Dynamic 3-sided Planar Range Queries with Expected Doubly Logarithmic Time

Gerth Stølting Brodal[1], Alexis C. Kaporis[2], Spyros Sioutas[3], Konstantinos Tsakalidis[1], Kostas Tsichlas[4]

[1] MADALGO*, Department of Computer Science, Aarhus University, Denmark
{gerth,tsakalid}@madalgo.au.dk
[2] Computer Engineering and Informatics Department, University of Patras, Greece
kaporis@ceid.upatras.gr
[3] Department of Informatics, Ionian University, Corfu, Greece
sioutas@ionio.gr
[4] Department of Informatics, Aristotle University of Thessaloniki, Greece
tsichlas@csd.auth.gr

Abstract. We consider the problem of maintaining dynamically a set of points in the plane and supporting range queries of the type $[a, b] \times (-\infty, c]$. We assume that the inserted points have their x-coordinates drawn from a class of *smooth* distributions, whereas the y-coordinates are arbitrarily distributed. The points to be deleted are selected uniformly at random among the inserted points. For the RAM model, we present a linear space data structure that supports queries in $O(\log\log n + t)$ expected time with high probability and updates in $O(\log\log n)$ expected amortized time, where n is the number of points stored and t is the size of the output of the query. For the I/O model we support queries 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 using linear space, where B is the disk block size. The data structures are deterministic and the expectation is with respect to the input distribution.

1 Introduction

We consider the dynamic 3-sided range reporting problem in the plane. That is, to design a data structure that supports insertions and deletions of points, and supports range reporting queries of the type $[a, b] \times (-\infty, c]$, i.e. report all points contained in the query rectangle with one side unbounded. The more general *orthogonal range searching* problem finds applications in databases and is used as a subroutine for solving general geometric problems. A survey can be found at [1]. In particular, multidimensional instances can be decomposed into two dimensional subproblems, where 3-sided queries are of major importance [2].

Previous results. In the internal memory, the most commonly used data structure for supporting 3-sided queries is the *priority search tree* of McCreight [3].

* Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

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 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 RAM model, the only dynamic sublogarithmic bounds for this problem are due to Willard [4] 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. In the I/O model, Arge et al. [5] proposed an indexing scheme that consumes $O(n/B)$ space, supports updates in $O(\log_B n)$ amortized I/Os, and 3-sided range queries in $O(\log_B n + t/B)$ I/Os, where B denotes the block size. Both data structures pose no assumptions on the input distribution.

Our results. We consider the case where the x-coordinates of inserted points are drawn from a *smooth* probabilistic distribution, and the y-coordinates are arbitrarily distributed. Moreover, the deleted points are selected uniformly at random among the points in the data structure and queries can be adversarial. The assumption on the x-coordinates is broad enough to include distributions used in practice, such as uniform, regular and classes of non-uniform ones [6, 7].

We present two linear space data structures, for the RAM and the I/O model respectively. In the former model, we achieve a query time of $O(\log\log n + t)$ expected with high probability and update time of $O(\log\log n)$ expected amortized. In the latter model, the I/O complexity is $O(\log\log_B n + t/B)$ expected with high probability for the query and $O(\log_B \log n)$ expected amortized for the updates. In both cases, our data structures are deterministic and the expectation is derived from a probabilistic distribution of the x-coordinates, and an expected analysis of updates of points with respect to their y-coordinates.

2 Preliminaries

Weight Balanced Exponential Tree. 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 [8, 9, 10] 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* [11]. 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. 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 smallest weight. Processing the overflown nodes u bottom up guarantees that, during the split of u, its children satisfy their weight constraints.

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. Possibly, v needs to "split" again if its weight after the merge is more than $\frac{3}{2}w_i$ ("share"). In either case, the traversal continues upwards, which guarantees that the children of the underflown nodes satisfy their weight constraints. The following lemma, which is similar to [11, Lem. 9], holds.

Lemma 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. 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 $[\frac{5}{8}w_i, \frac{14}{8}w_i]$. $\square$

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 [12] the RMQ problem is solved in $O(1)$ time using $O(n)$ space and preprocessing time.

