Kinetic kd-trees and longest-side kd-trees

SIAM J. COMPUT.
Vol. 39, No. 4, pp. 1219–1232

KINETIC kd-TREES AND LONGEST-SIDE kd-TREES*

MOHAMMAD ALI ABAM†, MARK DE BERG‡, AND BETTINA SPECKMANN‡

Abstract. We propose a simple variant of kd-trees, called *rank-based kd-trees*, for sets of n points in $\mathbb{R}^d$. We show that a rank-based kd-tree, like an ordinary kd-tree, supports orthogonal range 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 kinetic data structure (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*, for sets of points in $\mathbb{R}^2$. Rank-based longest-side kd-trees can be kinetized efficiently as well, and like longest-side kd-trees, they support ε-approximate nearest-neighbor, ε-approximate farthest-neighbor, and ε-approximate range queries with convex ranges 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.

Key words. computational geometry, kinetic data structures, kd-trees, range searching

AMS subject classification. 68U05

DOI. 10.1137/070710731

1. Introduction. *Background.* Due to the increased availability of global positioning 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 two- 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 (KDS) framework introduced by Basch, Guibas, and Hershberger [3]. A 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 proof that the maintained structure is correct. These certificates are inserted into a priority queue based on their time of expiration. The KDS then performs an

*Received by the editors December 12, 2007; accepted for publication (in revised form) June 5, 2009; published electronically September 9, 2009.
http://www.siam.org/journals/sicomp/39-4/71073.html

†MADALGO, Department of Computer Science, Aarhus University, Denmark (abam@madalgo.au.dk). This author's research was supported by the Netherlands' Organization for Scientific Research (NWO) under project 612.065.307 and the MADALGO Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

‡Department of Computing Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, the Netherlands (mdberg@win.tue.nl, speckman@win.tue.nl). The second author's research was supported by the Netherlands' Organization for Scientific Research (NWO) under project 639.023.301.

event-driven simulation of the motion of the objects, updating the structure whenever an *event* happens, that is, when a certificate fails. KDSs and their accompanying maintenance algorithms can be evaluated and compared with respect to four desired characteristics. A 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 KDSs for the orthogonal range-searching problem, where the query range is an axis-parallel box. Basch, Guibas, and Zhang [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, Arge, and Erickson [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 $O(\log^2 n)$.

Although these results are nice from a theoretical perspective, their practical value is limited for several reasons. First of all, they use superlinear 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, Arge, and Erickson [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 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 restructured. Hence, a variant of the kd-tree is needed when the points are moving.

Agarwal, Gao, and Guibas [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 v can be associated with at most $(1/2+\delta)n_v$ points, where n_v is the number of points in the subtree of v. In a δ-overlapping kd-tree, the regions corresponding to the children of v can overlap as long as the overlapping region contains at most δn_v 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 KDSs 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 two-dimensional case is given. Another possibility is to use a dynamic kd-tree, that is, a kd-tree that supports insertions and deletions of points. Then, whenever the projections of two points on one of the axes swap, we delete and reinsert those points. The best known bounds for dynamic kd-trees are obtained using the so-called divided kd-trees of Van Kreveld and Overmars [10], which have $O(n^{1-1/d}\log^{1/d} n + k)$ query time and $O(\log n)$ update time. (Note that the splitting hyperplanes in a divided kd-tree do not cycle through the d different orientations as one descends down the tree, so in this sense it is not a "real" kd-tree.) This solution is not completely satisfactory, since the query time is nonoptimal: it is worse by a factor $O(\log^{1/d} n)$ than the query time in standard kd-trees. Also, obtaining worst-case update times is fairly complicated. The goal of our paper is to develop 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, Duncan, and Goodrich [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: 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 for any constant-complexity convex range Q. (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.[1] 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 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 both the query time and the event-handling time as compared to the planar kd-tree variants of Agarwal, Gao, and Guibas [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*, for a set of n points in the plane. (We have been unable to generalize this result to higher dimensions.) A rank-based longest-side kd-tree uses $O(n)$ space and supports ε-approximate nearest-neighbor, ε-approximate farthest-neighbor, and ε-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 rank-based longest-side kd-tree maintains $O(n)$ certificates, processes $O(n^3 \log n)$ events if 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.

[1] To be more precise, the query time, space, and update time in our rank-based kd-trees are $O(d^2 n^{1-1/d} + dk)$, $O(dn)$, and $O(d\log n)$, respectively, while in the standard kd-tree, they are $O(dn^{1-1/d} + dk)$, $O(dn)$, and $O(\log n)$, respectively.

2. 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 v be a node of $\mathcal{S}(\mathcal{P})$. We denote the splitting hyperplane stored at v by $h(v)$; we denote the coordinate-axis to which $h(v)$ is orthogonal by $\text{axis}(v)$ and the set of points stored in the subtree rooted at v by $\mathcal{P}(v)$. A node v is called an *x_i-node* if $\text{axis}(v) = x_i$, and a node w is referred to as an *x_i-ancestor* of a node v if w is an ancestor of v and $\text{axis}(w) = x_i$. The first x_i-ancestor of a node v (that is, the x_i-ancestor closest to v) is the *x_i-parent*(v) of v.

