NEW TECHNIQUES FOR EXACT AND APPROXIMATE DYNAMIC CLOSEST-POINT PROBLEMS

SANJIV KAPOOR* AND MICHIEL SMID†

Abstract. Let S be a set of n points in $\mathbb{R}^D$. It is shown that a range tree can be used to find an L_∞-nearest neighbor in S of any query point, in $O((\log n)^{D-1} \log\log n)$ time. This data structure has size $O(n(\log n)^{D-1})$ and an amortized update time of $O((\log n)^{D-1} \log\log n)$. This result is used to solve the $(1+\epsilon)$-approximate L_2-nearest neighbor problem within the same bounds (up to a constant factor that depends on ϵ and D). In this o problem, for any query point p, a point $q \in S$ is computed such that the euclidean distance between p and q is at most $(1+\epsilon)$ times the euclidean distance between p and its true nearest neighbor. This is the first dynamic data structure for this problem having close to linear size and polylogarithmic query and update times.

New dynamic data structures are given that maintain a closest pair of S. For $D \geq 3$, a structure of size $O(n)$ is presented with amortized update time $O((\log n)^{D-1} \log\log n)$. The constant factor in this space (resp. time bound) is of the form $O(D)^D$ (resp. $2^{O(D^2)}$). For $D = 2$ and any non-negative integer constant k, structures of size $O(n \log n/(\log\log n)^k)$ (resp. $O(n)$) are presented having an amortized update time of $O(\log n \log\log n)$ (resp. $O((\log n)^2/(\log\log n)^k)$). Previously, no deterministic linear size data structure having polylogarithmic update time was known for this problem.

Key words. proximity, dynamic data structures, point location, approximation

AMS subject classification. 68U05

1. Introduction.

Closest-point problems are among the basic problems in computational geometry. In such problems, a set S of n points in D-dimensional space is given and we have to store it in a data structure such that a point in S nearest to a query point can be computed efficiently, or we have to compute a closest pair in S, or for each point in S another point in S that is closest to it. These problems are known as the *nearest-neighbor problem*, the *closest pair problem*, and the *all-nearest-neighbors-problem*, respectively. In the dynamic version of these problems, the set S is changed by insertions and deletions of points.

It is assumed that the dimension $D \geq 2$ is a constant independent of n. Moreover, distances are measured in the L_t-metric, where $1 \leq t \leq \infty$ is a fixed real number. In this metric, the distance $d_t(p,q)$ between the points p and q is defined as $d_t(p,q) = \left(\sum_{i=1}^D |p_i - q_i|^t\right)^{1/t}$ if $1 \leq t < \infty$, and for $t = \infty$ it is defined as $d_\infty(p,q) = \max_{1 \leq i \leq D} |p_i - q_i|$.

The planar version of the nearest-neighbor problem can be solved optimally, i.e., with $O(\log n)$ query time using $O(n)$ space, by means of Voronoi diagrams. (See [15].) In higher dimensions, however, the situation is much worse. The best results known are due to Clarkson [7] and Arya et al.[1]. In [7], a randomized data structure is given that finds a nearest-neighbor of a query point in $O(\log n)$ expected time. This structure has size $O(n^{\lceil D/2 \rceil + \delta})$, where δ is an arbitrarily small positive constant. In [1], the problem is solved with an expected query time of $O(n^{1-1/\lceil (D+1)/2 \rceil} (\log n)^{O(1)})$ using $O(n \log\log n)$ space.

* Department of Computer Science, Indian Institute of Technology, Hauz Khas, New Delhi 110016, India. E-mail: `skapoor@cse.iitd.ernet.in`. The research on this work started when this author visited the Max-Planck-Institut für Informatik.

† Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany. E-mail: `michiel@mpi-sb.mpg.de`. This author was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

TABLE 1

Deterministic data structures for the dynamic closest pair problem. In the last two lines, k is an arbitrary non-negative integer constant. All bounds are "big-oh" with constant factors that depend on the dimension D and, in the last two lines, on D and k. The update times are either worst-case (w) or amortized (a).

mode	dimension	update time	space	reference
insertions	$D \geq 2$	$\log n \ \ (w)$	n	[17, 18]
deletions	$D \geq 2$	$(\log n)^D \ \ (a)$	$n(\log n)^{D-1}$	[23]
dynamic	$D \geq 2$	$\sqrt{n} \log n \ \ (w)$	n	[16, 20]
dynamic	$D \geq 2$	$(\log n)^D \log\log n \ \ (a)$	$n(\log n)^D$	[22]
dynamic	$D \geq 3$	$(\log n)^{D-1} \log\log n \ \ (a)$	n	this paper
dynamic	2	$\log n \log\log n \ \ (a)$	$n \log n/(\log\log n)^k$	this paper
dynamic	2	$(\log n)^2/(\log\log n)^k \ \ (a)$	n	this paper

It seems that in higher dimensions it is impossible to obtain polylogarithmic query time using $O(n(\log n)^{O(1)})$ space. Moreover, even in the planar case there is no dynamic data structure known that has polylogarithmic query and update times and that uses $O(n(\log n)^{O(1)})$ space.

Therefore, it is natural to ask whether the *approximate nearest-neighbor problem* allows more efficient solutions. Let $\epsilon > 0$. A point $q \in S$ is called a $(1+\epsilon)$-approximate neighbor of a point $p \in \mathbb{R}^D$, if $d_t(p,q) \leq (1+\epsilon) d_t(p,p^*)$, where $p^* \in S$ is the true nearest-neighbor of p.

This approximate neighbor problem was considered in Bern [5] and Arya et al.[1, 2]. In the latter paper, the problem is solved optimally: They give a deterministic data structure of size $O(n)$ that can find a $(1+\epsilon)$-approximate neighbor in $O(\log n)$ time, for any positive constant ϵ. At this moment, however, no dynamic data structures are known for this problem.

In this paper, we first show that for the L_∞-metric, the nearest-neighbor problem can be solved efficiently. To be more precise, we show that the range tree (see [12, 14, 15, 25]) can be used to solve this problem. As a result, we solve the L_∞-neighbor problem with a query time of $O((\log n)^{D-1} \log\log n)$ and an amortized update time of $O((\log n)^{D-1} \log\log n)$, using $O(n(\log n)^{D-1})$ space. For the static version of this problem, the query time is $O((\log n)^{D-1})$ and the space bound is $O(n(\log n)^{D-2})$.

Using this result, we give a data structure that solves the approximate L_2-neighbor problem, for any positive constant ϵ, within the same complexity bounds. (The constant factors depend on D and ϵ.)

We also consider the dynamic closest pair problem. Note that the static version has been solved already for a long time. Several algorithms are known that compute a closest pair in $O(n \log n)$ time, which is optimal. (See [4, 10, 15, 19].) The dynamic version, however, has been investigated only recently. In Table 1, we give an overview of the currently best known data structures for maintaining a closest pair under insertions and/or deletions of points. For an up-to-date survey, we refer the reader to Schwarz's Ph.D. Thesis [17].

Note that all data structures of Table 1 are deterministic and can be implemented in the algebraic computation tree model. If we add randomization to this model, then there is a data structure of size $O(n)$ that maintains a closest pair in $O((\log n)^2)$ expected time per update. (See Golin et al.[9].)

In this paper, we give new deterministic data structures for the dynamic closest pair problem. These structures are based on our solution to the L_∞-neighbor problem, data structures for maintaining boxes of constant overlap, and a new transformation.

Given *any* dynamic closest pair data structure having more than linear size, this transformation produces another dynamic closest pair structure that uses less space. The complexity bounds of the new structures are shown in the last three lines of Table 1. The results of the last two lines are obtained by applying the transformation repeatedly. Note that we obtain the first linear size deterministic data structure that maintains a closest pair in polylogarithmic time.

Finally, we consider the all-nearest-neighbors problem. Again, in the planar case, the problem can be solved using Voronoi diagrams. For the D-dimensional case, Vaidya [24] has given an optimal $O(n \log n)$ time algorithm that computes the L_t-neighbor for each point of S. No non-trivial solutions seem to be known for maintaining for each point its nearest-neighbor. One of our data structures (not the ones mentioned in Table 1) for maintaining the closest pair basically maintains the L_∞-neighbor of each point in S. As a result, we get a data structure of size $O(n(\log n)^{D-1})$ that maintains for each point in S its L_∞-neighbor in $O((\log n)^{D-1} \log\log n)$ amortized time per update.

