Kinetic kd-Trees and Longest-Side kd-Trees*

Mohammad Ali Abam Mark de Berg Bettina Speckmann

Department of Mathematics and Computing Science,
TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{mabam, mdberg, speckman}@win.tue.nl

Abstract

We propose a simple variant of kd-trees, called rank-based kd-trees, for sets of points in $\mathbb{R}^d$. We show that a rank-based kd-tree, like an ordinary kd-tree, supports range search queries in $O(n^{1-1/d}+k)$ time, where k is the output size. The main advantage of rank-based kd-trees is that they can be efficiently kinetized: the KDS processes $O(n^2)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories, each event can be handled in $O(\log n)$ time, and each point is involved in $O(1)$ certificates.

We also propose a variant of longest-side kd-trees, called rank-based longest-side kd-trees (RBLS kd-trees, for short), for sets of points in $\mathbb{R}^2$. RBLS kd-trees can be kinetized efficiently as well and like longest-side kd-trees, RBLS kd-trees support nearest-neighbor, farthest-neighbor, and approximate range search queries in $O((1/\varepsilon)\log^2 n)$ time. The KDS processes $O(n^3 \log n)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories; each event can be handled in $O(\log^2 n)$ time, and each point is involved in $O(\log n)$ certificates.

Background. Due to the increased availability of GPS systems and to other technological advances, motion data is becoming more and more available in a variety of application areas: air-traffic control, mobile communication, geographic information systems, and so on. In many of these areas, the data are moving points in 2- or higher-dimensional space, and what is needed is to store these points in such a way that *range queries* ("Report all the points lying currently inside a query range") or *nearest-neighbor queries* ("Report the point that is currently closest to a query point") can be answered efficiently. Hence, there has been a lot of work on developing data structures for moving point data, both in the database community as well as in the computational-geometry community.

Within computational geometry, the standard model for designing and analyzing data structures for moving objects is the kinetic-data-structure framework introduced by Basch et al. [3]. A *kinetic data structure* (*KDS*) maintains a discrete attribute of a set of moving objects—the convex hull, for example, or the closest pair—where each object has a known motion trajectory. The basic idea is that although all objects move continuously there are only certain discrete moments in time when the combinatorial structure of the attribute—the ordered set of convex-hull vertices, or the pair that is closest—changes. A KDS contains a set of *certificates* that constitutes a proof that the maintained structure is correct. These certificates are inserted in a priority queue based on their time of expiration. The KDS then performs an event-driven simulation of the motion of the objects, updating the structure whenever an *event* happens, that is, when a certificate fails. Kinetic data structures and their accompanying maintenance algorithms can be evaluated and compared with respect to four desired characteristics. A good KDS is *compact* if it uses little space in addition to the input, *responsive* if the data structure invariants can be restored quickly after the failure of a certificate, *local* if it can be updated easily when the flight plan for an object changes, and *efficient* if the worst-case number of events handled by the data structure for a given motion is small compared to some worst-case number of "external events" that must be handled for that motion—see the surveys by Guibas [8, 9] for more details.

Related work. There are several papers that describe KDS's for the orthogonal range-searching problem, where the query range is an axis-parallel box. Basch et al. [4] kinetized d-dimensional range trees. Their KDS supports range queries in $O(\log^d n + k)$ time and uses $O(n \log^{d-1} n)$ storage. If the points follow constant-degree algebraic trajectories then their KDS processes $O(n^2)$ events; each event can be handled in $O(\log^{d-1} n)$ time. In the plane, Agarwal et al. [1] obtained an improved solution: their KDS supports orthogonal range-searching queries in $O(\log n + k)$ time, it uses $O(n \log n / \log\log n)$ storage, and the amortized cost of processing an event is $O(\log^2 n)$.

*M.A. was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 612.065.307. M.d.B. was supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 639.023.301.

Dagstuhl Seminar Proceedings 08081
Data Structures
http://drops.dagstuhl.de/opus/volltexte/2008/1530

Although these results are nice from a theoretical perspective, their practical value is limited for several reasons. First of all, they use super-linear storage, which is often undesirable. Second, they can perform only orthogonal range queries; queries with other types of ranges or nearest-neighbor searches are not supported. Finally, especially the solution by Agarwal et al. [1] is rather complicated. Indeed, in the setting where the points do not move, the static counterparts of these structures are usually not used in practice. Instead, simpler structures such as quadtrees, kd-trees, or bounding-volume hierarchies (R-trees, for instance) are used. In this paper we consider one of these structures, namely the kd-tree.

Kd-trees were initially introduced by Bentley [5]. A kd-tree for a set of points in the plane is obtained recursively as follows. At each node of the tree, the current point set is split into two equal-sized subsets with a line. When the depth of the node is even the splitting line is orthogonal to the x-axis, and when it is odd the splitting line is orthogonal to the y-axis. In d-dimensional space, the orientations of the splitting planes cycle through the d axes in a similar manner. Kd-trees use $O(n)$ storage and support orthogonal range searching queries in $O(n^{1-1/d} + k)$ time, where k is the number of reported points. Maintaining a standard kd-tree kinetically is not efficient. The problem is that a single event—two points swapping their order on x- or y-coordinate—can have a dramatic effect: a new point entering the region corresponding to a node could mean that almost the entire subtree must be re-structured. Hence, a variant of the kd-tree is needed when the points are moving.

Agarwal et al. [2] proposed two such variants for moving points in $\mathbb{R}^2$: the δ-pseudo kd-tree and the δ-overlapping kd-tree. In a δ-pseudo kd-tree each child of a node ν can be associated with at most $(1/2 + \delta)n_\nu$ points, where n_ν is the number of points in the subtree of ν. In a δ-overlapping kd-tree the regions corresponding to the children of ν can overlap as long as the overlapping region contains at most δn_ν points. Both kd-trees support orthogonal range queries in time $O(n^{1/2+\varepsilon} + k)$, where k is the number of reported points. Here ε is a positive constant that can be made arbitrarily small by choosing δ appropriately. These KDS's process $O(n^2)$ events if the points follow constant-degree algebraic trajectories. Although it can take up to $O(n)$ time to handle a single event, the amortized cost is $O(\log n)$ time per event. Neither of these two solutions is completely satisfactory: their query time is worse by a factor $O(n^\varepsilon)$ than the query time in standard kd-trees, there is only a good amortized bound on the time to process events, and only a solution for the 2-dimensional case is given. The goal of our paper is to developed a kinetic kd-tree variant that does not have these drawbacks.

Even though a kd-tree can be used to search with any type of range, there are only performance guarantees for orthogonal ranges. *Longest-side kd-trees*, introduced by Dickerson et al. [7], are better in this respect. In a longest-side kd-tree, the orientation of the splitting line at a node is not determined by the level of the node, but by the shape of its region: namely, the splitting line is orthogonal to the longest side of the region. Although a longest-side kd-tree does not have performance guarantees for exact range searching, it has very good worst-case performance for ε-approximate range queries, which can be answered in $O(\varepsilon^{1-d} \log^d n + k)$ time. (In an ε-approximate range query, points that are within distance $\varepsilon \cdot \text{diameter}(Q)$ of the query range Q may also be reported.) Moreover, a longest-side kd-tree can answer ε-approximate nearest-neighbor queries (or: farthest-neighbor queries) in $O(\varepsilon^{1-d} \log^d n)$ time. The second goal of our paper is to develop a kinetic variant of the longest-side kd-tree.

