and let $N(i) \in [n, r \cdot n]$, for constant $r > 1$, denote the structure after the epoch's i-th update. As long as $N(i)$ contains $O\left(\frac{n}{\log n}\right)$ buckets, the points in $N(i)$ remain μ-randomly distributed per i-th update.*

Proof 3.2. *The proof is analogous to [23, Lemma 2].*

Proposition 3.3. *Let $|S|$ denote the number of points stored in bucket S and let s be the value of its representative element. The probability that the next element drawn from $\mathcal{F}$ is less than s in $\mathcal{S}$ is equal to $\frac{1}{|\mathcal{S}|+1}$.*

Proof 3.4. *We employ backward analysis to prove the proposition. As a result, assume that the bucket contains $|S|+1$ elements and one element is deleted. Since all elements are drawn independently from the same arbitrary distribution $\mathcal{F}$, the element to be deleted may be any of the $|S|+1$ elements with equal probability. Thus, the probability of deleting the minimum element among the $|S|+1$ elements is $\frac{1}{|S|+1}$.*

Theorem 3.5. *For an epoch of $O(\log n)$ updates, the expected number of violations is $O(1)$, assuming that the x- and y-coordinates are being and continuously drawn from a μ-random distribution.*

Proof 3.6. *By [23, Th. 4], with high probability, each bucket j receives a x-coordinate with probability $p_j = \Theta(\frac{\log n}{n})$. It follows that during the i-th update operation, the elements in bucket j is a Binomial random variable X_j with mean $p_j \cdot N(i) = \Theta(\log n)$. By using the Chernoff bound, the probability that an arbitrary bucket j will have a number of elements $\leq \frac{p_j}{2} N(i)$ (less than half of the bucket's mean) is:*

$$Pr\left[X_j < \left(1 - \frac{1}{2}\right) p_j N(i)\right] < e^{-p_j N(i)/8} = n^{-\Theta(1)} \tag{1}$$

Suppose that an element is inserted in the i-th update. It induces a violation if its y-coordinate is strictly the minimum element of the bucket j it falls into.

- *If the bucket j contains $\geq \frac{p_j}{2} N(i) \geq \frac{p_j}{2} n$ coordinates, then by Proposition 3.3 element y incurs a violation with probability $O(\frac{1}{\log n})$.*
- *If the bucket j contains $< \frac{p_j}{2} N(i)$ coordinates, which happens with probability $n^{-\Theta(1)}$, then element y may induce 1 violation.*

Putting these cases together, element y expectedly induces at most $O(\frac{1}{\log n}) + n^{-\Theta(1)} = O(\frac{1}{\log n})$ violations. We conclude that during the whole epoch of $\log n$ insertions the expected number of violations is at most $\log n \cdot O(\frac{1}{\log n}) = O(1)$.

On the other hand, the upper level structure of the solution in Section 6 is a static structure, and thus it can be updated by global rebuilding [28] in linear time. Theorem 3.7 ensures that a few violations will occur in a broader epoch of linear size, if we restrict further the distribution of the y-coordinates.

Theorem 3.7. *For a sequence of $O(n)$ updates, the expected number of violating elements is $O(\log n)$, assuming that x- coordinates are drawn from a continuous smooth distribution and the y- coordinates are drawn from the restricted class of distributions (*Power-Law *or* Zipfian*).*

Proof 3.8. *Suppose an element is inserted with its y-coordinate following a discrete distribution (while its x-coordinate is arbitrarily distributed) in the universe $\{y_1, y_2, \ldots\}$ with $y_i < y_{i+1}, \forall i \geq 1$. Let $q = \Pr[y > y_1]$ and let y_j^* be the minimum y-coordinate of the elements in bucket j as soon as the current epoch starts. Clearly, the element just inserted incurs a violation when landing into bucket j with probability* $\Pr[y < y_j^*]$.

- *If the bucket contains $\geq \frac{p_j}{2} N(i) \geq \frac{p_j}{2} n$ coordinates, then coordinate y incurs a violation with probability $\leq q^{\frac{p_j}{2} n}$. (In other words, a violation may happen when at most all the $\Omega(\log n)$ coordinates of the elements in bucket j are $> y_1$; otherwise if $y_j^* = y_1$, then no violation is possible.)*
- *If the bucket contains $< \frac{p_j}{2} N(i)$ coordinates, which is as likely as in Eq. (1), then coordinate y may induce* 1 *violation.*

All in all, the new y coordinate expectedly induces $\leq q^{\Omega(\log n)} + n^{-\Theta(1)}$ violations. Thus, during the whole epoch of n insertions the expected number of violations is at most $n \cdot q^{\Omega(\log n)} + n^{1-\Theta(1)}$ violations. The constant in Θ is affected by p_j and we can choose it so that $n^{1-\Theta(1)} = O(1)$ and thus at most a constant number of violations are contributed by this part. The term $n \cdot q^{\Omega(\log n)}$ is at most $c \cdot \log n = O(\log n)$ provided that $q \leq \left(\frac{c \log n}{n}\right)^{(\log n)^{-1}} \to e^{-1}$ as $n \to \infty$. □

Remark 3.9. *Note that* Power-Law *and* Zipfian *distributions have the aforementioned property that $q \leq \left(\frac{c \log n}{n}\right)^{(\log n)^{-1}} \to e^{-1}$ as $n \to \infty$.*

4. The First Solution for x, y-Random Distributions

In this section we present the solution that works under the most general assumptions that the x- and y-coordinates are continuously drawn from an unknown μ-random distribution. The structure we propose consists of two levels, as well as an auxiliary data structure. All of them are implemented as PSTs. The lower level partitions the points into *buckets* of almost equal logarithmic size according to the x-coordinate of the points. That is, the points are sorted in increasing order according to x-coordinate and then divided into sets of $O(\log n)$ elements each of which constitutes a bucket. A bucket C is implemented as a PST and is represented by a point C^{min} which has the smallest y-coordinate among all points in it. This means that for each bucket the cost for insertion, deletion and search is equal to $O(\log \log n)$, since this is the height of the PST representing C.