We want to remark here that all our algorithms use classical and well-understood data structures, such as range trees, segment trees, and skewer trees. Moreover, we apply the well known technique of fractional cascading ([6, 14]) several times.

The rest of this paper is organized as follows. In Section 2, we recall the definition of a range tree. This data structure is used in Section 3 to solve the L_∞-neighbor problem. In Section 4, we give the data structure for solving the approximate L_2-nearest neighbor problem.

Our first data structure for maintaining the closest pair stores a dynamically changing set of boxes that are of constant overlap, i.e., there is a constant c such that each box contains the centers of at most c boxes in its interior. We have to maintain such boxes under insertions and deletions such that for any query point $p \in \mathbb{R}^D$, we can find all boxes that contain p. In Section 5, we give two data structures for this problem. The first one is based on segment trees. (See [13, 15].) Therefore, its size is superlinear. We also give a linear size solution having a slightly worse update time in the planar case. This solution is based on skewer trees. (See [8, 21].)

In Section 6, we use the results obtained to give a new data structure that maintains the closest pair in a point set. This structure has size $O(n(\log n)^{D-1})$ and an amortized update time of $O((\log n)^{D-1} \log\log n)$. It immediately gives the dynamic data structure for the all-L_∞-nearest-neighbors problem. In Section 7, we give the transformation that reduces the size of a dynamic closest pair structure. Applying this transformation repeatedly to the structure of Section 6 gives the new results mentioned in Table 1. Some concluding remarks are given in Section 8.

2. Range trees. In this section, we recall the definition of a range tree. See [12, 13, 15, 25]. The coordinates of a point p in $\mathbb{R}^D$ are denoted by p_i, $1 \leq i \leq D$. Moreover, we denote by p' the point $(p_2, \ldots, p_D)$ in $\mathbb{R}^{D-1}$. If S is a set of points in $\mathbb{R}^D$, then we define $S' := \{p' : p \in S\}$. We note that S' is to be considered as a multiset, i.e., elements may occur more than once.

The range tree: Let S be a set of n points in $\mathbb{R}^D$. A *D-dimensional range tree* for the set S is defined as follows. For $D = 1$, it is a balanced binary search tree, storing the elements of S in sorted order in its leaves.

For $D > 1$, it consists of a balanced binary search tree, called the *main tree*, storing the points of S in its leaves, sorted by their first coordinates. For each internal node v of this main tree, let S_v be the set of points that are stored in its subtree. Node v contains a pointer to the rightmost leaf in its subtree and (a pointer to) an *associated* *structure*, which is a $(D-1)$-dimensional range tree for the set S'_v.

Hence, a 2-dimensional range tree for a set S consists of a binary tree, storing the points of S in its leaves sorted by their x-coordinates. Each internal node v of this tree contains a pointer to the rightmost leaf in its subtree and (a pointer to) a binary tree that stores the points of S_v in its leaves sorted by their y-coordinates.

Let p be any point in $\mathbb{R}^D$. Consider the set $\{r \in S : r_1 \geq p_1\}$, i.e., the set of all points in S having a first coordinate that is at least equal to p's first coordinate. Using the range tree, we can decompose this set into $O(\log n)$ canonical subsets:

Initialize $M := \emptyset$. Starting in the root of the main tree, search for the leftmost leaf storing a point whose first coordinate is at least equal to p_1. During this search, each time we move from a node v to its left son, add the right son of v to the set M. Let v be the leaf in which this search ends. If the point stored in this leaf has a first coordinate that is at least equal to p_1, then add v to the set M.

LEMMA 2.1. *The set M of nodes of the main tree that is computed by the given algorithm satisfies*

$$\{r \in S : r_1 \geq p_1\} = \bigcup_{v \in M} S_v.$$

Moreover, M is computed in $O(\log n)$ time.