Our results. Our first contribution is a new and simple variant of the standard kd-tree for a set of n points in d-dimensional space. Our *rank-based kd-tree* supports orthogonal range searching in time $O(n^{1-1/d} + k)$ and it uses $O(n)$ storage—just like the original. But additionally it can be kinetized easily and efficiently. The rank-based kd-tree processes $O(n^2)$ events in the worst case if the points follow constant-degree algebraic trajectories[1] and each event can be handled in $O(\log n)$ worst-case time. Moreover, each point is involved only in a constant number of certificates. Thus we improve the both the query time and the event-handling time as compared to the planar kd-tree variants of Agarwal et al. [2], and in addition our results work in any fixed dimension.

Our second contribution is the first kinetic variant of the longest-side kd-tree, which we call the *rank-based longest-side kd-tree* (or *RBLS kd-tree*, for short), for a set of n points in the plane. (We have been unable to generalize this result to higher dimensions.) An RBLS kd-tree uses $O(n)$ space and supports approximate nearest-neighbor, approximate farthest-neighbor, and approximate range queries in the same time as the original longest-side kd-tree does for stationary points, namely $O((1/\varepsilon) \log^2 n)$ (plus the time needed to report the answers in case of range searching). The kinetic RBLS kd-tree maintains $O(n)$ certificates, processes $O(n^3 \log n)$ events if the points follow constant-degree algebraic trajectories[1],

[1]For the bound on the number of events in our rank-based kd-tree, it is sufficient that any pair of points swaps x- or y-order $O(1)$ times. For the bounds on the number of events in the RBLS kd-tree, we need that every two pairs of points define the same x- or y-distance $O(1)$ times.

each event can be handled in $O(\log^2 n)$ time, and each point is involved in $O(\log n)$ certificates.

1 Rank-based kd-trees

Let $\mathcal{P}$ be a set of n points in $\mathbb{R}^d$ and let us denote the coordinate-axes with $x_1, \ldots, x_d$. To simplify the discussion we assume that no two points share any coordinate, that is, no two points have the same x_1-coordinate, or the same x_2-coordinate, etc. (Of course coordinates will temporarily be equal when two points swap their order, but the description below refers to the time intervals in between such events.) In this section we describe a variant of a kd-tree for $\mathcal{P}$, the *rank-based kd-tree*. A rank-based kd-tree preserves all main properties of a kd-tree and, additionally, it can be kinetized efficiently.

Before we describe the actual rank-based kd-tree for $\mathcal{P}$, we first introduce another tree, namely the *skeleton* of a rank-based kd-tree, denoted by $\mathcal{S}(\mathcal{P})$. Like a standard kd-tree, $\mathcal{S}(\mathcal{P})$ uses axis-orthogonal splitting hyperplanes to divide the set of points associated with a node. As usual, the orientation of the axis-orthogonal splitting hyperplanes is alternated between the coordinate axes, that is, we first split with a hyperplane orthogonal to the x_1-axis, then with a hyperplane orthogonal to the x_2-axis, and so on. Let ν be a node of $\mathcal{S}(\mathcal{P})$. $h(\nu)$ is the splitting hyperplane stored at ν, axis(ν) is the coordinate-axis to which $h(\nu)$ is orthogonal, and $\mathcal{P}(\nu)$ is the set of points stored in the subtree rooted at ν. A node ν is called an *x_i-node* if axis(ν) $= x_i$ and a node ω is referred to as an *x_i-ancestor* of a node ν if ω is an ancestor of ν and axis(ω) $= x_i$. The first x_i-ancestor of a node ν (that is, the x_i-ancestor closest to ν) is the *x_i-parent(ν)* of ν.

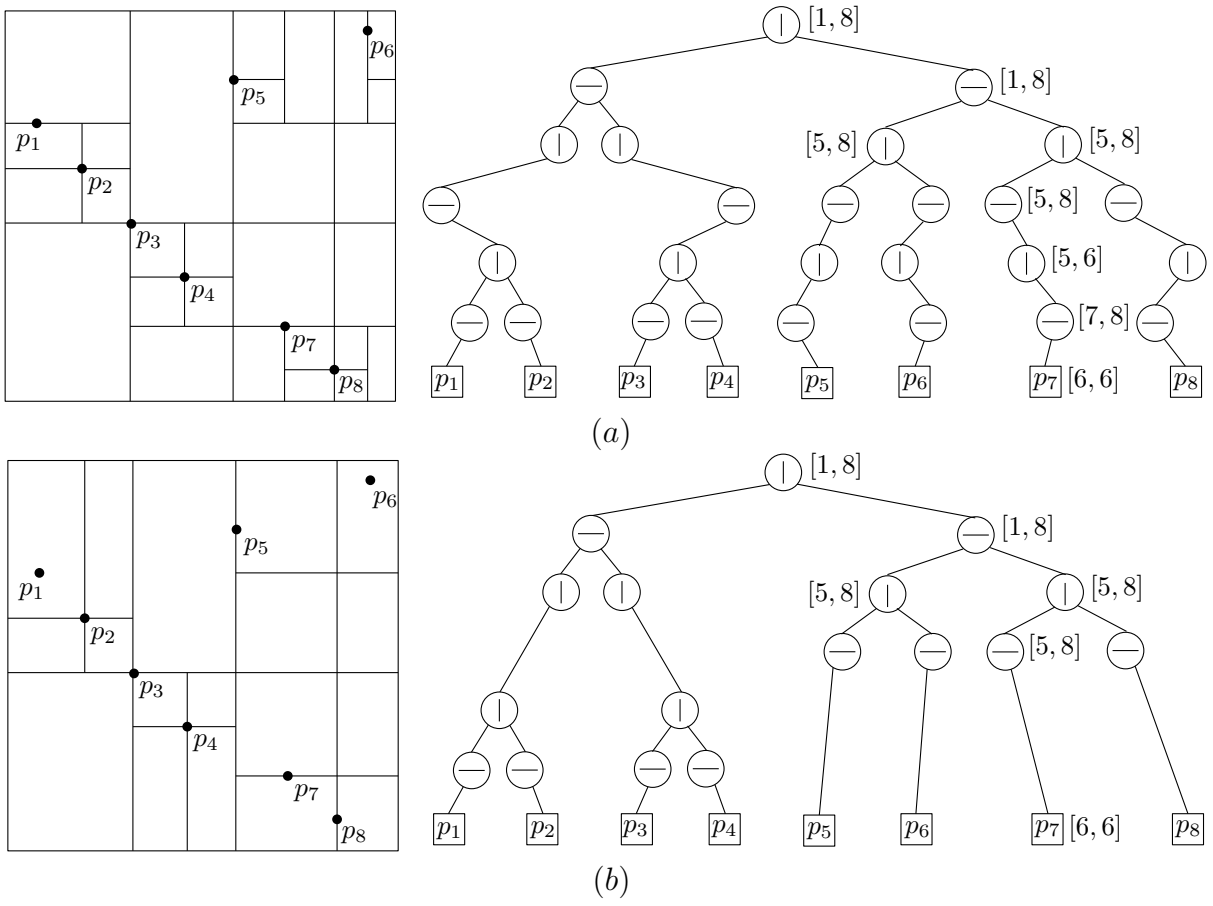

Figure 1: (a) The skeleton of a rank-based kd-tree and (b) the rank-based kd-tree itself. Note that points above (below) a horizontal splitting line go to the left subtree (right subtree)