The upper level is a PST on the representatives of the lower level. Thus, the number of leaves in the upper level is $O\left(\frac{n}{\log n}\right)$. As a result, the upper level supports the operations of insert, delete and search in $O(\log n)$ time. In addition, we keep an extra PST for insertions of violating points. Under this context, we call a point p *violating*, when its y-coordinate is less than C^{min} of the bucket C in which it should be inserted. In the case of a violating point we must change the representative of C and as a result we should make an update operation on the PST of the upper level, which costs too much, namely $O(\log n)$.

We assume that the x- and y-coordinates are drawn from an unknown μ-random distribution and that the μ function never changes. Under this assumption, according to the combinatorial game of bins and balls, presented in Section 5 of [23], the size of every bucket is $O(\log^c n)$, where $c > 0$ is a constant, and no bucket becomes empty with probability. We consider *epochs* of $O(\log n)$ update operations. During an epoch, according to Theorem 3.5, the number of violating points is expected to be $O(1)$ with high probability. The extra PST stores exactly those $O(1)$ violating points. When a new epoch starts, we take all points from the extra PST and insert them in the respective buckets in time $O(\log \log n)$ expected with high probability. Then we need to incrementally update the PST of the

upper level. This is done during the new epoch that just started. In this way, we keep the PST of the upper level updated and the size of the extra PST constant. As a result, the update operations are carried out in $O(\log\log n)$ time expected with high probability, since the update of the upper level costs $O(1)$ worst case time.

The 3-sided query can be carried out in the standard way. Assume the query $[a, b]\times(-\infty, c]$. First we search down the PST of the upper level for a and b. Let P_a be the search path for a and P_b for b respectively. Let $P_m = P_a \cap P_b$. Then, we check whether the points in the nodes on $P_a \cup P_b$ belong to the answer by checking their x-coordinate as well as their y-coordinate. Then, we check all right children of $P_a - P_m$ as well as all left children of $P_b - P_m$. In this case we just check their y-coordinate since we know that their x-coordinate belongs in $[a, b]$. When a point belongs in the query, we also check its two children and we do this recursively. After finishing with the upper level we go to the respective buckets by following a single pointer from the nodes of the upper level PST of which the points belong in the answer. Then we traverse in the same way the buckets and find the set of points to report. Finally, we check the extra PST for reported points. In total the query time is $O(\log n + t)$ worst case.

Note that deletions of points do not affect the correctness of the query algorithm. Indeed, removing the representative element of a bucket will force the higher levels to be updated, however this happens with low probability according to Theorems 3.5 and 3.7. If a non-violating point is deleted, it should reside on the lower level and thus it would be deleted online. Otherwise, the extra PST contains it and thus the deletion is online again. No deleted violating point is incorporated into the upper level, since by the end of the epoch the PST contains only inserted violating points.

Theorem 4.1. *There exists a dynamic main memory data structure that supports 3-sided queries in $O(\log n + t)$ worst case time, can be updated in $O(\log\log n)$ expected time with high probability and consumes linear space, under the assumption that the x- and y-coordinates are continuously drawn from a μ-random distribution.*

If we implement the above solution by using EPSTs [6], instead of PSTs, then the solution becomes I/O-efficient, however the update cost becomes amortized instead of worst case. Thus we get that:

Theorem 4.2. *There exists a dynamic external memory data structure that supports 3-sided queries in $O(\log_B n + t/B)$ worst case I/Os, can be updated in $O(\log_B \log n)$ amortized expected I/Os with high probability and consumes linear space, under the assumption that the x- and y-coordinates are continuously drawn from a μ-random distribution.*

5. The Second Solution for the x-Smooth Distributions

In this section we present the solution that poses no assumption on the distribution of the y-coordinates, but because it uses interpolation search trees to process the x-coordinates, it assumes that they are being drawn continuously from a smooth distribution. We will present the data structures for the RAM and the I/O model, respectively.

5.1. The Second Solution in RAM model

Our internal memory solution for storing n points in the plane consists of an IS-tree storing the points in sorted order with respect to the x-coordinates. We assume that the x-coordinates are drawn from a $(n^\alpha, n^{1/2})$-smooth distribution, for constants $1/2 < \alpha < 1$. On the sorted points, we maintain a weight balanced exponential search tree T with $c_2 = 3/2$ and $c_1 = 2^6$. Thus its height is $\Theta(\log\log n)$. In order to use T as a priority search tree, we augment it as follows. The root stores the point with overall minimum y-coordinate. Points are assigned to nodes in a top-down manner, such that a node u stores the point with minimum y-coordinate among the points in T_u that is not already stored at an ancestor of u. Note that the point from a leaf of T can only be stored at an ancestor of the leaf and that the y-coordinates of the points stored at a leaf-to-root path are monotonically non-increasing (*Min-Heap Property*). Finally, every node contains an RMQ-structure on the y-coordinates of the points in the children nodes and an array with pointers to the children nodes. Every point in a leaf can occur at most once in an internal node u and the RMQ-structure of u's parent. Since the space of the IS-tree is linear [30, 24], so is the total space.

5.1.1. Querying the Data Structure

Before we describe the query algorithm of the data structure, we will describe the query algorithm that finds all points with y-coordinate less than c in a subtree T_u. Let the query begin at an internal node u. At first we check if the y-coordinate of the point stored at u is smaller or equal to c (we call it a *member* of the query). If not we stop. Else, we identify the t_u children of u storing points with y-coordinate less than or equal to c using the RMQ-structure of u. That is, we first query the whole array and then recurse on the two parts of the array partitioned by the index of the returned point. The recursion ends when the point found has y-coordinate larger than c (*non-member* point).