Range trees can be maintained under insertions and deletions of points such that each binary tree that is part of the structure has a height logarithmic in the number of its leaves. The update algorithms use dynamic fractional cascading. For details, we refer the reader to Mehlhorn and Näher [14]. In the next theorem, we state the complexity of the range tree.

THEOREM 2.2. *A D-dimensional range tree, where $D \geq 2$, for a set of n points has $O(n(\log n)^{D-1})$ size and can be maintained in $O((\log n)^{D-1} \log\log n)$ amortized time per insertion and deletion.*

3. The L_∞-neighbor problem. Recall the following notations. For any point $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$, we denote by p' the point $(p_2, \ldots, p_D)$ in $\mathbb{R}^{D-1}$. If S is a set of points in $\mathbb{R}^D$, then $S' = \{p' : p \in S\}$. Finally, if v is a node of the main tree of a range tree storing a set S, then S_v denotes the set of points that are stored in the subtree of v.

Let S be a set of n points in $\mathbb{R}^D$. We want to store this set into a data structure such that for any query point $p \in \mathbb{R}^D$, we can find a point in S having minimal L_∞-distance to p. Such a point is called an *L_∞-neighbor* of p in S.

Let q be an L_∞-neighbor of p in the set $\{s \in S : s_1 \geq p_1\}$. We call q a *right-L_∞-neighbor* of p in S. Similarly, a point r is called a *left-L_∞-neighbor* of p in S, if it is an L_∞-neighbor of p in the set $\{s \in S : s_1 \leq p_1\}$. In order to guarantee that both neighbors always exist, we add the $2D$ artificial points $(a_1, \ldots, a_D)$, where all a_i are zero, except for one which is either ∞ or $-\infty$, to the set S. Note that none of these points can be L_∞-neighbor of any query point.

The data structure that solves the L_∞-neighbor problem is just a D-dimensional range tree storing the points of S and the artificial points. In Figure 1, our recursive algorithm is given that finds an L_∞-neighbor of any query point in $\mathbb{R}^D$.

We prove the correctness of this query algorithm. It is clear that the algorithm is correct in the 1-dimensional case. So let $D \geq 2$ and assume that the algorithm is correct for smaller values of D. The following lemma turns out to be useful.

LEMMA 3.1. *Let $p \in \mathbb{R}^D$ and let q be its right-L_∞-neighbor in S. Let w be any node in the main tree of the range tree such that $q \in S_w$ and p has a first coordinate*

Algorithm Neighbor(p, S, D) (* returns an L_∞-neighbor of p in S *)
begin
1. Assume $D = 1$.
$q^L :=$ maximal element in the 1-dimensional range tree that is less than p;
$q^R :=$ minimal element in the 1-dimensional range tree that is at least equal to p;
if $|p - q^R| \leq |p - q^L|$ **then** return q^R **else** return q^L **fi**;
2. Assume $D \geq 2$.
2a. Compute a right-L_∞-neighbor:
Stage 1. Compute the set M of nodes of the main tree, see Lemma 2.1. Number these nodes $v_1, v_2, \ldots, v_m$, $m = |M|$, where v_i is closer to the root than v_{i-1}, $2 \leq i \leq m$.
Stage 2. (* one of the sets S_{v_i} contains a right-L_∞-neighbor of p *)
$C := \emptyset$; $i := 1$; $stop := false$;
while $i \leq m$ **and** $stop = false$
do $x' :=$ Neighbor$(p', S'_{v_i}, D - 1)$;
$r :=$ the point stored in the rightmost leaf of the subtree of v_i;
if $d_\infty(p', x') > |p_1 - r_1|$
then $C := C \cup \{x\}$; $i := i + 1$
else $v := v_i$; $stop := true$
fi
od;
if $stop = false$
then $q^R :=$ a point of C having minimal L_∞-distance to p;
goto **2b**
fi;
Stage 3. (* the set $C \cup S_v$ contains a right-L_∞-neighbor of p *)
while v is not a leaf
do $w :=$ left son of v;
$x' :=$ Neighbor$(p', S'_w, D - 1)$;
$r :=$ the point stored in the rightmost leaf of the subtree of w;
if $d_\infty(p', x') > |p_1 - r_1|$
then $C := C \cup \{x\}$; $v :=$ right son of v
else $v := w$
fi
od;
$q^R :=$ a point of $C \cup S_v$ having minimal L_∞-distance to p;
2b. Compute a left-L_∞-neighbor:
In a completely symmetric way, compute a left-L_∞-neighbor q^L of p;
2c. if $d_\infty(p, q^R) \leq d_\infty(p, q^L)$ **then** return q^R **else** return q^L **fi**
end