A standard kd-tree chooses $h(\nu)$ such that $\mathcal{P}(\nu)$ is divided roughly in half. In contrast, $\mathcal{S}(\mathcal{P})$ chooses $h(\nu)$ based on a range of ranks associated with ν, which can have the effect that the sizes of the children of ν are completely unbalanced. We now explain this construction in detail. We use d arrays $\mathcal{A}_1, \ldots, \mathcal{A}_d$ to store

the points of $\mathcal{P}$ in d sorted lists; the array $\mathcal{A}_i[1,n]$ stores the sorted list based on the x_i-coordinate. As mentioned above, we associate a range $[r,r']$ of ranks with each node ν, denoted by range(ν), with $1 \leq r \leq r' \leq n$. Let ν be an x_i-node. If x_i-parent(ν) does not exist, then range(ν) is equal to $[1,n]$. Otherwise, if ν is contained in the left subtree of x_i-parent(ν), then range(ν) is equal to the first half of range(x_i-parent(ν)), and if ν is contained in the right subtree of x_i-parent(ν), then range(ν) is equal to the second half of range(x_i-parent(ν)). If range(ν) $= [r,r']$ then $\mathcal{P}(\nu)$ contains at most $r'-r+1$ points. We explicitly ignore all nodes (both internal as well as leaf nodes) that do not contain any points, they are not part of $\mathcal{S}(\mathcal{P})$, independent of their range of ranks. A node ν is a leaf of $\mathcal{S}(\mathcal{P})$ if range(ν) $= [j,j]$ for some j. Clearly a leaf contains exactly one point, but not every node that contains only one point is a leaf. (We could prune these nodes, which always have a range $[j,k]$ with $j < k$, but we chose to keep them in the skeleton for ease of description.) If ν is not a leaf and axis(ν) $= x_i$ then $h(\nu)$ is defined by the point whose rank in $\mathcal{A}_i$ is equal to the median of range(ν). (This is similar to the approach used in the kinetic BSP of [6].) It is not hard to see that this choice of the splitting plane $h(\nu)$ is equivalent to the following. Let region(ν) $= [a_1 : b_1] \times \cdots \times [a_d : b_d]$ and suppose for example that ν is an x_1-node. Then, instead of choosing $h(\nu)$ according to the median x_1-coordinate of all points in region(ν), we choose $h(\nu)$ according to the median x_1-coordinate of all points in the slab $[a_1,b_1] \times [-\infty:\infty] \times \cdots \times [-\infty:\infty]$.

We construct $\mathcal{S}(\mathcal{P})$ incrementally by inserting the points of $\mathcal{P}$ one by one. (Even though we proceed incrementally, we still use the rank of each point with respect to the whole point set, not with respect to the points inserted so far.) Let p be the point that we are currently inserting into the tree and let ν be the last node visited by p; initially $\nu = \text{root}(\mathcal{S}(\mathcal{P}))$. Depending on which side of $h(\nu)$ contains p we select the appropriate child ω of ν to be visited next. If ω does not exist, then we create it and compute range(ω) as described above. We recurse with $\nu = \omega$ until range(ν) $= [j,j]$ for some j. We always reach such a node after $d \log n$ steps, because the length of range(ν) is a half of the length of range(x_i-parent(ν)) and depth(ν) $=$ depth(x_i-parent(ν)) $+ d$ for an x_i-node ν. Figure 1(a) illustrates $\mathcal{S}(\mathcal{P})$ for a set of eight points. Since each leaf of $\mathcal{S}(\mathcal{P})$ contains exactly one point of $\mathcal{P}$ and the depth of each leaf is $d \log n$, the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$.

Lemma 1 *The depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$ and the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$ for any fixed dimension d. $\mathcal{S}(\mathcal{P})$ can be constructed in $O(n \log n)$ time.*

A node $\nu \in \mathcal{S}(\mathcal{P})$ is *active* if and only if both its children exist, that is, both its children contain points. A node ν is *useful* if it is either active, or a leaf, or its first $d-1$ ancestors contain an active node. Otherwise a node is *useless*. We derive the rank-based kd-tree for $\mathcal{P}$ from the skeleton by pruning all useless nodes from $\mathcal{S}(\mathcal{P})$. The parent of a node ν in the rank-based kd-tree is the first unpruned ancestor of ν in $\mathcal{S}(\mathcal{P})$. Roughly speaking, in the pruning phase every long path whose nodes have only one child each is shrunk to a path whose length is less than d. The rank-based kd-tree has exactly n leaves and each contains exactly one point of $\mathcal{P}$. Moreover, every node ν in the rank-based kd-tree is either active or it has an active ancestor among its first $d-1$ ancestors. The rank-based kd-tree derived from Figure 1(a) is illustrated in Figure 1(b).

Lemma 2

(i) A rank-based kd-tree on a set of n points in $\mathbb{R}^d$ has depth $O(\log n)$ and size $O(n)$.

(ii) Let ν be an x_i-node in a rank-based kd-tree. In the subtree rooted at a child of ν, there are at most 2^{d-1} x_i-nodes ω such that x_i-parent(ω) $= \nu$.

(iii) Let ν be an x_i-node in a rank-based kd-tree. On every path starting at ν and ending at a descendant of ν and containing at least $2d-1$ nodes, there is an x_i-node ω such that x_i-parent(ω) $= \nu$.

Proof.

(i) A rank-based kd-tree is at most as deep as its skeleton $\mathcal{S}(\mathcal{P})$. Since the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$ by Lemma 1, the depth of a rank-based kd-tree is also $O(\log n)$. To prove the second claim, we charge every node that has only one child to its first active ancestor. Recall that each active node has two children. We charge at most $2(d-1)$ nodes to each active node, because after pruning there is no path in the rank-based kd-tree whose length is at least d and in which all nodes have one child. Therefore, to bound the size of the rank-based kd-tree it is sufficient to bound the number of active nodes. Let $\mathcal{T}$ be a tree containing all active nodes and all leaves of the rank-based kd-tree.

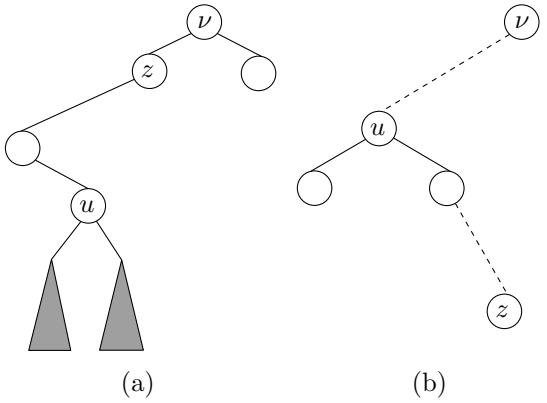

Figure 2: Illustration for the proof of Lemma 2.

A node ν is the parent of a node ω in $\mathcal{T}$ if and only if ν is the first active ancestor of ω in the rank-based kd-tree. Obviously, $\mathcal{T}$ is a binary tree with n leaves where each internal node has two children. Hence, the size of $\mathcal{T}$ is $O(n)$ and consequently the size of the rank-based kd-tree is $O(n)$.