Lemma 5.1. *For an internal node u and value c, all points stored in T_u with y-coordinate $\leq c$ can be found in $O(t+1)$ time, when t points are reported.*

Proof 5.2. *Querying the RMQ-structure at a node v that contains t_v member points will return at most $t_v + 1$ non-member points. We only query the RMQ-structure of a node v if we have already reported its point as a member point. Summing over all visited nodes we get a total cost of $O\left(\sum_v (2t_v + 1)\right) = O(t+1)$.* □

In order to query the whole structure, we first process a 3-sided query $[a,b] \times (-\infty, c]$ by searching for a and b in the IS-tree. The two accessed leaves a, b of the IS-tree comprise leaves of T as well. We traverse T from a and b to the root. Let P_a (resp. P_b) be the root-to-leaf path for a (resp. b) in T and let $P_m = P_a \cap P_b$. During the traversal we also record the index of the traversed child. When we traverse a node u on the path $P_a - P_m$ (resp. $P_b - P_m$), the recorded index comprises the leftmost (resp. rightmost) margin of a query to the RMQ-structure of u. Thus all accessed children by the RMQ-query will be completely contained in the query's x-range $[a,b]$. Moreover, by Lem. 5.1 the RMQ-structure returns all member points in T_u.

For the lowest node in P_m, i.e. the lowest common ancestor (LCA) of a and b, we query the RMQ-structure for all subtrees contained completely within a and b. We don't execute RMQ-queries on the rest of the nodes of P_m, since they root subtrees that contain the query's x-range. Instead, we merely check if the x- and y-coordinates of their stored point lies within the query. Since the paths P_m, $P_a - P_m$ and $P_b - P_m$ have length $O(\log\log n)$, the query time of T becomes $O(\log\log n + t)$. When the x-coordinates are smoothly distributed, the query to the IS-Tree takes $O(\log\log n)$ expected time with high probability [30]. Hence the total query time is $O(\log\log n + t)$ expected with high probability.

5.1.2. Inserting and Deleting Points

Before we describe the update algorithm of the data structure, we will first prove some properties of updating the points in T. Suppose that we decrease the y-value of a point p_u at node u to the value y'. Let v be the ancestor node of u highest in the tree with y-coordinate bigger than y'. We remove p_u from u. This creates an "empty slot" that has to be filled by the point of u's child with smallest y-coordinate. The same procedure has to be applied to the affected child, thus causing a *"bubble down"* of the empty slot until a node is reached with no points at its children. Next we replace v's point p_v with p_u (*swap*). We find the child of v that contains the leaf corresponding to p_v and swap its point with p_v. The procedure recurses on this child until an empty slot is found to place the last swapped out point (*"swap down"*). In case of increasing the y-value of a node the update to T is the same, except that p_u is now inserted at a node along the path from u to the leaf corresponding to p_u.

For every swap we will have to rebuild the RMQ-structures of the parents of the involved nodes, since the RMQ-structures are static data structures. This has a linear cost to the size of the RMQ-structure (Sect. 2).

Lemma 5.3. *Let i be the highest level where the point has been affected by an update. Rebuilding the RMQ-structures due to the update takes $O(w_i^{c_2-1})$ time.*

Proof 5.4. *The executed "bubble down" and "swap down", along with the search for v, traverse at most two paths in T. We have to rebuild all the RMQ-structures that lie on the two v-to-leaf paths, as well as that of the parent of the top-most node of the two paths. The RMQ-structure of a node at level j is proportional to its degree, namely $O\left(w_j/w_{j-1}\right)$. Thus, the total time becomes $O\left(\sum_{j=1}^{i+1} w_j/w_{j-1}\right) = O\left(\sum_{j=0}^{i} w_j^{c_2-1}\right) = O\left(w_i^{c_2-1}\right)$.* □

To insert a point p, we first insert it in the IS-tree. This creates a new leaf in T, which might cause several of its ancestors to overflow. We split them as described in Sec. 2. For every split a new node is created that contains no point. This empty slot is filled by "bubbling down" as described above. Next, we search on the path to the root for

the node that p should reside according to the Min-Heap Property and execute a "swap down", as described above. Finally, all affected RMQ-structures are rebuilt.

To delete a point p, we first locate it in the IS-tree, which points out the corresponding leaf in T. By traversing the leaf-to-root path in T, we find the node in T that stores p. We delete the point from the node and "bubble down" the empty slot, as described above. Finally, we delete the leaf from T and rebalance T if required. Merging two nodes requires one point to be "swapped down" through the tree. In case of a share, we additionally "bubble down" the new empty slot. Finally we rebuild all affected RMQ-structures and update the IS-tree.

Analysis: We assume that the point to be deleted is selected uniformly at random among the points stored in the data structure. Moreover, we assume that the inserted points have their x-coordinates drawn independently at random from an $(n^\alpha, n^{1/2})$-smooth distribution for a constant $1/2<\alpha<1$, and that the y-coordinates are drawn from an arbitrary distribution. Searching and updating the IS-tree needs $O(\log\log n)$ expected with high probability [30, 24], under the same assumption for the x-coordinates.