FIG. 1. *Finding an L_∞-neighbor.*

that is at most equal to the first coordinate of any point in S_w. Let r be the point of S that is stored in the rightmost leaf of w's subtree. Finally, let

$$N := |\{x \in S_w : |p_j - x_j| \leq |p_1 - r_1|, 2 \leq j \leq D\}|.$$

If $N = 0$, then q' is an L_∞-neighbor of p' in the $(D-1)$-dimensional set S'_w.

Proof. The proof is by contradiction. So let s be a point in S_w such that s' is an L_∞-neighbor of p' in the set S'_w and assume that $d_\infty(p', s') < d_\infty(p', q')$. Let $\delta := |p_1 - r_1|$.

Since q and s are elements of S_w, we have $|p_1 - q_1| \leq \delta$ and $|p_1 - s_1| \leq \delta$. Since $N = 0$, there is a j, $2 \leq j \leq D$, such that $|p_j - q_j| > \delta$. Therefore, $d_\infty(p, q) = d_\infty(p', q')$. Similarly, there is a k, $2 \leq k \leq D$, such that $|p_k - s_k| > \delta$ and, hence, $d_\infty(p, s) = d_\infty(p', s')$. This implies that $d_\infty(p, s) < d_\infty(p, q)$, i.e., q is not a right-L_∞-neighbor of p in S. This is a contradiction. □

We analyze part **2a** of the algorithm. As we will see, this part computes a right-L_∞-neighbor of p. Consider the nodes $v_1, \ldots, v_m$ that are computed in Stage 1. We know that

$$\{r \in S : r_1 \geq p_1\} = \bigcup_{i=1}^{m} S_{v_i}.$$

Hence, one of the sets S_{v_i} contains a right-L_∞-neighbor of p.

Note that each point in $S_{v_{i-1}}$ has a first coordinate that is at most equal to the first coordinate of any point in S_{v_i}, $2 \leq i \leq m$.

LEMMA 3.2. *If the variable stop has value false after the while-loop of Stage 2 has been completed, then the set C contains a right-L_∞-neighbor of p in S.*

Proof. Let i, $1 \leq i \leq m$, be an index such that the set S_{v_i} contains a right-L_∞-neighbor q of p. We consider what happens during the i-th iteration of the while-loop. Let x and r be the points that are selected in this iteration. Since $d_\infty(p', x') > |p_1 - r_1|$, it is clear that the value of the integer

$$N := |\{y \in S_{v_i} : |p_j - y_j| \leq |p_1 - r_1|, 2 \leq j \leq D\}|$$

is zero. Then, Lemma 3.1 implies that q' is an L_∞-neighbor of p' in S'_{v_i}. Hence, $d_\infty(p', x') = d_\infty(p', q')$. Since $d_\infty(p', x') = d_\infty(p, x)$ and $d_\infty(p', q') = d_\infty(p, q)$, it follows that $d_\infty(p, x) = d_\infty(p, q)$. This proves that x is a right-L_∞-neighbor of p. Since x is added to C during this iteration, the proof is completed. □

LEMMA 3.3. *If the variable stop has value true after the while-loop of Stage 2 has been completed, then the set $C \cup S_v$ contains a right-L_∞-neighbor of p in S.*

Proof. Consider the integer i such that during the i-th iteration of the while-loop the variable *stop* is set to the value *true*. Then $v = v_i$.