Dynamic External Memory 3-sided Range Queries for $O(B^2)$ Points. [5, Lem. 1] A set of $K \leq B^2$ points can be stored in $O(K/B)$ blocks, so that 3-sided queries need $O(t/B+1)$ I/Os and updates $O(1)$ I/Os, for output size t.

Smooth Distribution and Interpolation Search Structures. 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 (no "sharp peaks" exist).

Formally, given two functions f_1 and f_2, a density function $\mu = \mu[a,b](x)$ is (f_1, f_2)-*smooth* [13, 6] if there exists a constant β, such that for all c_1, c_2, c_3 where $a \leq c_1 < c_2 < c_3 \leq b$, and for all integers n and $\Delta = (c_3 - c_1)/f_1(n)$, it holds that $\int_{c_2-\Delta}^{c_2} \mu[c_1,c_3](x)dx \leq \frac{\beta \cdot f_2(n)}{n}$, when $\mu[c_1,c_3](x) = 0$ 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 $p = \int_{c_1}^{c_3} \mu(x)dx$.

The *IS-tree* [13, 14] is a dynamic data structure based on interpolation search that consumes linear space and can be updated in $O(1)$ time when the update position is given. Furthermore, the elements can be searched in $O(\log^2 n)$ worst case time, or $O(\log\log n)$ time expected with high probability when they are drawn from an $(n^\alpha, n^{1/2})$-smooth distribution, for constant $1/2<\alpha<1$. Its externalization, the *ISB-tree* [15], consumes linear space and can be updated in $O(1)$ I/Os for given update position. It supports searches in $O(\log_B n)$ I/Os worst case, or $O(\log_B \log n)$ I/Os expected with high probability when the elements are drawn from an $(n/(\log\log n)^{1+\varepsilon}, n^{1/B})$-smooth distribution, for constant $\varepsilon>0$.

3 The Internal Memory Data Structure

Our internal memory construction for storing n points in the plane consists of an IS-tree storing the points in sorted order with respect to the x-coordinates. 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 decreasing (*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 [13, 14], so is the total space.

3.1 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 2. *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. 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. 2 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 overlap 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 [13]. Hence the total query time is $O(\log \log n + t)$ expected with high probability.

3.2 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 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. 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 $O\big(\sum_{j=1}^{i+1} w_j/w_{j-1}\big) = O\big(\sum_{j=0}^{i} w_j^{c_2-1}\big) = O\big(w_i^{c_2-1}\big)$. □

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 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 [13, 14], under the same assumption for the x-coordinates.

Lemma 4. *Starting with an empty weight balanced exponential tree, the amortized time of rebalancing it due to insertions or deletions is $O(1)$.*

Proof. A sequence of n updates requires at most $O(n/w_i)$ rebalancings at level i (Lem. 1). Rebuilding the RMQ-structures after each rebalancing costs $O\big(w_i^{c_2-1}\big)$ time (Lem. 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. *The expected amortized time for inserting or deleting a point in a weight balanced exponential tree is $O(1)$.*

Proof. The insertion of a point creates a new leaf and thus T may rebalance, which by Lemma 4 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 $k/\frac{1}{2}w_i$. It follows that p ends up at the i-th ancestor or higher with probability at most $O\Big(\sum_{j=i+k}^{height(T)} \frac{2w_{j-1}}{\frac{1}{2}w_j} + \frac{k}{\frac{1}{2}w_i}\Big) = O\Big(\sum_{j=i+k}^{height(T)} w_{j-1}^{1-c_2} + \frac{k}{w_i}\Big) = O\Big(w_{i+k-1}^{1-c_2} + \frac{k}{w_i}\Big) = O\Big(w_i^{(1-c_2)c_2^{k-1}} + \frac{k}{w_i}\Big) = O\Big(\frac{1}{w_i}\Big)$ for $c_2 = 3/2$ and $k = 3$. Thus the expected cost of "swapping down" p becomes $O\Big(\sum_{i=1}^{height(T)} \frac{1}{w_i} \cdot \frac{w_{i+1}}{w_i}\Big) = O\Big(\sum_{i=1}^{height(T)} w_i^{c_2-2}\Big) = O\Big(\sum_{i=1}^{height(T)} c_1^{(c_2-2)c_2^i}\Big) = O(1)$ 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 1. *In the RAM model, using $O(n)$ space, 3-sided queries can be supported in $O(\log\log n + t/B)$ expected time with high probability, and updates in $O(\log\log n)$ time expected amortized, 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.*