Lemma 5.5. *Starting with an empty weight balanced exponential tree, the amortized time of rebalancing it due to insertions or deletions is $O(1)$.*

Proof 5.6. *A sequence of n updates requires at most $O(n/w_i)$ rebalancings at level i (Lem. 2.1). Rebuilding the RMQ-structures after each rebalancing costs $O\left(w_i^{c_2-1}\right)$ time (Lem. 5.3). Summing over all levels, the total time becomes $O(\sum_{i=1}^{height(T)} \frac{n}{w_i}\cdot w_i^{c_2-1}) = O(n\sum_{i=1}^{height(T)} w_i^{c_2-2}) = O(n)$, when $c_2<2$.* □

Lemma 5.7. *The expected amortized time for inserting or deleting a point in a weight balanced exponential tree is $O(1)$.*

Proof 5.8. *The insertion of a point creates a new leaf and thus T may rebalance, which by Lemma 5.5 costs $O(1)$ amortized time. Note that the shape of T only depends on the sequence of updates and the x-coordinates of the points that have been inserted. The shape of T is independent of the y-coordinates, but the assignment of points to the nodes of T follows uniquely from the y-coordinates, assuming all y-coordinates are distinct. Let u be the ancestor at level i of the leaf for the new point p. For any integer $k \geq 1$, the probability of p being inserted at u or an ancestor of u can be bounded by the probability that a point from a leaf of T_u is stored at the root down to the k-th ancestor of u plus the probability that the y-coordinate of p is among the k smallest y-coordinates of the leaves of T. The first probability is bounded by $\sum_{j=i+k}^{height(T)} \frac{2w_{j-1}}{\frac{1}{2}w_j}$, whereas the second probability is bounded by $k\big/\frac{1}{2}w_i$. It follows that p ends up at the i-th ancestor or higher with probability at most $O\left(\sum_{j=i+k}^{height(T)} \frac{2w_{j-1}}{\frac{1}{2}w_j} + \frac{k}{\frac{1}{2}w_i}\right) = O\left(\sum_{j=i+k}^{height(T)} w_{j-1}^{1-c_2} + \frac{k}{w_i}\right) = O\left(w_{i+k-1}^{1-c_2} + \frac{k}{w_i}\right) = O\left(w_i^{(1-c_2)c_2^{k-1}} + \frac{k}{w_i}\right) = O\left(\frac{1}{w_i}\right)$ for $c_2 = 3/2$ and $k = 3$. Thus the expected cost of "swapping down" p becomes $O\left(\sum_{i=1}^{height(T)} \frac{1}{w_i}\cdot\frac{w_{i+1}}{w_i}\right) = O\left(\sum_{i=1}^{height(T)} w_i^{c_2-2}\right) = O\left(\sum_{i=1}^{height(T)} c_1^{(c_2-2)c_2^i}\right) = O(1)$ for $c_2 < 2$.*

A deletion results in "bubbling down" an empty slot, whose cost depends on the level of the node that contains it. Since the point to be deleted is selected uniformly at random and there are $O(n/w_i)$ points at level i, the probability that the deleted point is at level i is $O(1/w_i)$. Since the cost of an update at level i is $O(w_{i+1}/w_i)$, we get that the expected "bubble down" cost is $O\left(\sum_{i=1}^{height(T)} \frac{1}{w_i}\cdot\frac{w_{i+1}}{w_i}\right) = O(1)$ for $c_2 < 2$. □

Theorem 5.9. *In the RAM model using $O(n)$ space, 3-sided queries can be supported in $O(\log\log n + t)$ expected time with high probability, and updates in $O(\log\log n)$ expected amortized time, given that the x-coordinates of the inserted points are drawn from an $(n^\alpha, n^{1/2})$-smooth distribution for constant $1/2<\alpha<1$, the y-coordinates from an arbitrary distribution, and that the deleted points are drawn uniformly at random among the stored points.*

5.2. The Second Solution in I/O model

We now convert our internal memory into a solution for the I/O model. First we substitute the IS-tree with its variant in the I/O model, the ISB-Tree [22]. Thus we assume that the x-coordinates are drawn from a $(n/(\log\log n)^{1+\epsilon}, n^{1/B})$-smooth distribution, for constant $\epsilon > 0$. We implement every consecutive $\Theta(B^2)$ leaves of the ISB-Tree with the data structure of Arge et al. [6]. Each such structure constitutes a leaf of a weight balanced exponential tree T that we build on top of the $O(n/B^2)$ leaves.

In T every node now stores B points sorted by y-coordinate, such that the maximum y-coordinate of the points in a node is smaller than all the y-coordinates of the points of its children (Min-Heap Property). The B points with overall smallest y-coordinates are stored at the root. At a node u we store the B points from the leaves of T_u with smallest y-coordinates that are not stored at an ancestor of u. At the leaves we consider the B points with smallest y-coordinate among the remaining points in the leaf to comprise this list. Moreover, we define the weight parameter of a node at level i to be $w_i{=}B^{2\cdot(7/6)^i}$. Thus we get $w_{i+1}{=}w_i^{7/6}$, which yields a height of $\Theta(\log\log_B n)$. Let $d_i{=}\frac{w_i}{w_{i-1}}{=}w_i^{1/7}$ denote the *degree parameter* for level i. All nodes at level i have degree $O(d_i)$. Also every node stores an array that indexes the children according to their x-order.

We furthermore need a structure to identify the children with respect to their y-coordinates. We replace the RMQ-structure of the internal memory solution with a table. For every possible interval $[k,l]$ over the children of the node, we store in an entry of the table the points of the children that belong to this interval, sorted by y-coordinate. Since every node at level i has degree $O(d_i)$, there are $O(d_i^2)$ different intervals and for each interval we store $O(B\cdot d_i)$ points. Thus, the total size of this table is $O(B\cdot d_i^3)$ points or $O(d_i^3)$ disk blocks.

The ISB-Tree consumes $O(n/B)$ blocks [22]. Each of the $O(n/B^2)$ leaves of T contains B^2 points. Each of the n/w_i nodes at level i contains B points and a table with $O(B\cdot d_i^3)$ points. Thus, the total space is $O\left(n{+}\sum_{i=1}^{height(T)} n\cdot B\cdot d_i^3/w_i\right) = O\left(n{+}\sum_{i=1}^{height(T)} n\cdot B\Big/\left(B^{2\cdot\frac{7}{6}i}\right)^{\frac{4}{7}}\right) = O(n)$ points, i.e. $O(n/B)$ disk blocks.