Each node v in $\mathcal{S}(\mathcal{P})$—or in any kd-tree, for that matter—can be associated with a region, denoted $\text{region}(v)$, which is bounded by splitting hyperplanes stored at ancestors of v. This region is defined as follows. The region associated with the root of a rank-based kd-tree is the entire space. The region corresponding to the right child of a node v is the part of $\text{region}(v)$ lying to the positive side of, or on, the hyperplane $h(v)$; the region corresponding to the left child of v is the part of $\text{region}(v)$ to the negative side of $h(v)$. A point $p \in \mathcal{P}$ is contained in $\mathcal{P}(v)$ if and only if p lies in $\text{region}(v)$.

A standard kd-tree chooses $h(v)$ such that $\mathcal{P}(v)$ is divided roughly in half. In contrast, $\mathcal{S}(\mathcal{P})$ chooses $h(v)$ based on a range of ranks associated with v, which can have the effect that the sizes of the children of v 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 v, denoted by $\text{range}(v)$, and defined as follows.

Let v be an x_i-node. If x_i-$\text{parent}(v)$ does not exist, then $\text{range}(v)$ is equal to $[1, n]$. Otherwise, if v is contained in the left subtree of x_i-$\text{parent}(v)$, then $\text{range}(v)$ is equal to the first half of $\text{range}(x_i\text{-parent}(v))$, and if v is contained in the right subtree of x_i-$\text{parent}(v)$, then $\text{range}(v)$ is equal to the second half of $\text{range}(x_i\text{-parent}(v))$. If $\text{range}(v) = [r, r']$, then $\mathcal{P}(v)$ 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 v is a leaf of $\mathcal{S}(\mathcal{P})$ if $|\mathcal{P}(v)| = 1$. If v is not a leaf and $\text{axis}(v) = x_i$, then $h(v)$ is defined by the point whose rank in $\mathcal{A}_i$ is equal to the median of $\text{range}(v)$. (This is similar to the approach used in the kinetic binary space partition of [6].) It is not hard to see that this choice of the splitting plane $h(v)$ is equivalent to the following. Let $\text{region}(v) = [a_1 : b_1] \times \cdots \times [a_d : b_d]$ and suppose, for example, that v is an x_1-node. Then, instead of choosing $h(v)$ according to the median x_1-coordinate of all points in $\text{region}(v)$, we choose $h(v)$ according to the median x_1-coordinate of all points in the slab $[a_1, b_1] \times [-\infty : \infty] \times \cdots \times [-\infty : \infty]$.

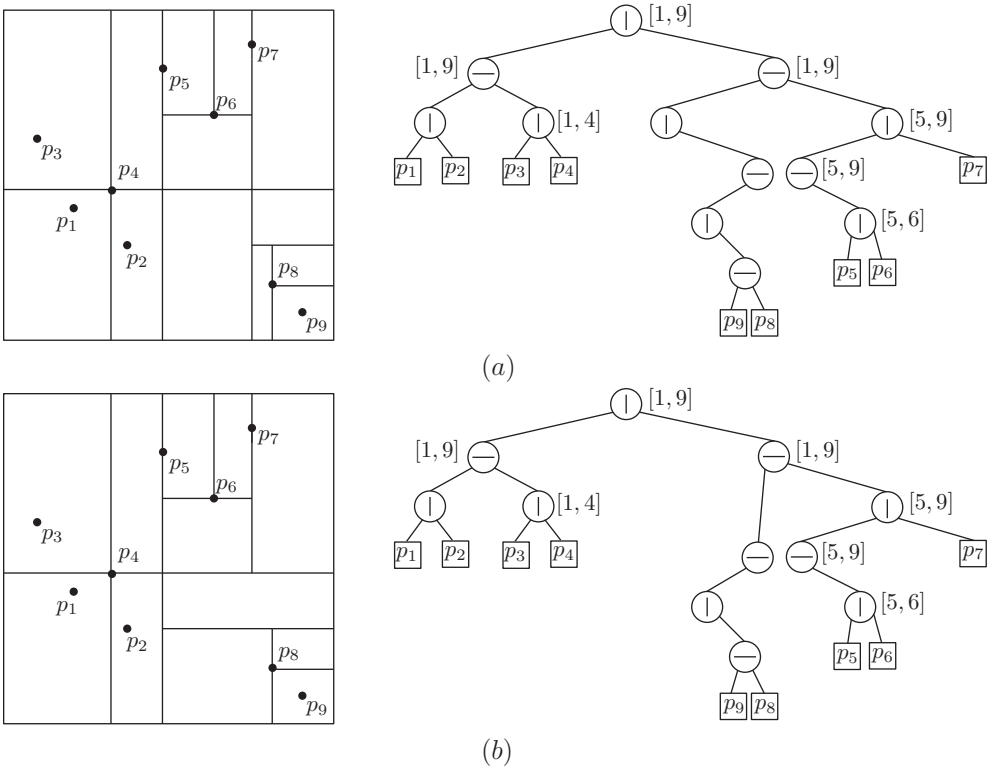

FIG. 1. (a) *The skeleton of a rank-based kd-tree, and* (b) *the rank-based kd-tree itself. Note that points above (below) a horizontal splitting line go to the right subtree (left subtree). The vertical (or horizontal) bar in a node indicates which axis is used to split the point set. The ranges of some nodes are also specified.*