(ii) To simplify notation, let ω' denote the node in $\mathcal{S}(\mathcal{P})$ that corresponds to a node ω in the rank-based kd-tree. Let z be a child of ν and and let u be the first active node in the subtree rooted at z as depicted in Fig. 2(a), that is, u is the highest active node in the subtree rooted at z. Note that the definition of active node ensures that u is unique, and note that u can be z. Now assume x_i-parent$(\omega) = \nu$ where ω is an x_i-node in the subtree rooted at z. If ω is not a node in the subtree rooted at u, then there is just one node ω in the subtree rooted at z satisfying x_i-parent$(\omega) = \nu$, since every node between z and u has only one child. This means that we are done. Otherwise, if ω is a node in the subtree rooted at u, then ω' must be in the subtree rooted at u' of $\mathcal{S}(\mathcal{P})$. Let s' be the first x_i-node on the path from u' to ω'. Because one of any d consecutive nodes in $\mathcal{S}(\mathcal{P})$ uses a hyperplane orthogonal to the x_i-axis as a splitting plane, depth$(s') \leq$ depth$(u') + d - 1$. Since u' is active and depth$(s') \leq$ depth$(u') + d - 1$, the node s' must appear as a node, s, in the rank-based kd-tree. This and the assumption that x_i-parent$(\omega) = \nu$ imply $\omega = s$ which means depth$(\omega') \leq$ depth$(u') + d - 1$. Hence the number of nodes ω is at most 2^{d-1}.

(iii) Let u be the first active node on the path starting at ν and ending at a descendant z of ν and containing at least $2d - 1$ nodes as depicted in Fig. 2(b). Because there is no path in the rank-based kd-tree that contains d nodes such that every node in the path has only one child, depth$(u) \leq$ depth$(\nu)+d-1$ which implies depth$(z) \geq$ depth$(u)+d-1$—note that on the path from ν to z there are $2d - 1$ nodes. Let ω' be the first x_i-node in the path starting at u' and ending at z' in $\mathcal{S}(\mathcal{P})$. Because one of any d consecutive nodes in $\mathcal{S}(\mathcal{P})$ uses a hyperplane orthogonal to the x_i-axis to split points, and depth$(z') \geq$ depth$(u') + d - 1$, the node ω' exists. The node ω' must appear as a node, ω, in the kd-tree, because either $\omega' = u'$ or among the first $d - 1$ ancestor of ω' there is an active ancestor, namely u'. Putting it all together we can conclude that depth$(\omega) \leq$ depth$(\nu) + 2d - 2$ which implies the claim.

□

The region associated with a node ν, denoted by region(ν), is the maximal volume bounded by the splitting hyperplanes stored at the ancestors of ν. More precisely, the region associated with the root of a rank-based kd-tree is simply the whole region, and the region corresponding to the right child of a node ν is the maximal subregion of region(ν) on the right side of $h(\nu)$ and the region corresponds to the left child of ν is the rest of region(ν) (for an appropriate definition of left and right in d dimensions). A point p is contained in $\mathcal{P}(\nu)$ if and only if p lies in region(ν). Like a kd-tree, a rank-based kd-tree can be used to report all points inside a given orthogonal range search query—the reporting algorithm is exactly the same. At first sight, the fact that the splits in our rank-based kd-tree can be very unbalanced may seem to have a big, negative impact on the query time. Fortunately this is not the case. To prove this, we next bound the number of cells intersected by an axis-parallel plane h. As for normal kd-trees, this is immediately gives a bound on the total query time.

Lemma 3 *Let h be a hyperplane orthogonal to the x_i-axis for some i. The number of nodes in a rank-based kd-tree whose regions are intersected by h is $O(n^{1-1/d})$.*

Proof. Imagine a dummy node μ with $\text{axis}(\mu) = x_i$ as the parent of the root. We charge every node ν whose region is intersected by h to x_i-parent(ν). Thanks to μ, x_i-parent(ν) exists for every node of the tree and hence every node is indeed charged to an x_i-node. Lemma 2(iii) implies $\text{depth}(\nu) \leq \text{depth}(x_i\text{-parent}(\nu)) + 2d - 2$ which implies that at most 2^{2d-2} nodes are charged to each x_i-node. Therefore it is sufficient to bound the number of x_i-nodes whose regions are intersected by h.

Let $\mathcal{T}$ be the tree containing all x_i-nodes in the rank-based kd-tree and let $\mathcal{T}'$ be the tree containing all x_i-nodes in the skeleton $\mathcal{S}(\mathcal{P})$. A node ν is the parent of a node ω in $\mathcal{T}$ if and only if $x_i\text{-parent}(\omega) = \nu$ in the rank-based kd-tree; the equivalent definition holds for $\mathcal{T}'$. According to Lemma 2(ii), every node ν in $\mathcal{T}$ has at most 2^d children and each side of $h(\nu)$ contains the regions corresponding to at most 2^{d-1} children of ν. Note that the dummy node has at most 2^{d-1} children in total. Let $\mathcal{T}^*$ be yet another tree containing all nodes in $\mathcal{T}$ whose regions are intersected by h. Since h is parallel to $h(\nu)$ for every node ν of $\mathcal{T}$, it can intersect only the regions that lie to one side of $h(\nu)$. Hence every node of $\mathcal{T}^*$ has at most 2^{d-1} children. The idea behind the proof is to consider a top part of $\mathcal{T}^*$ consisting of $n^{1-1/d}$ nodes of $\mathcal{T}^*$, and then argue that all subtree below this top part together contain $n^{1-1/d}$ nodes as well. Next we make this idea precise.

Let $\text{TOP}(\mathcal{T}^*)$ be a tree containing all nodes of $\mathcal{T}^*$ whose depths in $\mathcal{T}^*$ are at most $\lfloor (1/d)\log n \rfloor$, and let $\nu_1, \ldots, \nu_c$ be the leaves of $\text{TOP}(\mathcal{T}^*)$ whose depthes are exactly $\lfloor (1/d)\log n \rfloor$. Clearly c is at most $(2^{d-1})^{(1/d)\log n} = n^{1-1/d}$ and hence the size of $\text{TOP}(\mathcal{T}^*)$ is at most $2n^{1-1/d}$. Let $\nu'_1, \ldots, \nu'_c$ be the nodes corresponding to $\nu_1, \ldots, \nu_c$ in $\mathcal{T}'$. Furthermore, let $u'_1, \ldots, u'_m$ be the distinct nodes in $\mathcal{T}'$ at depth $\lfloor (1/d)\log n \rfloor$ such that every u'_k has at least one node ν'_j as descendant and every ν'_j has a node u'_k as an ancestor—note that due to pruning the depth of ν'_j can be larger than $\lfloor (1/d)\log n \rfloor$. Because the nodes ν'_j are disjoint, we have $\sum_1^c |\mathcal{P}(\nu'_j)| \leq \sum_1^m |\mathcal{P}(u'_k)|$.