5.2.1. Querying the Data Structure

The query is similar to the internal memory solution. First we access the ISB-Tree, spending $O(\log_B\log n)$ expected I/Os with high probability, given that the x-coordinates are smoothly distributed [22]. This points out the leaves of T that contain a,b. We perform a 3-sided range query at the two leaf structures. Next, we traverse upwards the leaf-to-root path P_a (resp. P_b) on T, while recording the index k (resp. l) of the traversed child in the table. That costs $\Theta(\log\log_B n)$ I/Os. At each node we report the points of the node that belong to the query range. For all nodes on P_a-P_b and P_b-P_a we query as follows: We access the table at the appropriate children range, recorded by the index k and l. These ranges are always $[k+1,$last child$]$ and $[0,l-1]$ for the node that lie on P_a-P_b and P_b-P_a, respectively. The only node where we access a range $[k+1,l-1]$ is the LCA of the leaves that contain a and b. The recorded indices facilitate access to these entries in $O(1)$ I/Os. We scan the list of points sorted by y-coordinate, until we reach a point with y-coordinate bigger than c. All scanned points are reported. If the scan has reported all B elements of a child node, the query proceeds recursively to that child, since more member points may lie in its subtree. Note that for these recursive calls, we do not need to access the B points of a node v, since we accessed them in v's parent table. The table entries they access contain the complete range of children. If the recursion accesses a leaf, we execute a 3-sided query on it, with respect to a and b [6].

The list of B points in every node can be accessed in $O(1)$ I/Os. The construction of [6] allows us to load the B points with minimum y-coordinate in a leaf also in $O(1)$ I/Os. Thus, traversing P_a and P_b costs $\Theta(\log\log_B n)$ I/Os worst case. There are $O(\log\log_B n)$ nodes u on P_a-P_m and P_b-P_m. The algorithm recurses on nodes that lie within the x-range. Since the table entries that we scan are sorted by y-coordinate, we access only points that belong to the answer. Thus, we can charge the scanning I/Os to the output. The algorithm recurses on all children nodes whose B points have been reported. The I/Os to access these children can be charged to their points reported by their parents, thus to the output. That allows us to access the child even if it contains only $o(B)$ member points to be reported. The same property holds also for the access to the leaves. Thus we can perform a query on a leaf in $O(t/B)$ I/Os. Summing up, the worst case query complexity of querying T is $O(\log\log_B n+\frac{t}{B})$ I/Os. Hence in total the query costs $O(\log\log_B n+\frac{t}{B})$ expected I/Os with high probability.

5.2.2. Inserting and Deleting Points

Insertions and deletions of points are in accordance with the internal solution. For the case of insertions, first we update the ISB-tree. This creates a new leaf in the ISB-tree that we also insert at the appropriate leaf of T in $O(1)$ I/Os [6]. This might cause some ancestors of the leaves to overflow. We split these nodes, as in the internal memory solution. For every split B empty slots "bubble down". Next, we update T with the new point. For the inserted point p we locate the highest ancestor node that contains a point with y-coordinate larger than p's. We insert p in the list of the node. This causes an excess point, namely the one with maximum y-coordinate among the B points stored in the node, to "swap down" towards the leaves. Next, we scan all affected tables to replace a single point with a new one.

In case of deletions, we search the ISB-tree for the deleted point, which points out the appropriate leaf of T. By traversing the leaf-to-root path and loading the list of B point, we find the point to be deleted. We remove the point from the list, which creates an empty slot that "bubbles down" T towards the leaves. Next we rebalance T as in the internal solution. For every merge we need to "swap down" the B largest excess points. For a share, we need to "bubble down" B empty slots. Next, we rebuild all affected tables and update the ISB-tree.
Analysis: Searching and updating the ISB-tree requires $O(\log_B \log n)$ expected I/Os with high probability, given that the x-coordinates are drawn from an $(n/(\log\log n)^{1+\varepsilon}, n^{1/B})$-smooth distribution, for constant $\varepsilon>0$ [22].

Lemma 5.10. *For every path corresponding to a "swap down" or a "bubble down" starting at level i, the cost of rebuilding the tables of the paths is $O\left(d_{i+1}^3\right)$ I/Os.*

Proof 5.11. *Analogously to Lem. 5.3, a "swap down" or a "bubble down" traverse at most two paths in T. A table at level j costs $O(d_j^3)$ I/Os to be rebuilt, thus all tables on the paths need $O\left(\sum_{j=1}^{i+1} d_j^3\right) = O\left(d_{i+1}^3\right)$ I/Os.* □

Lemma 5.12. *Starting with an empty external weight balanced exponential tree, the amortized I/Os for rebalancing it due to insertions or deletions is $O(1)$.*

Proof 5.13. *We follow the proof of Lem. 5.5. Rebalancing a node at level i requires $O\left(d_{i+1}^3 + B\cdot d_i^3\right)$ I/Os (Lem. 5.10), since we get B "swap downs" and "bubble downs" emanating from the node. The total I/O cost for a sequence of n updates is $O\left(\sum_{i=1}^{height(T)} \frac{n}{w_i}\cdot(d_{i+1}^3 + B\cdot d_i^3)\right) = O\left(n\cdot\sum_{i=1}^{height(T)} w_i^{-1/2} + B\cdot w_i^{-4/7}\right) = O(n)$.* □

Lemma 5.14. *The expected amortized I/Os for inserting or deleting a point in an external weight balanced exponential tree is $O(1)$.*