We recursively construct $\mathcal{S}(\mathcal{P})$ as follows. Let the points in $\mathcal{P}$ be inside a box, which is the region associated with the root, and let v be a node whose subtree must be constructed; initially $v = \text{root}(\mathcal{S}(\mathcal{P}))$ and $\text{axis}(v) = x_1$. If $\mathcal{P}(v)$ contains only one point, then v is a leaf of $\mathcal{S}(\mathcal{P})$. If $\mathcal{P}(v)$ contains more than one point, we determine the range(v) and axis(v) as described above. Suppose $\text{axis}(v) = x_i$. $h(v)$, the splitting hyperplane of v, is orthogonal to axis(v) and specified by the point whose rank in $\mathcal{A}_i$ is the median of range(v). If there is a point of $\mathcal{P}(v)$ on the left side of $h(v)$ (resp., on the right side of $h(v)$ or on $h(v)$), a node is created as the left child (resp., the right child) of v. The points of $\mathcal{P}(v)$ which are on the left side of $h(v)$ are associated with the left child of v; the remainder is associated with the right child of v. The recursion ends when $|\mathcal{P}(v)| = 1$, which happens after at most $d\lceil \log n \rceil$ steps, because the length of range(v) is a half of the length of range$(x_i\text{-parent}(v))$ and $\text{depth}(v) = \text{depth}(x_i\text{-parent}(v)) + d$ for an x_i-node v. Figure 1(a) illustrates $\mathcal{S}(\mathcal{P})$ for a set of nine points. Since each leaf of $\mathcal{S}(\mathcal{P})$ contains exactly one point of $\mathcal{P}$ and the depth of each leaf is $O(d \log n)$, the size of $\mathcal{S}(\mathcal{P})$ is $O(dn \log n)$. Furthermore, it is easy to see that it takes $O(|\mathcal{P}(v)|)$ time to split the points at a node v. Hence we spend $O(n)$ time at each level of $\mathcal{S}(\mathcal{P})$ during construction for a total construction time of $O(dn \log n)$.

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

Next we explain how the rank-based kd-tree $\mathcal{T}(\mathcal{P})$ is obtained from the skeleton $\mathcal{S}(\mathcal{P})$.

We call a node $v \in \mathcal{S}(\mathcal{P})$ *active* if and only if both its children exist, that is, the regions of both its children contain points. We call a node v of $\mathcal{S}(\mathcal{P})$ *useful* if at least one of the following holds:

- The node v is active.
- The node v is a leaf of $\mathcal{S}(\mathcal{P})$.
- The node v is not active and not a leaf—that is, it is a degree-one node—but its splitting hyperplane $h(v)$ defines a facet of region(w), where w is v's highest active descendant.

A node that is not useful is called *useless*. We derive the rank-based kd-tree from the skeleton by pruning all useless nodes from $\mathcal{S}(\mathcal{P})$. Another way to describe the pruning process is as follows. Consider a path of degree-one nodes, and let v be the node immediately below this path. Then we prune all nodes of the path, except the at-most $2d$ nodes whose splitting planes contain a facet of the region of v. The rank-based kd-tree, which we denote by $\mathcal{T}(P)$, has exactly n leaves, and each contains exactly one point of $\mathcal{P}$. The rank-based kd-tree derived from Figure 1(a) is illustrated in Figure 1(b).

LEMMA 2.

(i) *A rank-based kd-tree $\mathcal{T}(\mathcal{P})$ on a set $\mathcal{P}$ of n points in $\mathbb{R}^d$ has depth $O(d \log n)$, size $O(dn)$, and can be constructed in $O(dn \log n)$ time.*

(ii) *Let v be an active node in $\mathcal{S}(\mathcal{P})$. Then the region corresponding to v in $\mathcal{T}(\mathcal{P})$ is the same as the region corresponding to v in $\mathcal{S}(\mathcal{P})$.*

Proof.

(i) $\mathcal{T}(\mathcal{P})$ is at most as deep as its skeleton $\mathcal{S}(\mathcal{P})$. Thus the bound on the depth of $\mathcal{T}(\mathcal{P})$ follows from Lemma 1.
Since $\mathcal{T}(\mathcal{P})$ has n leaves, it has $n-1$ active nodes. We charge the remaining nodes—these are degree-one nodes—to the first active node below them. This way any active node gets charged at most $2d$ degree-one nodes. Hence, the total size of $\mathcal{T}(\mathcal{P})$ is $O(dn)$, as claimed.
Constructing $\mathcal{S}(\mathcal{P})$ takes $O(dn \log n)$ time, and pruning the useless nodes can then easily be done in linear time in the size of $\mathcal{S}(\mathcal{P})$, which finishes the proof of part (i).