Let U_k be the set of splitting hyperplanes stored in the ancestors of u'_k in $\mathcal{T}'$. Recall that all nodes u'_k are x_i-nodes whose regions are intersected by h. Furthermore, all nodes u'_k have the same depth in $\mathcal{T}'$. Together this implies that $U_k = U_l$ for all $1 \leq k, l \leq m$ because their x_i-ranges must be the same. Let h_1 be the last hyperplane in U_k on the left side of region(u'_1) and let h_2 be the first hyperplane in U_k on the right side of region(u'_1). Because $U_k = U_l$ for all $1 \leq k, l \leq m$, all regions u'_k are bounded by h_1 and h_2. We know that range(u'_k) contains $n/2^{(1/d)\log n} = n^{1-1/d}$ ranks, hence there are at most $n^{1-1/d}$ points inside the region bounded by h_1 and h_2. Since the nodes u'_k are disjoint and the region bounded by h_1 and h_2 contains $n^{1-1/d}$ points, we have $\sum_1^m |\mathcal{P}(u'_k)| \leq n^{1-1/d}$ which implies $\sum_1^c |\mathcal{P}(\nu_j)| = \sum_1^c |\mathcal{P}(\nu'_j)| \leq n^{1-1/d}$.

Finally, let $f(n)$ denote the number of x_i-nodes whose regions are intersected by h. We have $f(n) = |\text{TOP}(\mathcal{T}^*)| + \sum_1^c f(|\mathcal{P}(\nu_j)|)$. Since $f(|\mathcal{P}(\nu_j)|) \leq |\mathcal{P}(\nu_j)|$, $\sum_1^c |\mathcal{P}(\nu_j)| \leq n^{1-1/d}$, and $|\text{TOP}(\mathcal{T}^*)| \leq 2n^{1-1/d}$, we can conclude that $f(n) = O(n^{1-1/d})$. □

The following theorem summarizes our results.

Theorem 4 *A rank-based kd-tree for a set $\mathcal{P}$ of n points in d dimensions uses $O(n)$ storage and can be built in $O(n \log n)$ time. An orthogonal range search query on a rank-based kd-tree takes $O(n^{1-1/d} + k)$ time where k is the number of reported points.*

Remark. The time complexity based on d of a range query in a rank-based kd-tree is $O(d^2 n^{1-1/d} + k)$ while in a standard kd-tree, this is $O(dn^{1-1/d} + k)$. To prove the claim it suffices to show that the number of nodes in a rank-based kd-tree whose regions are intersected by a hypeplane h being orthogonal to x_i-axis is $O(dn^{1-1/d})$. To show this, for each leaf ν in the rank-based kd-tree, if $\text{axis}(\text{parent}(\nu)) \neq x_i$, we imagine a dummy node as the parent of ν and the child of the real parent of ν such that whose axis is x_i. We can simply show that Lemma 2 is still true and also Lemma 3 where we show that the number of x_i-node intersected by h is $O(n^{1-1/d})$ is still true. In the proof of Lemma 3, we charge every node ν whose region is intersected by h to x_i-parent(ν). Let ω be a x_i node. A node ν is charged to ω if and only if it belongs to the subtree rooted at ω and ended at x_i-nodes whose x_i-parents are ω. Since there is no path in the rank-based kd-tree whose nodes have just one child and whose length is at least d, we can simply show that the size of the subtree is at most d times the number of x_i-nodes which are the leaves of the subtree and whose regions are intersected by h. This simply implies the number of nodes whose regions are intersected by h is $O(dn^{1-1/d} + k)$.

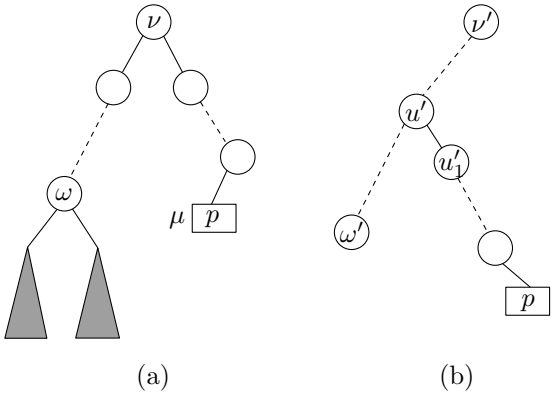

Figure 3: Deleting and inserting point p.

The KDS. We now describe how to kinetize a rank-based kd-tree for a set of continuously moving points $\mathcal{P}$. The combinatorial structure of a rank-based kd-tree depends only on the ranks of the points in the arrays $\mathcal{A}_i$, that is, it does not change as long as the order of the points in the arrays $\mathcal{A}_i$ remains the same. Hence it suffices to maintain a certificate for each pair p and q of consecutive points in every array $\mathcal{A}_i$, which fails when p and q change their order. Now assume that a certificate, involving two points p and q and the x_i-axis, fails at time t. To handle the event, we simply delete p and q and re-insert them in their new order. (During the deletion and re-insertion there is no need to change the ranks of the other points.) These deletions and insertions do not change anything for the other points, because their ranks are not influenced by the swap and the deletion and re-insertion of p and q. Hence the rank-based kd-tree remains unchanged except for a small part that involves p and q. A detailed description of this "small part" can be found below.

Deletion. Let ν be the first active ancestor of the leaf μ containing p—see Figure 3(a). The leaf μ and all nodes on the path from μ to ν must be deleted, since they do not contain any points anymore (they only contained p and p is now deleted). Furthermore, ν stops being active. Let ω be the first active descendent of ν if it exists and otherwise let ω be the leaf whose ancestor is ν. There are at most d nodes on the path from ν to ω. Since ν is not active anymore, any of the nodes on this path might become useless and hence have to be deleted.

Insertion. Let ν be the highest node in the rank-based kd-tree such that its region contains p and the region corresponding to its only child ω does not contain p—note that p cannot reach a leaf when we re-insert p, because the range of a leaf is $[j, j]$ for some j and there cannot be two points in this range. Let ν' and ω' be the nodes in $\mathcal{S}(P)$ corresponding to ν and ω. Let u' be the lowest node on the path from ν' to ω' whose region contains both region(ω') and p as illustrated in Figure 3(b)—note that we do not maintain $\mathcal{S}(\mathcal{P})$ explicitly but with the information maintained in ν and ω the path between ν' and ω' can be constructed temporarily. Because u' will become an active node, it must be added to the rank-based kd-tree and also every node on the path from u' to ω' must be added to the rank-based kd-tree if they are useful. From u', the point p follows a new path $u'_1, \ldots, u'_k$ which is created during the insertion. All first $d-1$ nodes in the list $u'_1, \ldots, u'_k$ and the leaf u'_k must be added to the rank-based kd-tree—note that range$(u'_k) = [j, j]$ for some j.

Theorem 5 *A kinetic rank-based kd-tree for a set $\mathcal{P}$ of n moving points in d dimensions uses $O(n)$ storage and processes $O(n^2)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories. Each event can be handled in $O(\log n)$ time and each point is involved in $O(1)$ certificates.*

2 Rank-based longest-side kd-trees

Longest-side kd-trees are a variant of kd-trees that choose the orientation of the splitting hyperplane for a node ν according to the shape of the region associated with ν, always splitting the longest side

first. Dickerson et al. [7] showed that a longest-side kd-tree can be used to answer the following queries quickly:

$(1+\varepsilon)$**-nearest neighbor query:** For a set $\mathcal{P}$ of points in $\mathbb{R}^d$, a query point $q \in \mathbb{R}^d$, and $\varepsilon > 0$, this query returns a point $p \in \mathcal{P}$ such that $d(p,q) \leq (1+\varepsilon)d(p^*,q)$, where p^* is the true nearest neighbor to q and $d(\cdot,\cdot)$ denotes the Euclidean distance.