Proof 5.15. *By similar arguments as in Lem. 5.7 and considering that a node contains B points, we bound the probability that point p ends up at the i-th ancestor or higher by $O(B/w_i)$. An update at level i costs $O(d_{i+1}^3) = O\left(w_i^{1/2}\right)$ I/Os. Thus "swapping down" p costs $O\left(\sum_{i=1}^{height(T)} w_i^{1/2}\cdot\frac{B}{w_i}\right) = O(1)$ expected I/Os. The same bound holds for deleting p, following similar arguments as in Lem. 5.7.* □

Theorem 5.16. *In the I/O model using $O(n/B)$ disk blocks, 3-sided queries can be supported in $O(\log\log_B n + t/B)$ expected I/Os with high probability, and updates in $O(\log_B \log n)$ expected amortized I/Os, given that the x-coordinates of the inserted points are drawn from an $(n/(\log\log n)^{1+\varepsilon}, n^{1/B})$-smooth distribution for a constant $\varepsilon > 0$, the y-coordinates from an arbitrary distribution, and that the deleted points are drawn uniformly at random among the stored points.*

6. The Third Solution for the x-Smooth and the y-Restricted Distributions

In this section we modify the solution of Section 4 such that it improves over both the query and the update complexity of the previous solutions at the expense of further restricting the distribution of the y-coordinates. The query time of the solution of Section 4 is too high, namely $O(\log n)$ and that is due to locating the leaves a and b in the upper level PST. To further improve it to doubly-logarithmic query time, we assume that the x-coordinates are drawn from a (n^α, n^β)-smooth, for constants $0 < \alpha, \beta < 1$, and use an IS-tree to index them. By doing that, we pay with high probability $O(\log\log n)$ time to locate a and b.

We can also retain doubly-logarithmic update time, by getting rid of the upper level PST all in all and substituting it with a static MPST that is dynamized by global rebuilding [28]. However, this costs $O(n)$ time and thus we must ensure that in a broader epoch of n updates, wherein the rebuilding can completed in $O(1)$ amortized or even worst case time, the number of violations does not allow the extra PST to affect the overall complexity. Theorem 3.7 ensures its size is $O(\log n)$ during an epoch, if the y-coordinates are drawn from a restricted class of distributions.

When a new epoch starts we take all points from the extra PST and insert them in the respective buckets in time $O(\log\log n)$ with high probability. During the epoch we gather all the violating points that should access the MPST and the points that belong to it and build in parallel a new MPST that also contains them. At the end of the $O(n)$ epoch, we have built the updated version of the MPST, which we use for the next epoch that just started. By this way, we keep the MPST of the upper level updated and the size of the extra PST logarithmic. By incrementally constructing the

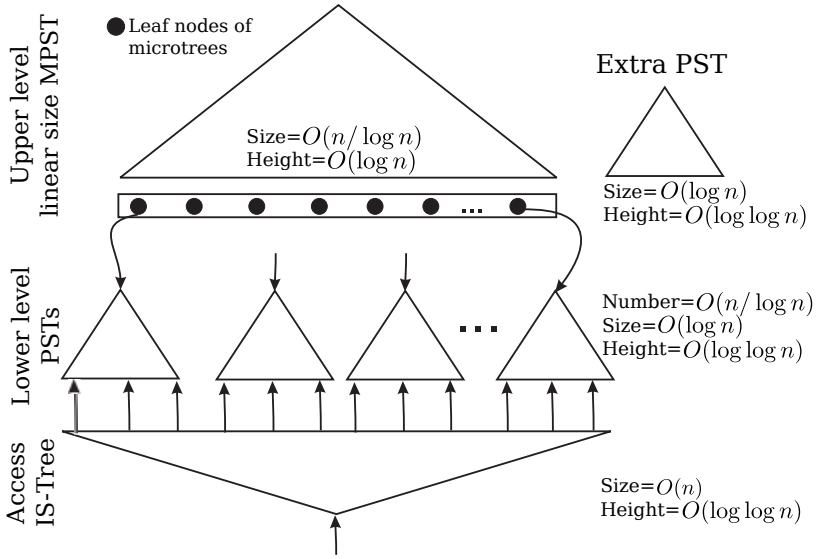

Figure 3. The internal memory solution for x-smooth and y-restricted distributions.

new MPST we spend $O(1)$ time worst case for each update of the epoch. As a result, the update operation is carried out in $O(\log\log n)$ time expected with high probability.

For the 3-sided query $[a,b]\times(-\infty,c]$, we first access the leaves of the lower level that contain a and b, through the IS-tree. This costs $O(\log\log n)$ time with high probability. Then the query proceeds bottom up in the standard way. First it traverses the buckets that contain a and b and then it accesses the MPST from the leaves of the buckets' representatives. Once the query reaches the node of the MPST with y-coordinate bigger than c, it continues top down to the respective buckets, which contain part of the answer, by following a single pointer from the nodes of the upper level MPST. Then we traverse top down these buckets and complete the set of points to report. Finally, we check the extra PST for reported points. The traversal of the MPST is charged on the size of the answer $O(t)$ and the traversal of the lower level costs $O(\log\log n)$ expected with high probability. Due to Theorem 3.7, the size of the extra PST is with high probability $O(\log n)$, thus the query spends $O(\log\log n)$ expected with high probability for it. Hence, in total the query time is $O(\log\log n + t)$.