We distinguish two cases. First let $j \geq i+1$ and assume that the set S_{v_j} contains a right-L_∞-neighbor q of p in S. Consider what happens during the i-th iteration of the while-loop. Let x and r be the points that are selected during this iteration. Moreover, let $\delta := |p_1 - r_1|$. Since $|p_1 - x_1| \leq \delta$ and $d_\infty(p', x') \leq \delta$, we have $d_\infty(p, x) \leq \delta$. Since $q \in S_{v_j}$, we have

$$d_\infty(p, q) \geq q_1 - p_1 \geq r_1 - p_1 = \delta.$$

This implies that $d_\infty(p, x) \leq d_\infty(p, q)$. Hence, x is also a right-L_∞-neighbor of p. Since $x \in S_v$, the claim of the lemma follows.

Next let $1 \leq j \leq i$ and assume that the set S_{v_j} contains a right-L_∞-neighbor of p in S. If $j = i$, then we are done. If $j < i$, then in the same way as in the proof of Lemma 3.2, it follows that such a neighbor is added to C during the j-th iteration of the while-loop. This completes the proof. □

LEMMA 3.4. *During the while-loop of Stage 3, the set $C \cup S_v$ contains a right-L_∞-neighbor of p in S.*

Proof. The previous lemma implies that the claim holds before the while-loop is entered. Consider an iteration and assume that $C \cup S_v$ contains a right-L_∞-neighbor of p. (We consider C as it is at the beginning of this iteration.) Let x and r be the points that are selected during this iteration. Moreover, let w_l (resp. w_r) be the left (resp. right) son of v.

First assume that $d_\infty(p', x') > |p_1 - r_1|$. We have to show that $C \cup \{x\} \cup S_{w_r}$ contains a right-L_∞-neighbor of p. Since the set $C \cup S_{w_l} \cup S_{w_r}$ contains a right-L_∞-neighbor of p, we only have to consider the case where the set S_{w_l} contains such a neighbor. In this case, it follows in the same way as in the proof of Lemma 3.2 that point x is a right-L_∞-neighbor of p. Since this point is added to C during this iteration, the claim follows.

It remains to consider the case where $d_\infty(p', x') \leq |p_1 - r_1|$. Now we have to show that the set $C \cup S_{w_l}$ contains a right-L_∞-neighbor of p. Assume that S_{w_r} contains such a neighbor. As in the proof of the previous lemma, it follows that x is also a right-L_∞-neighbor of p. But, x is an element of S_{w_l}. This completes the proof. □

Lemmas 3.2 and 3.4 imply that point q^R which is computed in part **2a** is a right-L_∞-neighbor of p in S. Similarly, point q^L computed in part **2b** is a left-L_∞-neighbor of p. This proves that the point that is returned in part **2c** is an L_∞-neighbor of p. Hence, algorithm Neighbor(p, S, D) is correct.

LEMMA 3.5. *For $D \geq 2$, the running time of algorithm* Neighbor(p, S, D) *is bounded by $O((\log n)^{D-1})$.*

Proof. Let $Q(n, D)$ denote the running time on a set of n points in $\mathbb{R}^D$. It is clear that $Q(n, 1) = O(\log n)$. Let $D \geq 2$. Consider part **2a**. Stage 1 takes $O(\log n)$ time. The while-loop in Stage 2 takes time $O(Q(n, D-1)\log n)$. If *stop* $=$ *false* after this loop, then $O(|C|) = O(\log n)$ time is needed to select the point q^R. Hence, Stage 2 takes time $O(Q(n, D-1)\log n)$. Stage 3 takes the same amount of time. Part **2b** can be analyzed in the same way. Clearly, part **2c** takes only constant time. This proves that $Q(n, D) = O(Q(n, D-1)\log n)$, implying that $Q(n, D) = O((\log n)^D)$.

Now consider the planar case. The algorithm follows a path in the main tree and locates the y-coordinate of the query point in the associated structure—a binary search tree—of some of the nodes on this path. It is well known that layering or fractional cascading (see [6, 15]) can be applied to improve the query time from $O((\log n)^2)$ to $O(\log n)$, i.e., $Q(n, 2) = O(\log n)$. As a result, the query time for the D-dimensional case, where $D \geq 2$, is improved to $O((\log n)^{D-1})$. □

Hence, we have a data structure for the L_∞-neighbor problem that has a query time of $O((\log n)^{D-1})$ and that uses $O(n(\log n)^{D-1})$ space. We can improve the space bound by noting that the planar version can be solved by means of the L_∞-Voronoi diagram. (See Lee [11].) That is, the planar L_∞-neighbor problem has a solution with a query time of $O(\log n)$ using only $O(n)$ space. As a result, the D-dimensional problem can be solved with a query time of $O((\log n)^{D-1})$ using $O(n(\log n)^{D-2})$ space.