$(1-\varepsilon)$**-farthest neighbor query:** For a set $\mathcal{P}$ of points in $\mathbb{R}^d$, a query point $q \in \mathbb{R}^d$, and $\varepsilon > 0$, this query returns a point $p \in \mathcal{P}$ such that $d(p,q) \geq (1-\varepsilon)d(p^*,q)$, where p^* is the true farthest neighbor to q.

ε**-approximate range search query:** For a set $\mathcal{P}$ of points in $\mathbb{R}^d$, a query region Q with diameter D_Q, and $\varepsilon > 0$, this query returns (or counts) a set $\mathcal{P}'$ such that $\mathcal{P} \cap Q \subset \mathcal{P}' \subset \mathcal{P}$ and for every point $p \in \mathcal{P}'$, $d(p,Q) \leq \varepsilon D_Q$.

The main property of a longest-side kd-tree—which is used to bound the query time—is that the number of disjoint regions associated with its nodes and intersecting at least two opposite sides of a hypercube $\mathcal{C}$ is bounded by $O(\log^{d-1} n)$. It seems difficult to directly kinetize a longest-side kd-tree. Hence, using similar ideas as in the previous section, we introduce a simple variation of 2-dimensional longest-side kd-trees, so called *ranked-based longest-side kd-trees* (RBLS kd-trees, for short). An RBLS kd-tree does not only preserve all main properties of a longest-side kd-tree but it can be kinetized easily and efficiently. As in the previous section we first describe another tree, namely the skeleton of an RBLS kd-tree denoted by $\mathcal{S}(\mathcal{P})$. We then show how to extract an RBLS kd-tree from the skeleton $\mathcal{S}(\mathcal{P})$ by pruning.

We recursively construct $\mathcal{S}(\mathcal{P})$ as follows. We again use two arrays $\mathcal{A}_1$ and $\mathcal{A}_2$ to store the points of $\mathcal{P}$ in two sorted lists; the array $\mathcal{A}_i[1,n]$ stores the sorted list based on the x_i-coordinate. Let the points in $\mathcal{P}$ be inside a box, which is the region associated with the root, and let ν be a node whose subtree must be constructed; initially $\nu = \text{root}(\mathcal{S}(\mathcal{P}))$. If $\mathcal{P}(\nu)$ contains only one point, then the subtree is just a single leaf, i.e, ν is a leaf of $\mathcal{S}(\mathcal{P})$. (Note that this is slightly different from the previous section.) If $\mathcal{P}(\nu)$ contains more than one point, then we have to determine the proper splitting line. Let the longest side of region(ν) be parallel to the x_i-axis. We set axis(ν) to be x_i. If x_i-parent(ν) does not exist, then we set range(ν) $= [1,n]$. Otherwise, if ν is contained in the left subtree of x_i-parent(ν), then range(ν) is equal to the first half of range(x_i-parent(ν)), and if ν is contained in the right subtree of x_i-parent(ν), then range(ν) is equal to the second half of range(x_i-parent(ν)). The splitting line of ν, denoted by $l(\nu)$, is orthogonal to axis(ν) and specified by the point whose rank in $\mathcal{A}_i$ is the median of range(ν). If there is a point of $\mathcal{P}(\nu)$ on the left side of $l(\nu)$ (on the right side of $l(\nu)$ or on $l(\nu)$), a node is created as the left child (the right child) of ν. The points of $\mathcal{P}(\nu)$ which are on the left side of $l(\nu)$ are associated with the left child of ν, the remainder is associated with the right child of ν. The region of the right child is the maximal subregion of region(ν) on the right side of $l(\nu)$ and the region of the left child is the rest of region(ν).

Lemma 6 *The depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$, the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$, and $\mathcal{S}(\mathcal{P})$ can be constructed in $O(n \log n)$ time.*

Proof. Assume for contradiction that the depth of a leaf ν is at least $2\log n + 1$. Now consider the path from the root to ν. Because there are only two distinct axes, there are at least $\log n + 1$ nodes on this path whose axes are the same, for example x_i. Let $\nu_1, \ldots, \nu_k$ be these nodes. Since $|\text{range}(\nu_{j+1})| \leq \lceil (1/2)|\text{range}(\nu_j)| \rceil$ $(j = 1, \ldots, k-1)$ and $k > \log n$, ν_k must be empty, which is a contradiction. Hence the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$.

Since each leaf contains exactly one point and the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$, the size of $\mathcal{S}(\mathcal{P})$ is $O(n \log n)$. Furthermore it is easy to see that it takes $O(|\mathcal{P}(\nu)|)$ time to split the points at a node ν. Hence we spend $O(n)$ time at each level of $\mathcal{S}(\mathcal{P})$ during construction, for a total construction time of $O(n \log n)$. □

The following lemma shows that RBLS kd-trees preserve the main property of longest-side kd-trees, which is used to bound the query time.

Lemma 7 *Let $\mathcal{C}$ be any square, and let N be any set of nodes whose regions are pairwise disjoint and such that these regions all intersect two opposite sides of $\mathcal{C}$. Then $|N| = O(\log n)$.*

Proof. Dickerson et al. [7] showed that a longest-side kd-tree on a set of points in $\mathbb{R}^2$ has this property. Their proof uses only two properties of a longest side kd-tree: (*i*) the depth of a longest-side kd-tree is $O(\log n)$ and (*ii*) the longest side of a region is split first. Since an RBLS kd-tree has these two properties, their proof simply applies. □

As in the previous section, we obtain our structure by pruning useless nodes from $\mathcal{S}(\mathcal{P})$. It will be convenient to alter the definition of useful nodes slightly, as follows. A node ν is useful if ν is a leaf, or an active node, or $l(\nu)$ defines one of the sides of the boundary of region(ω) where ω is an active descendant of ν. Otherwise ν is useless. An RBLS kd-tree is obtained from $\mathcal{S}(\mathcal{P})$ by pruning useless nodes. The parent of a node ν in the RBLS kd-tree is the first unpruned ancestor of ν in $\mathcal{S}(\mathcal{P})$. The following lemma shows that an RBLS kd-tree has linear size and that it preserves the main property of a longest-side kd-tree.

Theorem 8

(i) An RBLS longest-side kd-tree on a set of n points in $\mathbb{R}^2$ has depth $O(\log n)$ and size $O(n)$.

(ii) The number of nodes in an RBLS longest-side kd-tree whose regions are disjoint and that intersect at least two opposite sides of a square $\mathcal{C}$ is $O(\log n)$.

Proof.

(i) An RBLS kd-tree is at most as deep as its skeleton $\mathcal{S}(\mathcal{P})$. Since the depth of $\mathcal{S}(\mathcal{P})$ is $O(\log n)$ by Lemma 6, the depth of an RBLS kd-tree is also at most $O(\log n)$. To prove the second claim, we first show that there is no path containing five nodes such that every node on the path has only one child. Assume for contradiction that there is such a path from ν to one of its descendants ω. Because there are only two distinct axes, there must be three nodes u_1, u_2, and u_3 on this path using the same axis. Clearly at most two of $l(u_1)$, $l(u_2)$, and $l(u_3)$ can define one of the sides of the boundary of any region associated with a descendant of ω. Therefore, at least one of u_1, u_2, and u_3 must be useless, which is a contradiction. We now charge every node that has only one child to its first active ancestor. Because there is no path containing five nodes such that every node on the path has only one child, we charge at most eight nodes to each active node. Since the number of active nodes is linear, the size of an RBLS longest-side kd-tree is $O(n)$.