4 The External Memory Data Structure

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 [15]. We implement every consecutive $\Theta(B^2)$ leaves of the ISB-Tree with the data structure of Arge et al. [5]. 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 [15]. 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 $O\Big(n + \sum_{i=1}^{height(T)} n \cdot B \cdot d_i^3 / w_i\Big) = O\Big(n + \sum_{i=1}^{height(T)} n \cdot B \big/ \big(B^{2 \cdot \frac{7}{6}^i}\big)^{\frac{4}{7}}\Big) = O(n)$ points, i.e. $O(n/B)$ disk blocks.

4.1 Querying the Data Structure

The query is similar to the internal memory construction. 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 [15]. 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 [5].

The list of B points in every node can be accessed in $O(1)$ I/Os. The construction of [5] 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.

4.2 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 [5]. 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 $\varepsilon{>}0$ [15].

Lemma 6. *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 $O\big(d_{i+1}^3\big)$ I/Os.*

Proof. Analogously to Lem. 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 $O\big(\sum_{j=1}^{i+1} d_j^3\big) = O\big(d_{i+1}^3\big)$ I/Os. □

Lemma 7. *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. We follow the proof of Lem. 4. Rebalancing a node at level i requires $O\big(d_{i+1}^3{+}B{\cdot}d_i^3\big)$ I/Os (Lem. 6), 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 $O\big(\sum_{i=1}^{height(T)} \frac{n}{w_i}{\cdot}(d_{i+1}^3{+}B{\cdot}d_i^3)\big){=}O\big(n{\cdot}\sum_{i=1}^{height(T)} w_i^{-1/2}{+}B{\cdot}w_i^{-4/7}\big){=}O(n)$. □

Lemma 8. *The expected amortized I/Os for inserting or deleting a point in an external weight balanced exponential tree is $O(1)$.*

Proof. By similar arguments as in Lem. 5 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 $O(d_{i+1}^3){=}O(w_i^{1/2})$ I/Os. Thus "swapping down" p costs $O\big(\sum_{i=1}^{height(T)} w_i^{1/2}{\cdot}\frac{B}{w_i}\big){=}O(1)$ expected I/Os. The same bound holds for deleting p, following similar arguments as in Lem. 5. □

Theorem 2. *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 $O(\log_B \log n)$ I/Os expected amortized, 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.*

References

[1] Agarwal, P., Erickson, J.: Geometric range rearching and its relatives. In Chazelle, B., Goodman, J., Pollack, R., eds.: Advances in Discrete and Computational Geometry. Contemporary Mathematics. American Mathematical Society Press (1999) 1–56

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

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

[4] Willard, D.E.: Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. SIAM J. Comput. **29**(3) (2000) 1030–1049

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

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

[7] Kaporis, A., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, C.: Improved bounds for finger search on a RAM. In: Proc. ESA. Volume 2832 of Springer LNCS. (2003) 325–336

[8] Andersson, A.: Faster deterministic sorting and searching in linear space. In: Proc. IEEE FOCS. (1996) 135–141

[9] Thorup, M.: Faster deterministic sorting and priority queues in linear space. In: Proc. ACM-SIAM SODA. (1998) 550–555

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

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

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

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

[14] Kaporis, A., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, C.: Dynamic interpolation search revisited. In: Proc. ICALP. Volume 4051 of Springer LNCS. (2006) 382–394

[15] Kaporis, A.C., Makris, C., Mavritsakis, G., Sioutas, S., Tsakalidis, A.K., Tsichlas, K., Zaroliagis, C.D.: ISB-tree: A new indexing scheme with efficient expected behaviour. In: Proc. ICALP. Volume 3827 of Springer LNCS. (2005) 318–327