(ii) Consider an active node v. We must prove that none of the nodes w defining a facet of region(v) in $\mathcal{S}(\mathcal{P})$ is pruned. We will prove this by induction on the depth of v in $\mathcal{S}(\mathcal{P})$.
If the depth is zero, then v is the root of $\mathcal{S}(P)$, and the statement trivially holds. Now consider a node v at depth greater than zero, and let w be a node such that $h(w)$ defines a facet of region(v) in $\mathcal{S}(P)$. Assume w is not active; otherwise, it will certainly not be pruned. Let u be the lowest active node that is an ancestor of v. If w lies on the path from u to v, then v is w's highest active descendant, and so w will not be pruned. Otherwise, w is an ancestor of u. Since region$(v) \subset$ region(u), this implies that w defines a facet of u. Hence, by induction, w will not be pruned. □

Like a kd-tree, a rank-based kd-tree can be used to report all points inside a given axis-aligned box—the reporting algorithm is exactly the same. At first sight, the fact that the splits in our rank-based kd-tree $\mathcal{T}(\mathcal{P})$ can be very unbalanced may seem to have a big, negative impact on the query time. Fortunately, this intuition is incorrect. To analyze the query time, we next bound the number of cells intersected by an axis-parallel plane h.

LEMMA 3. *Let h be a hyperplane orthogonal to the x_i-axis for some i, with $1 \leqslant i \leqslant d$. The number of nodes in $\mathcal{T}(\mathcal{P})$ whose regions intersect h is $O(dn^{1-1/d})$.*

Proof. We call a node whose region intersects h a *visited node*. Let N denote the set of all visited active nodes in $\mathcal{T}(\mathcal{P})$. We will prove that $|N| = O(n^{1-1/d})$, from which it follows that the total number visited nodes is as claimed.

Let i be such that h is orthogonal to the x_i-axis. For the purpose of the analysis, we construct a tree $\mathcal{S}_i$ on the x_i-nodes in $\mathcal{S}(\mathcal{P})$, as follows. We connect every x_i-node to its x_i-parent. This gives us a collection of trees, each rooted at an x_i-node that does not have an x_i-parent. If $i = 1$, there is only one such node, namely, root$(\mathcal{S}(\mathcal{P}))$, and the construction of $\mathcal{S}_i$ is finished. Otherwise, we add a dummy node which we define to be the x_i-parent of the nodes that do not have an x_i-parent yet, and we obtain $\mathcal{S}_i$ by connecting the dummy node to these nodes.

The nodes in $\mathcal{S}_i$ have degree at most 2^d, with the dummy root node having degree at most 2^{d-1}. Define $N_i(\ell)$ to be the collection of visited nodes at level ℓ in $\mathcal{S}_i$, where the level of the root is defined to be zero. Observe that if v is a visited node in $\mathcal{S}_i$, then at most 2^{d-1} of v's children are visited. Hence, $|N_i(\ell)| \leqslant 2^{\ell(d-1)}$.

Now let $\ell^* = \lfloor (1/d) \log n \rfloor$. Then

$$|N_i(\ell^*)| \leqslant 2^{\ell^*(d-1)} \leqslant 2^{((d-1)/d)\log n} = n^{1-1/d}.$$

Now we can start counting the nodes in N. Since the nodes in N are active, their regions in $\mathcal{S}(\mathcal{P})$ and $\mathcal{T}(\mathcal{P})$ are the same by Lemma 2(ii). Hence, they are also visited in $\mathcal{S}(\mathcal{P})$. Thus any node in N is either a proper ancestor of a node in $N_i(\ell^*)$, or it is a node in one of the subtrees rooted at a node in $N_i(\ell^*)$. We consider these two types of nodes in N separately.

Type (i): *Proper ancestors of the nodes in $N_i(\ell^*)$.* First consider the x_i-nodes of this type. The number of these nodes is bounded by

$$\sum_{\ell=1}^{\ell^*-1} |N_i(\ell)| \leqslant \sum_{\ell=1}^{\ell^*-1} 2^{\ell(d-1)} = \frac{\left(2^{d-1}\right)^{\ell^*} - 1}{2^{d-1} - 1}.$$

The other nodes of Type (i), which are x_j-nodes for some $j \neq i$, are charged to their x_i-parent. This way every x_i-node of Type (i) is charged by at most 2^{d-1} other nodes. Hence, the total number of nodes of Type (i) is at most

$$\left(1 + 2^{d-1}\right) \cdot \frac{\left(2^{d-1}\right)^{\ell^*} - 1}{2^{d-1} - 1} = O\left(2^{(d-1)\ell^*}\right).$$

Since $\ell^* = \lfloor (1/d) \log n \rfloor$, this is $O(n^{1-1/d})$.

Type (ii): *Nodes in $N_i(\ell^*)$ and their descendants.* By definition, the level (in the tree $\mathcal{S}_i$) of the nodes in $N_i(\ell^*)$ is ℓ^*. This implies that the ranges associated to these nodes consist of $\lceil n/2^{\ell^*-1} \rceil$ ranks. (The term -1 arises because the nodes at level 1 still have range $[1, n]$, since the dummy node does not have a range associated to it.) The crucial property is that, because of the way $\mathcal{S}(\mathcal{P})$ is constructed, the ranks of the nodes in $N_i(\ell^*)$ are identical; indeed, when recursively subdividing $[1, n]$ into smaller and smaller ranges, there is always one unique range of ranks whose corresponding range of x_i-coordinates contains the x-coordinate of the hyperplane h. Since there is only one point of each rank, this implies that the overall number of points in the subtrees of $\mathcal{T}$ rooted at the nodes in $N_i(\ell^*)$ is at most

$$\left\lceil \frac{n}{2^{\ell^*-1}} \right\rceil = \left\lceil \frac{n}{2^{\lfloor (1/d)\log n \rfloor - 1}} \right\rceil = O\left(n^{1-1/d}\right).$$