(ii) Let L be a set of nodes in an RBLS kd-tree whose regions are disjoint and that intersect at least two opposite sides of a square $\mathcal{C}$. We define a set L' of nodes as follows. Consider a node $\nu \in L$. If ν is active then we add ν to L'. If ν is not active, then we consider the first active ancestor u of ν. We add the child w of u to L' that is on the path from u to ν (note that w could be ν). The regions in L' are disjoint and we have $|L| = |L'|$. Since the region associated with a node is a subregion of the region associated with its ancestor, the regions associated with the nodes in L' intersect at least two opposite sides of $\mathcal{C}$. Let ν' be the corresponding node to ν in $\mathcal{S}(\mathcal{P})$. The definition of a useful node implies region(ν) = region(ν') for every active node ν—note that this may be false for other nodes. Thus, if $\nu \in L'$ is active, then region(ν) = region(ν') and if ν is a child of an active node ω, then region(ν) = region(u') where u' is the child of ω' that is on the path from ω' to ν'. Thus, for every node ν in L', there is a node ω' in $\mathcal{S}(\mathcal{P})$ such that region(ν) = region(ω'). This observation together with Lemma 7 shows that $|L'| = O(\log n)$ which implies $|L| = O(\log n)$.

□

Using an RBLS kd-tree, similar algorithms to the algorithms of Dickerson et al. [7] can be used to answer $(1+\varepsilon)$-nearest neighbor, $(1-\varepsilon)$-farthest neighbor and ε-approximate range search queries.

Theorem 9 *An RBLS kd-tree for a set of n points in the plane supports $(1+\varepsilon)$-nearest or $(1-\varepsilon)$-farthest neighbor queries in $O((1/\varepsilon)\log^2 n)$ time. Moreover, for any constant-complexity convex region and any constant-complexity non-convex region a counting (or reporting) ε-approximate range search query can be performed in time $O((1/\varepsilon)\log^2 n)$ and $O((1/\varepsilon^2)\log^2 n)$, respectively (plus the output size in the reporting case).*

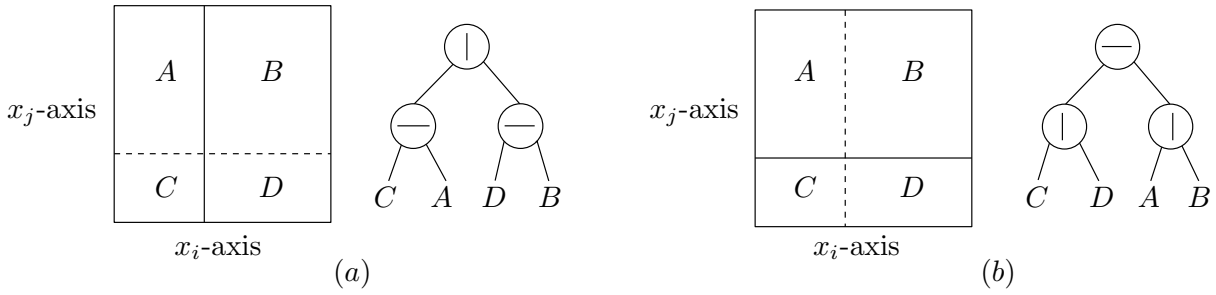

Figure 4: The status of the RBLS kd-tree before handling a longest-side event and after handling the event.

The KDS. We now describe how to kinetize a RBLS kd-tree for a set of continuously moving points $\mathcal{P}$. Clearly the combinatorial structure of an RBLS kd-tree changes only when one of the following two events occurs.

Ordering event: Two points change their ordering on one of the coordinate-axes.

Longest-side event: A side of a region starts to be the longest side of that region.

We first describe how to detect these events, then we explain how to handle them. Ordering events can be easily detected. We maintain a certificate for each pair p and q of consecutive points in the two arrays $\mathcal{A}_1$ and $\mathcal{A}_2$, which fails when p and q change their order.

Longest-side events are a bit tricky to detect efficiently. An easy way would be to maintain a certificate $s_1(\nu) < s_2(\nu)$ (or $s_2(\nu) < s_1(\nu)$) for each node ν in $\mathcal{S}(\mathcal{P})$ where $s_i(\nu)$ denotes the length of the x_i-side of region(ν). Let $x_i(p)$ denote the x_i-coordinate of p. We have $s_i(\nu) = x_i(p) - x_i(q)$ where p and q are two points specifying two splitting lines in the x_i-ancestors of ν in $\mathcal{S}(\mathcal{P})$. More precisely, the splitting lines defined by p and q are associated with the first *left ancestor* and the first *right ancestor* of ν in $\mathcal{S}(\mathcal{P})$, that is, the first nodes u and w such that ν is a left child of u and a right child of w. The problem with this approach lies in the fact that $x_i(p) - x_i(q)$ can be the side length of a linear number of regions and hence our KDS would not be local. It would also not be responsive, because if two points change their ordering we might have to update a linear number of longest-side certificates.

We avoid these problems by not maintaining a separate longest-side certificate for every region of the RBLS kd-tree. Instead, we identify all pairs of points that can define either the vertical or the horizontal side length of a region. We add all these pairs to one single list, the so-called *side-length list* which is sorted on the length of the sides. A longest-side event can happen only when two adjacent elements in the side-length list define the same length. (More precisely, they also have to define both a vertical and a horizontal side—nothing happens if two vertical sides have the same length. In fact, even when a vertical side and a horizontal side get the same length, it is possible that nothing happens, because they need not be sides of the same region.) So we have to maintain a certificate for each pair of consecutive elements in the side-length list. It remains to explain which sides precisely appear in the side-length list. To determine this, we construct two one-dimensional rank-based kd-trees $\mathcal{T}_i$ on the x_i-coordinates of the points in $\mathcal{P}$. Since all splitting lines for the nodes of $\mathcal{T}_i$ are orthogonal to the x_i-axis, $\mathcal{T}_i$ is in fact a balanced binary search tree. Let ν be a node in $\mathcal{T}_i$ and let ν_r and ν_ℓ be the first right and the first left ancestors of ν in $\mathcal{T}_i$. If p and q are the two points used in ν_r and ν_ℓ as splitting points, then $x_i(p) - x_i(q)$ appears in the side-length list. Since the number of nodes in $\mathcal{T}_i$ is $O(n)$ and a node can be either the first left ancestor or the first right ancestor of at most $O(\log n)$ nodes, the number of elements in the side-length list is $O(n)$ and each point is involved in $O(\log n)$ elements of the side-length list. Moreover, all sides of all regions in $\mathcal{S}(\mathcal{P})$ exist in the side-length list.

Ordering event. When handling an ordering event that involves two points p and q and the x_i-axis, we have to update $\mathcal{A}_i$, the side-length list and the RBLS kd-tree. We update the array $\mathcal{A}_i$ by swapping p and q and updating the at most three certificates in which p and q are involved. We update the side-length list by replacing p by q and vice versa and computing the failure times of all certificates affected by these replacements. To quickly find in which elements of the side-length list a point p is involved we maintain for each rank i a list of elements of the side-length list in which rank i is involved. Since the number

of elements in the side-length list is $O(n)$ and two ranks are involved in each element, this additional information uses $O(n)$ space. Since each rank is involved in $O(\log n)$ elements of the side-length list, updating the side-length list takes $O(\log n)$ time and inserting the failures times of the new certificates into the event queue takes $O(\log^2 n)$. To update the RBLS kd-tree, we first delete p and q from the RBLS kd-tree and then we re-insert them in their new order.