Theorem 6.1. *There exists a dynamic main memory data structure that supports 3-sided queries in $O(\log\log n + t)$ expected time with high probability, can be updated in $O(\log\log n)$ expected time with high probability and consumes linear space, under the assumption that the x-coordinates are drawn from a (n^α, n^β)-smooth distribution, for arbitrary constants $0<\alpha,\beta<1$, and the y-coordinates are drawn from a restricted class of distributions.*

In order to extend the above structure to work in external memory we will follow a similar scheme with the above structure (Figure 3). We use an extra EPST and index the leaves of the main structure with and ISB-tree. This imposes that the x-coordinates are drawn from a $(n/(\log\log n)^{1+\epsilon}, n^{1/B})$-smooth distribution, for constant $\epsilon>0$, otherwise the search bound would not be expected to be doubly-logarithmic. Moreover, the main structure consists of three levels, instead of two (Figure 4. That is, we divide the n elements into $n'=\frac{n}{\log n}$ buckets of size $\log n$, which we implement as EPSTs (instead of PSTs). This will constitute the lower level of the whole structure. The n' representatives of the EPSTs are again divided into buckets of size $O(B)$, which constitute the middle level. The $n''=\frac{n'}{B}$ representatives are stored in the leaves of an external MPST (EMPST), which constitutes the upper level of the whole structure. In total, the space of the aforementioned structures is $O(n'+n''+n''\log^{(k)} n'') = O(\frac{n}{\log n}+\frac{n}{B\log n}+\frac{n}{B\log n}B) = O(\frac{n}{\log n}) = O(\frac{n}{B})$, where k is such that $\log^{(k)} n'' = O(B)$ holds.

The update algorithm is similar to the variant of internal memory. The query algorithm first proceeds bottom up. We locate the appropriate structures of the lower level in $O(\log_B \log n)$ I/Os with high probability, due to the assumption on the x-coordinates. The details for this procedure in the I/O model can be found in [22]. Note that if

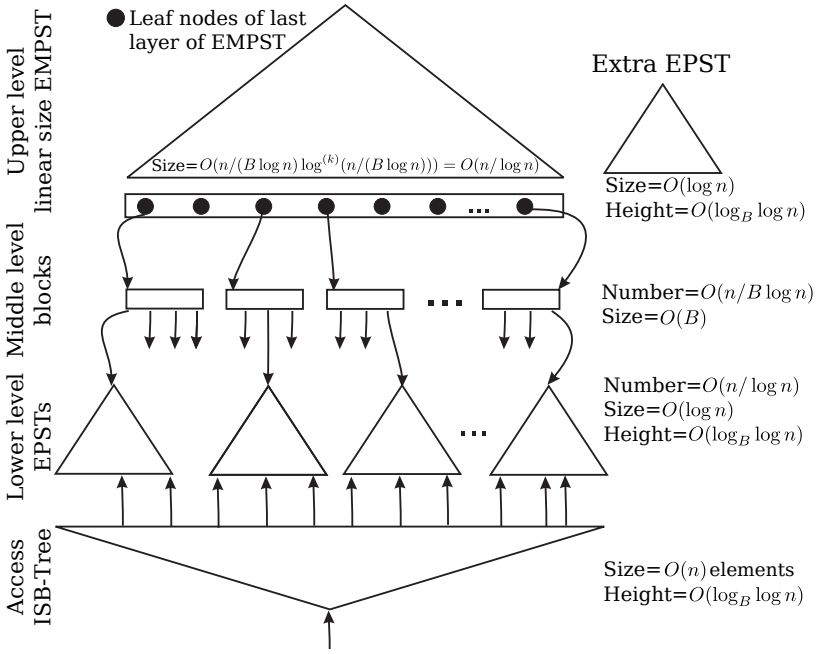

Figure 4. The external memory solution for x-smooth and y-restricted distributions.

we assume that the x-coordinates are drawn from the *grid distribution* with parameters $[1, M]$, then this access step can be realized in $O(1)$ I/Os. That is done by using an array A of size M as the access data structure. The position $A[i]$ keeps a pointer to the leaf with x-coordinate not bigger than i [34]. Then, by executing the query algorithm, we locate the at most two structures of the middle level that contain the representative leaves of the EPSTs we have accessed. Similarly we find the representatives of the middle level structures in the EMPST. Once we reached the node whose minimum y-coordinate is bigger than c, the algorithm continues top down. It traverses the EMPST and accesses the structures of the middle and the lower level that contain parts of the answer. The query time spent on the EMPST is $O(t/B)$ I/Os. All accessed middle level structures cost $O(2 + t/B)$ I/Os. The access on the lower level costs $O(\log_B \log n + t/B)$ I/Os. Hence, the total query time becomes $O(\log_B \log n + t/B)$ I/Os expected with high probability. We get that:

Theorem 6.2. *There exists a dynamic external memory data structure that supports 3-sided queries in $O(\log_B \log n + t/B)$ expected I/Os with high probability, can be updated in $O(\log_B \log n)$ expected I/Os with high probability and consumes $O(n/B)$ blocks of space, under the assumption that the x-coordinates are drawn from an $(n/(\log\log n)^{1+\varepsilon}, n^{1/B})$-smooth distribution, for constant $\varepsilon > 0$, and the y-coordinates are drawn from the restricted class of distributions.*

7. Conclusions

We considered the problem of answering 3-sided range queries of the form $[a, b] \times (-\infty, c]$ under sequences of insertions and deletions of points, trying to attain linear space and doubly-logarithmic expected with high probability operation complexities, under assumptions on the input distributions. We proposed three solutions, which we modified appropriately in order to work for the RAM and the I/O model. All of them consist of combinations of known data structures that support the 3-sided query operation, along with two novel structures introduced here.

The proposed solutions are practically implementable. Thus, we leave as a future work an experimental performance evaluation, in order to certify in practice the improved query performance and scalability of the proposed methods. Also, we leave as future work the extension of MPSTs to the Cache-Oblivious model.

References

[1] A. Aggarwal, S. Vitter, Jeffrey, The input/output complexity of sorting and related problems, Commun. ACM 31 (9) (1988) 1116–1127.

[2] S. Alstrup, G. Stølting Brodal, T. Rauhe, New data structures for orthogonal range searching, in: Proc. of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00), Computer Society, 2000, pp. 198–207.

[3] A. Andersson, Faster deterministic sorting and searching in linear space, in: Proc. IEEE FOCS, 1996, pp. 135–141.