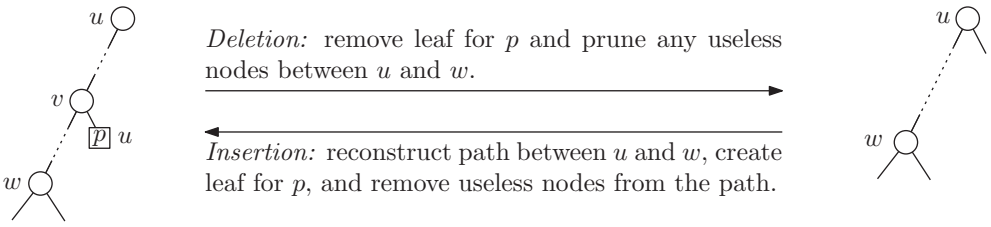

FIG. 2. *Deleting and inserting point p.*

Hence, the total number of active visited nodes of Type (ii) is bounded by $O(n^{1-1/d})$ as well. □

The algorithm for answering an orthogonal range query with a range R in a rank-based kd-tree is the same as for a standard kd-tree. Thus we proceed as follows when we visit a node v (which is initially the root): If region$(v) \subset R$, we report every point lying in the subtree rooted at v. Otherwise, we recurse on the children of v whose regions intersect R. This can be either one child or both children—which children to recurse on can be decided by comparing R to the splitting plane $h(v)$. When we reach a leaf, we check if the point lies inside R and if so, report it.

Reporting all points in a subtree rooted at a node v takes $O(d|P(v)|)$ time. It remains to bound the number of nodes whose regions intersect, but are not contained in, R. Any such region is intersected by a facet of R. Since R has $2d$ facets and each side intersects at most $O(dn^{1-1/d})$ nodes by Lemma 3, the query time is $O(d^2n^{1-1/d} + dk)$. 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 $O(dn)$ storage and can be built in $O(dn \log n)$ time. An orthogonal–range-search query on a rank-based kd-tree takes $O(d^2n^{1-1/d} + dk)$ time, where k is the number of reported points.*

The KDS. We now describe how to kinetize a rank-based kd-tree $\mathcal{T}(\mathcal{P})$ for a set of continuously moving points $\mathcal{P}$. The combinatorial structure of $\mathcal{T}(\mathcal{P})$ 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 reinsert them with their new ranks. (During the deletion and reinsertion, 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 reinsertion of p and q. Hence, $\mathcal{T}(\mathcal{P})$ remains unchanged except for a small part that involves p and q, as explained next.

Deletion. Let v be the parent of the leaf u containing p; see Figure 2. Note that v is active. The leaf u must be deleted. Furthermore, v stops being active. Let w be the first active descendent of v, if it exists, and otherwise let w be the leaf whose ancestor is v. Let u be the first active ancestor of v. Because v is no longer active, it and/or the at-most $2d$ nodes between v and u may have to be pruned. Thus we need to check if their splitting hyperplanes define a facet of region(w); if not, they have become useless and need to be pruned.

Insertion. Search with p in $\mathcal{T}(\mathcal{P})$ for the lowest active node u whose region contains p. Assume without loss of generality that p should go to the left subtree of u.

If the left child of u is a leaf, storing some point q, then we proceed as follows. We construct the subtree of $\mathcal{S}(\mathcal{P})$ containing the points p and q—this is easily done in $O(d \log n)$ time—and then prune the useless nodes.

Otherwise, let w be the highest active node in the u's left subtree. Note that all nodes in $\mathcal{T}(\mathcal{P})$ between u and w are degree-one nodes. We replace the path in $\mathcal{T}(\mathcal{P})$ between u and w by the path in $\mathcal{S}(\mathcal{P})$ between the nodes corresponding to u and w. (We do not maintain $\mathcal{S}(\mathcal{P})$ explicitly, but we can easily construct the path between u and w in $O(d \log n)$ time from the information available in $\mathcal{T}(\mathcal{P})$.) We now find the highest node v on this path whose region contains p and create a leaf child of v storing p. Finally, we prune all useless nodes between u and v and between v and w.

THEOREM 5. *A kinetic rank-based kd-tree for a set $\mathcal{P}$ of n moving points in d dimensions uses $O(dn)$ 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 $O(d \log n)$ time, and each point is involved in $O(1)$ certificates.*

Remark. For the bound on the number of events in our rank-based kd-tree, it is sufficient that any pair of points swaps their ordering along the x_i-axis $O(1)$ times for any $1 \leqslant i \leqslant d$.

3. Rank-based longest-side kd-trees. Longest-side kd-trees are a variant of kd-trees, where the orientation of the splitting hyperplane for a node v is chosen according to the shape of the region associated with v: one always chooses the splitting hyperplane orthogonal to the longest side of region(v). Dickerson, Duncan, and Goodrich [7] showed that a longest-side kd-tree storing a set $\mathcal{P}$ of points in $\mathbb{R}^d$ can be used to answer the following queries quickly:

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

$(1-\varepsilon)$-*farthest neighbor query.* Given a query point $q \in \mathbb{R}^d$ and a parameter $\varepsilon > 0$, this query returns a point $p \in \mathcal{P}$ such that $d(p, q) \geqslant (1 - \varepsilon) \cdot d(p^*, q)$, where $p^* \in \mathcal{P}$ is the true farthest neighbor of q.