Deletion. Let ν be the lowest active node whose region contains p. The leaf containing p is a child of ν. This leaf must be removed. Let ω be the first active ancestor of ν. All nodes on the path from ω to ν must be checked whether they are useless. If so, they must be removed from the RBLS kd-tree.

Insertion. Let ν be the highest node in the RBLS kd-tree whose region contains p and such that the region corresponding to its only child ω does not contain p. Let ν' and ω' be the nodes in $\mathcal{S}(P)$ corresponding to ν and ω. Let u' be the lowest node on the path from ν' to ω' whose region contains both region(ω') and p as illustrated in Figure 3(b)— note that we do not explicitly maintain $\mathcal{S}(\mathcal{P})$ but the path between ν' and ω' can be constructed temporarily in $O(\log n)$ time. Because u' will become active, it must be added as a node, u, to the RBLS kd-tree and also every node on the path from ν' to u' must be added to the RBLS kd-tree if they are useful. The point p is maintained in a leaf whose parent is u.

Longest-side event. When handling a longest-side event that occurs at time t we first update the side-length list and the certificates involved in the event. Then we update the RBLS kd-tree as follows. Let p, q, p', and q' be the points involved in the event, more precisely, let $x_i(p(t)) - x_i(q(t)) = x_j(p'(t)) - x_j(q'(t))$. If $i = j$, then there is nothing to do, because the certificate failure can not correspond to a real longest-side event. Otherwise, we need to determine which, if any, of the regions of $\mathcal{S}(\mathcal{P})$ corresponds to the event. Because two sides of the region are given, we can follow a path from the root to some node while temporally constructing each node from $\mathcal{S}(\mathcal{P})$ on the path which does not appear in the RBLS kd-tree. If there is no region with the two given sides, then we delete the temporary nodes and stop handling the event.

Otherwise there is exactly one region in $\mathcal{S}(\mathcal{P})$ that is specified by the two sides that triggered the event. (Note that this is only true in two dimensions, in higher dimensions the boundary of many regions can be defined by two sides—this is the only problem when attempting to extend these results to higher dimensions.) Let ν be the node that is associated with the event region. We add the two children ν_r and ν_ℓ of ν in $\mathcal{S}(\mathcal{P})$ to the RBLS kd-tree provided that they do not already exist in the RBLS kd-tree. Let the x_i-side of region(ν) be bigger than the x_j-side of region(ν) at the point in time just before t, denoted by t^-. At time t^-, $l(\nu)$ must be orthogonal to the x_i-axis and $l(\nu_\ell)$ and $l(\nu_r)$ must be orthogonal to the x_j-axis as illustrated in Figure 4(a)—note that region(ν) is a square at time t. Moreover, $l(\nu_\ell) = l(\nu_r)$, because the median of all points between the two x_i-sides of region(ν) is chosen to specify $l(\nu_\ell)$ and $l(\nu_r)$. Let A, B, C, and D be the four regions defined by $l(\nu)$, $l(\nu_\ell)$ and $l(\nu_r)$ as illustrated in Figure 4(a). We now split region(ν) with a line that is orthogonal to the x_j-axis and region(ν_r) and region(ν_ℓ) with a line that is orthogonal to the x_i-axis. Clearly $l(\nu)$ at time t is equal to $l(\nu_\ell)$ and $l(\nu_r)$ at time t^- and $l(\nu_\ell)$ and $l(\nu_r)$ at time t are equal to $l(\nu)$ at time t^-. The four subregion A, B, C, and D do not change and we only have to put them in the correct positions in the RBLS kd-tree as illustrated in Figure 4(b). Finally every node on the path from the root to ν as well as ν_r and ν_ℓ must be checked whether they are useless. If so, they must be removed from the RBLS kd-tree.

The number of events. Assume that the points in $\mathcal{P}$ follow constant-degree algebraic trajectories. Clearly the number of ordering events is $O(n^2)$. To count the number of longest-side events, we charge a longest-side event in which two sides s_1 and s_2 are involved to the side (either s_1 or s_2) that appeared in the side-length list later. At any point in time there are $O(n)$ elements in the side-length list and elements are only added or deleted whenever a ordering event occurs. During each ordering event, $O(\log n)$ elements can be added to the side-length list. All longest-side events that involve one of these "new" elements and one of the "old" elements are charged to one of the new elements, hence a total of $O(n \log n)$ events is charged to the new elements that are created during one ordering event. Since there are $O(n^2)$ ordering events, the number of longest-side events is $O(n^3 \log n)$. (This bound subsumes events that involve two new elements or two of the initial elements of the side-length list.)

Theorem 10 *A kinetic RBLS kd-tree for a set $\mathcal{P}$ of n moving points in $\mathbb{R}^2$ uses $O(n)$ storage and processes $O(n^3 \log n)$ events in the worst case, assuming that the points follow constant-degree algebraic trajectories. Each event can be handled in $O(\log^2 n)$ time and each point is involved in $O(\log n)$ certificates.*

3 Conclusions

We presented a variant of kd-tress, called rank-based kd-trees, for sets of points in $\mathbb{R}^d$. We showed that our rank-based kd-tree supports orthogonal range searching in $O(n^{1-1/d} + k)$ time and it uses $O(n)$ storage—just like the original. But additionally it can be kinetized easily and efficiently. In the dynamic setting, either inserting or deleting a point affects the ranks of points which may cause a dramatic change in the rank-based kd-tree. A challenging problem is how to adapt the rank-based kd-tree to the insertion and deletion of points such that the query time does not change asymptotically.

We also proposed a variant of longest-side kd-trees, called rank-based longest-side kd-trees, for sets of points in $\mathbb{R}^2$. We showed RBLS kd-trees can be kinetized efficiently as well and like longest-side kd-trees, RBLS kd-trees support nearest-neighbor, farthest-neighbor, and approximate range search queries in $O((1/\varepsilon) \log^2 n)$ time. Unfortunately we have been unable to generalize this result to higher dimension. We leave it as an interesting open problem for future research.

References

[1] P. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Journal of Computer and System Sciences*, 66(1):207-243, 2003.

[2] P. Agarwal, J. Gao, and L. Guibas. Kinetic medians and kd-trees. In *Proc. 10th European Symposium on Algorithms*, pages 5–16, Lecture Notes in Computer Science 2461, Springer Verlag, 2002.

[3] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31:1–28, 1999.

[4] J. Basch, L. Guibas, and L. Zhang. Proximity problems on moving points. In *Proc. 13th Symposium on Computational Geometry*, pages 344–351, 1997.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[6] M. de Berg and J. Comba and L. J. Guibas. A segment-tree based kinetic BSP. In *Proc. 17th Symposium on Computational Geometry*, pages 134–140, 2001.

[7] M. Dickerson, C. A. Duncan, and M. T. Goodrich. K-d trees are better when cut on the longest side. In *Proc. 8th European Symposium on Algorithms*, pages 179–190, Lecture Notes in Computer Science 1879, Springer Verlag, 2000.

[8] L. Guibas. Kinetic data structures: A state of the art report. In *Proc. 3rd Workshop on Algorithmic Foundations of Robotics*, pages 191–209, 1998.

[9] L. Guibas. Motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. CRC Press, 2nd edition, 2004.