[4] A. Andersson, C. Mattsson, Dynamic interpolation search in o(log log n) time, in: Proc. ICALP, Vol. 700 of Springer LNCS, 1993, pp. 15–27.

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

[6] L. Arge, V. Samoladas, J. S. Vitter, On two-dimensional indexability and optimal range search indexing, in: Proc. ACM SIGMOD-SIGACT-SIGART PODS, 1999, pp. 346–357.

[7] L. Arge, J. S. Vitter, Optimal dynamic interval management in external memory (extended abstract), in: Proc. IEEE FOCS, 1996, pp. 560–569.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02, ACM, New York, NY, USA, 2002, pp. 1–16.

[9] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, An asymptotically optimal multiversion B-tree, The VLDB Journal 5 (4) (1996) 264–275.

[10] G. Blackenagel, R. Gueting, XP-Trees - External priority search trees, in: Technical report, Fern University Hagen, Informatik-Bericht, Nr.92, 1990.

[11] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, J. S. Vitter, Efficient indexing methods for probabilistic threshold queries over uncertain data, in: Proceedings of the Thirtieth international conference on Very Large Data Dases - Volume 30, VLDB '04, VLDB Endowment, 2004, pp. 876–887.

[12] M. De Berg, M. Van Kreveld, M. Overmars, O. C. Schwarzkopf, Computational geometry, Springer, 2000.

[13] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: B. Chen, M. Paterson, G. Zhang (Eds.), Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, Vol. 4614 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2007, pp. 459–470.

[14] O. Fries, K. Mehlhorn, S. Näher, A. Tsakalidis, A log log n data structure for three-sided range queries, Inf. Process. Lett. 25 (4) (1987) 269–273.

[15] H. N. Gabow, R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, Journal of Computer and System Sciences 30 (2) (1985) 209 – 221.

[16] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surv. 30 (2) (1998) 170–231.

[17] D. Gusfield, Algorithms on strings, trees, and sequences: computer science and computational biology, Cambridge University Press, New York, NY, USA, 1997.

[18] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.

[19] C. Icking, R. Klein, T. Ottmann, Priority search trees in secondary memory (extended abstract), in: H. Göttler, H.-J. Schneider (Eds.), Graph-Theoretic Concepts in Computer Science, Vol. 314 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1988, pp. 84–93.

[20] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, J. S. Vitter, Indexing for data models with constraints and classes, in: Proc. ACM SIGACT-SIGMOD-SIGART PODS, 1993, pp. 233–243.

[21] P. Kanellakis, S. Ramaswamy, D. E. Vengroff, J. S. Vitter, Indexing for data models with constraints and classes, Journal of Computer and System Sciences 52 (3) (1996) 589 – 612.

[22] A. Kaporis, C. Makris, G. Mavritsakis, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, ISB-tree: A new indexing scheme with efficient expected behaviour, in: X. Deng, D.-Z. Du (Eds.), Algorithms and Computation, Vol. 3827 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2005, pp. 318–327.

[23] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, Improved bounds for finger search on a RAM, in: Proc. ESA, Vol. 2832 of Springer LNCS, 2003, pp. 325–336.

[24] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, Dynamic interpolation search revisited, in: Proc. ICALP, Vol. 4051 of Springer LNCS, 2006, pp. 382–394.

[25] N. Kitsios, C. Makris, S. Sioutas, A. Tsakalidis, J. Tsaknakis, B. Vassiliadis, 2-d spatial indexing scheme in optimal time, in: J. Stuller, J. Pokorny, B. Thalheim, Y. Masunaga (Eds.), Current Issues in Databases and Information Systems, Vol. 1884 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 107–116.

[26] D. Knuth, Deletions that preserve randomness, Software Engineering, IEEE Transactions on SE-3 (5) (1977) 351–359.

[27] K. G. Larsen, R. Pagh, I/O-efficient data structures for colored range and prefix reporting, in: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '12, SIAM, 2012, pp. 583–592.

[28] C. Levcopoulos, M. Overmars, A balanced search tree with O(1) worst-case update time, Acta Informatica 26 (3) (1988) 269–277.

[29] E. M. McCreight, Priority search trees, SIAM J. Comput. 14 (2) (1985) 257–276.

[30] K. Mehlhorn, A. Tsakalidis, Dynamic interpolation search, J. ACM 40 (3) (1993) 621–634.

[31] K. Mouratidis, S. Bakiras, D. Papadias, Continuous monitoring of top-k queries over sliding windows, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06, ACM, New York, NY, USA, 2006, pp. 635–646.

[32] M. H. Overmars, Efficient data structures for range searching on a grid, Journal of Algorithms 9 (2) (1988) 254 – 275.

[33] S. Ramaswamy, S. Subramanian, Path caching (extended abstract): a technique for optimal external searching, in: Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '94, ACM, New York, NY, USA, 1994, pp. 25–35.

[34] S. Sioutas, C. Makris, N. Kitsios, G. Lagogiannis, J. Tsaknakis, K. Tsichlas, B. Vassiliadis, Geometric retrieval for grid points in the RAM model., J. UCS 10 (9) (2004) 1325–1353.

[35] S. Subramanian, S. Ramaswamy, The P-range tree: a new data structure for range searching in secondary memory, in: Proceedings of the

sixth annual ACM-SIAM symposium on Discrete algorithms, SODA '95, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995, pp. 378–387.
[36] M. Thorup, Faster deterministic sorting and priority queues in linear space, in: Proc. ACM-SIAM SODA, 1998, pp. 550–555.
[37] J. S. Vitter, External memory algorithms and data structures: dealing with massive data, ACM Comput. Surv. 33 (2) (2001) 209–271.
[38] D. E. Willard, Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree, SIAM J. Comput. 29 (3) (2000) 1030–1049.