ε-*approximate range query.* Given a query region Q with diameter $\mathrm{diam}(Q)$ and a parameter $\varepsilon > 0$, this query returns (or counts) a set $\mathcal{P}'$ such that $\mathcal{P} \cap Q \subset \mathcal{P}' \subset \mathcal{P}$ and $d(p, Q) \leqslant \varepsilon \cdot \mathrm{diam}(Q)$ for every point $p \in \mathcal{P}'$.

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 two-dimensional longest-side kd-trees, so called *rank-based longest-side kd-trees.* A rank-based longest-side kd-tree not only preserves all main properties of a longest-side kd-tree, but it can be kinetized easily and efficiently as well. As in the previous section, we first describe another tree, namely, the skeleton $\mathcal{S}(\mathcal{P})$ on which the rank-based longest-side kd-tree is based. We then show how to extract a rank-based longest-side 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 v be a node whose subtree must be constructed; initially $v = \mathrm{root}(\mathcal{S}(\mathcal{P}))$. If $\mathcal{P}(v)$ contains only one point, then v is a leaf of $\mathcal{S}(\mathcal{P})$. If

$\mathcal{P}(v)$ contains more than one point, then we have to determine the proper splitting line. Suppose the longest side of region(v) is parallel to the x_i-axis. We set axis(v) to be x_i. If x_i-parent(v) does not exist, then we set range(v) $= [1, n]$. Otherwise, if v is contained in the left subtree of x_i-parent(v), then range(v) is equal to the first half of range(x_i-parent(v)), and if v is contained in the right subtree of x_i-parent(v), then range(v) is equal to the second half of range(x_i-parent(v)). The splitting line of v, denoted by $l(v)$, is orthogonal to axis(v) and specified by the point whose rank in $\mathcal{A}_i$ is the median of range(v). If there is a point of $\mathcal{P}(v)$ on the left side of $l(v)$ (resp., on the right side of $l(v)$ or on $l(v)$), a node is created as the left child (resp., the right child) of v. The points of $\mathcal{P}(v)$ which are on the left side of $l(v)$ are associated with the left child of v, the remainder is associated with the right child of v.

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. Let $v_1, \ldots, v_k$ be the nodes on the path from the root to a leaf using the same axis. Note that $|\text{range}(v_{j+1})| \leqslant \lceil |\text{range}(v_j)|/2 \rceil$ for any $j = 1, \ldots, k-1$, and $|\text{range}(v_k)| \geqslant 1$. Hence, $k \leqslant \lceil \log n \rceil$. Since there are two axes, this implies that the length of any path from the root to a leaf is at most $2\lceil \log n \rceil$. 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 $O(|\mathcal{P}(v)|)$ time to split the points at a node v. 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 rank-based longest-side 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, Duncan, and Goodrich [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 a rank-based longest-side kd-tree has these two properties, the proof applies. ∎

As in the previous section, we obtain our structure by pruning useless nodes from $\mathcal{S}(\mathcal{P})$. As before, useful nodes are defined as follows. A node v is useful if v is a leaf, or an active node, or $l(v)$ defines one of the sides of the boundary of region(w) where w is the highest active descendant of v; otherwise, v is useless. A rank-based longest-side kd-tree is obtained from $\mathcal{S}(\mathcal{P})$ by pruning useless nodes. Thus the parent of a node v in the rank-based longest-side kd-tree is the first unpruned ancestor of v in $\mathcal{S}(\mathcal{P})$. In other words, the rank-based longest-side kd-tree is obtained by removing for every path of degree-one nodes all of its nodes, except the at-most four nodes defining a side of the region of the node immediately below this path.

The following lemma shows that a rank-based longest-side kd-tree has linear size and that it preserves the main property of a longest-side kd-tree.

THEOREM 8.

(i) *A rank-based longest-side kd-tree on a set of n points in $\mathbb{R}^2$ has depth $O(\log n)$, size $O(n)$, and it can be constructed in $O(n \log n)$ time.*

(ii) *The number of nodes in a rank-based 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) The proof is the same as the proof of Lemma 2(i).

(ii) Let L be a set of nodes in a rank-based longest-side kd-tree $\mathcal{T}(\mathcal{P})$ whose regions are disjoint and that intersect at least two opposite sides of a square $\mathcal{C}$. The idea of the proof is to find a suitable set L' of nodes in $\mathcal{S}(\mathcal{P})$ and then apply Lemma 7. In the rest of the proof, we use v' to denote the node in $\mathcal{S}(\mathcal{P})$ corresponding to any node v in $\mathcal{T}(\mathcal{P})$.

The set L' is defined as follows. Consider a node $v \in L$. If v is active, then we add v to L'. Otherwise, let w be the lowest active ancestor of v, and let u' be node in $\mathcal{S}(\mathcal{P})$ that is the child of w' on the path to v'; note that u' could be v'. We add u' to L'.

We have added a node to L' for each node in L. We claim that the added nodes are all distinct, so that $|L'| = |L|$. Indeed, an active node in $\mathcal{S}(\mathcal{P})$ is only added once (namely, if its corresponding node in $\mathcal{T}(\mathcal{P})$ is in L), and a child of an active node is only added once because L can contain only one node from any path of degree-one nodes.

We also claim that the regions corresponding to the nodes in L' are disjoint. To see this, we observe that the region corresponding to an active node v in $\mathcal{T}(\mathcal{P})$ is the same as the region corresponding to the node v' in $\mathcal{S}(\mathcal{P})$; (see Lemma 2(ii)), which also applies here since the pruning strategy is the same. The node u' we add to L' for a nonactive node $v \in L$ may have a larger region than v. However, L' does not contain any node from the subtree rooted at v, so this enlargement of the region does not cause it to start intersecting any of the other regions.

It follows that we can apply Lemma 7 to conclude that $|L| = O(\log n)$, which proves part (ii) of the lemma, since $|L| = |L'|$. □

Using a rank-based longest-side kd-tree, similar algorithms to the algorithms of Dickerson, Duncan, and Goodrich [7] can be used to answer $(1+\varepsilon)$-nearest neighbor, $(1-\varepsilon)$-farthest neighbor, and ε-approximate range queries.

THEOREM 9. *A rank-based longest-side 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 nonconvex region, a counting (or reporting) ε-approximate range 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).*

The KDS. We now describe how to kinetize a rank-based longest-side kd-tree $\mathcal{T}(\mathcal{P})$ for a set of continuously moving points $\mathcal{P}$. Clearly, the combinatorial structure of $\mathcal{T}(\mathcal{P})$ 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 $s_1(v) < s_2(v)$ (or $s_2(v) < s_1(v)$) for each node v in $\mathcal{S}(\mathcal{P})$, where $s_i(v)$ denotes the length of the x_i-side of region(v). Let $x_i(p)$ denote the x_i-coordinate of p. We have $s_i(v) = x_i(p) - x_i(q)$, where p and q are two points specifying two splitting lines in the x_i-ancestors of v in $\mathcal{S}(\mathcal{P})$. More precisely, the splitting lines defined by p and q are associated with the *left ancestor* and the *right ancestor* of v in $\mathcal{S}(\mathcal{P})$. Here the left ancestor is defined as the lowest ancestor u of v such that v is in u's left subtree; similarly, the right ancestor is defined as the lowest ancestor w of v

such that v is in w's right subtree. 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 $\mathcal{T}(\mathcal{P})$. 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, one of them should define a vertical side and the other should define a horizontal side—nothing happens if two vertical sides, or two horizontal 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. (Note, however, that the way in which a subset of points is split at a node should be the same as in our rank-based longest-side kd-tree, so one cannot just take a standard balanced search tree such as a red-black tree.)

Let v be a node in $\mathcal{T}_i$, and let v_r and v_ℓ be the first right and the first left ancestors of v in $\mathcal{T}_i$. If p and q are the two points used in v_r and v_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 $\mathcal{T}(\mathcal{P})$. 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 $x_i(p)$ by $x_i(q)$ and vice versa and by 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 $O(\log^2 n)$ time. To update $\mathcal{T}(\mathcal{P})$, we first delete p and q from $\mathcal{T}(\mathcal{P})$ and then we reinsert them in their new order. These deletions and insertions are performed similarly to the way they are performed in the rank-based kd-tree of the previous section; the only difference is how a path in the skeleton is (temporarily) constructed: this reconstruction should be done using the longest-side splitting rule.

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 $\mathcal{T}(\mathcal{P})$ 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

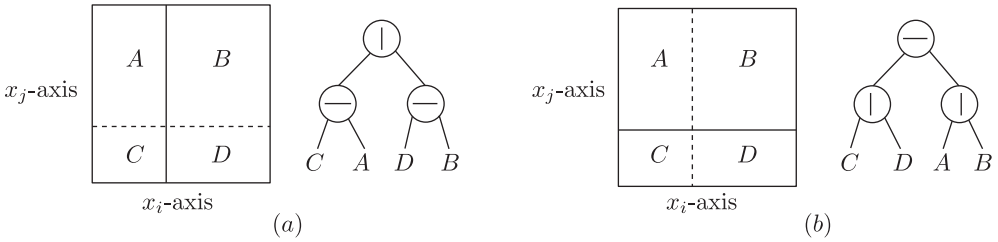

FIG. 3. *The status of the rank-based longest-side kd-tree $\mathcal{T}(\mathcal{P})$ before handling a longest-side event and after handling the event.*

certificate failure can not correspond to a real longest-side event. Otherwise, we need to determine which, if any, of the regions of $\mathcal{S}(\mathcal{P})$ correspond to the event. The two lines passing through p and q and orthogonal to the x_i-axis, together with the two lines passing through p' and q' and orthogonal to the x_j-axis specify a unique rectangular region R. We thus have to search for a node v in the tree with region$(v) = R$. It is clear there is at most one such a node in two dimensions.[2] While we search in $\mathcal{T}(\mathcal{P})$ with R, we temporarily insert all the skeleton nodes into the path that have been pruned. Note that it is easy to compute these nodes as we descend.

If there is no node whose region matches the region R, then we delete the temporary nodes and stop handling the event.

Otherwise, there is exactly one node v in $\mathcal{S}(\mathcal{P})$ with region$(v) = R$. We add the two children v_r and v_ℓ of v in $\mathcal{S}(\mathcal{P})$ to $\mathcal{T}(\mathcal{P})$, provided that they do not already exist in $\mathcal{T}(\mathcal{P})$. Let the x_i-side of region(v) be bigger than the x_j-side of region(v) at the point in time just before t, denoted by t^-. (Note that region(v) is a square at time t.) At time t^-, the line $l(v)$ must be orthogonal to the x_i-axis, and $l(v_\ell)$ and $l(v_r)$ must be orthogonal to the x_j-axis, as illustrated in Figure 3(a). Moreover, $l(v_\ell) = l(v_r)$, because the median of all points between the two x_i-sides of region(v) defines $l(v_\ell)$ and $l(v_r)$. Let A, B, C, and D be the four regions defined by $l(v)$, $l(v_\ell)$, and $l(v_r)$, as illustrated in Figure 3(a). We now split region(v) with a line that is orthogonal to the x_j-axis and region(v_r) and region(v_ℓ) with a line that is orthogonal to the x_i-axis. Clearly $l(v)$ at time t is equal to $l(v_\ell)$ and $l(v_r)$ at time t^-, and $l(v_\ell)$ and $l(v_r)$ at time t are equal to $l(v)$ 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 $\mathcal{T}(\mathcal{P})$, as illustrated in Figure 3(b). Finally every node on the path from the root to v as well as v_r and v_ℓ must be checked whether they are useless. If so, they must be removed from $\mathcal{T}(\mathcal{P})$.

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,

[2] In higher dimensions, instead of the above four lines, we have two hyperplanes orthogonal to the x_i-axis and two hyperplanes orthogonal to the x_j-axis. Because a box in higher dimensions has more than four sides, these four hyperplanes do not uniquely specify a region in $\mathcal{T}(\mathcal{P})$. This is the problem when attempting to extend these results to higher dimensions.

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 rank-based longest-side 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.*

Remark. For the bound on the number of events in our rank-based longest-side kd-tree, we do not need constant-degree algebraic trajectories: it is sufficient that any pair of points swaps x- or y-order $O(1)$ times, and every two pairs of points define the same x- or y-distance $O(1)$ times, that is, every two elements in the side-length list swap at most $O(1)$ times.

4. Conclusions. We presented a variant of kd-trees, 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. The main advantage of our rank-based kd-tree is that it can be kinetized easily and efficiently. Unfortunately, our rank-based kd-tree does not allow efficient insertions and deletions, since these may cause a dramatic change in the rank-based kd-tree. A challenging problem is how to adapt the rank-based kd-tree so that it can handle updates while 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 $\mathbb{R}^2$, and we showed that rank-based longest-side kd-trees can be kinetized efficiently. Like longest-side kd-trees, rank-based longest-side kd-trees support ε-approximate nearest-neighbor, ε-approximate farthest-neighbor, and ε-approximate range 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.

Acknowledgments. We would like to thank two anonymous referees for all their comments, which improved the presentation of our paper considerably.

REFERENCES

[1] P. AGARWAL, L. ARGE, AND J. ERICKSON, *Indexing moving points*, J. Comput. System Sci., 66 (2003), pp. 207–243.

[2] P. AGARWAL, J. GAO, AND L. GUIBAS, *Kinetic medians and kd-trees*, in Proceedings of the 10th European Symposium on Algorithms, pp. 5–16, Lecture Notes in Comput. Sci. 2461, Springer-Verlag, Berlin, 2002.

[3] J. BASCH, L. GUIBAS, AND J. HERSHBERGER, *Data structures for mobile data*, J. Algorithms, 31 (1999), pp. 1–28.

[4] J. BASCH, L. GUIBAS, AND L. ZHANG, *Proximity problems on moving points*, in Proceedings of the 13th ACM Symposium on Computational Geometry, 1997, pp. 344–351.

[5] J. L. BENTLEY, *Multidimensional binary search trees used for associative searching*, Commun. ACM, 18 (1975), pp. 509–517.

[6] M. DE BERG, J. COMBA, AND L. J. GUIBAS, *A segment-tree based kinetic BSP*, in Proceedings of the 17th ACM Symposium on Computational Geometry, 2001, pp. 134–140.

[7] M. DICKERSON, C. A. DUNCAN, AND M. T. GOODRICH, *K-d trees are better when cut on the longest side*, in Proceedings of the 8th European Symposium on Algorithms, pp. 179–190, Lecture Notes in Comput. Sci. 1879, Springer-Verlag, Berlin, 2000.

[8] L. GUIBAS, *Kinetic data structures: A state of the art report*, in Proceedings of the 3rd Workshop on Algorithmic Foundations of Robotics, 1998, pp. 191–209.

[9] L. GUIBAS, *Motion*, in Handbook of Discrete and Computational Geometry, 2nd ed., J. Goodman and J. O'Rourke, ed., CRC Press, Boca Raton, FL, 2004, pp. 1117–1134.

[10] M. J. VAN KREVELD AND M. H. OVERMARS, *Divided k-d Trees*, Algorithmica, 6 (1991), pp. 840–858.