Until now, we only considered the static version of the problem. We saw in Theorem 2.2 that range trees can be maintained under insertions and deletions of points. Therefore, our solution that is based on the range tree only can be used for the dynamic problem. Because of dynamic fractional cascading, the query time increases by a factor of $O(\log\log n)$. This proves the following result.

THEOREM 3.6. *Let S be a set of n points in $\mathbb{R}^D$. There exists a data structure of size $O(n(\log n)^{D-2})$ that, given a query point $p \in \mathbb{R}^D$, finds an L_∞-neighbor of p in S in $O((\log n)^{D-1})$ time.*

For the dynamic version of the problem, there is a structure of size $O(n(\log n)^{D-1})$

having $O((\log n)^{D-1} \log\log n)$ query time and $O((\log n)^{D-1} \log\log n)$ amortized update time.

Remark: Let c be a positive integer. The range tree can also be used to compute the c L_∞-neighbors of a query point. The algorithm is basically the same. If c is a constant, the query time remains the same as in the above theorem.

4. The approximate L_2-neighbor problem. We mentioned in the introduction that the L_2-nearest-neighbor problem is hard to solve exactly. Therefore, it is natural to consider a weaker version of the problem.

Let S be a set of n points in $\mathbb{R}^D$ and let $\epsilon > 0$ be a fixed constant. For any point $p \in \mathbb{R}^D$, we denote by p^* its L_2-neighbor in S, i.e., p^* is a point of S such that $d_2(p, p^*) = \min\{d_2(p,q) : q \in S\}$. A point $q \in S$ is called a $(1+\epsilon)$-*approximate L_2-neighbor* of p if $d_2(p,q) \leq (1+\epsilon) d_2(p,p^*)$.

We want to store the set S in a data structure such that for any query point $p \in \mathbb{R}^D$ we can find a $(1+\epsilon)$-*approximate L_2-neighbor* of it.

Let $p \in \mathbb{R}^D$ and let q be an L_∞-neighbor of p in S. Let $\delta := d_\infty(p, p^*)$ and consider the D-dimensional axes-parallel box centered at p having sides of length 2δ. Clearly, q lies inside or on the boundary of this box. Therefore, $d_2(p,q) \leq \sqrt{D} \cdot \delta$. Since $\delta = d_\infty(p,p^*) \leq d_2(p,p^*)$, we infer that $d_2(p,q) \leq \sqrt{D} \cdot d_2(p,p^*)$, i.e., q is a $\sqrt{D}$-approximate L_2-neighbor of p.

This shows that we can solve the $\sqrt{D}$-approximate L_2-neighbor problem using the results of the previous section.

We extend this solution. First note that the L_∞-metric depends on the coordinate system: If we rotate the (XY)-system, then the L_∞-metric changes. The L_2-metric, however, is invariant under such rotations. We store the set S in a constant number of range trees, where each range tree stores the points according to its own coordinate system. Then given p, we use the range trees to compute L_∞-neighbors in all coordinate systems. As we will see, one of these L_∞-neighbors is a $(1+\epsilon)$-approximate L_2-neighbor of p.

Let $(\mathcal{F}_i)$ be a family of orthonormal coordinate systems all sharing the same origin. The coordinates of any point x in the system $\mathcal{F}_i$ are denoted by $x_{i1}, x_{i2}, \ldots, x_{iD}$. Moreover, $d_{i,\infty}(\cdot,\cdot)$ denotes the L_∞-distance function in $\mathcal{F}_i$. Assume that for any point x in D-dimensional space there is an index i such that for all $1 \leq j \leq D$,

$$0 \leq x_{ij} \leq x_{iD} \leq (1+\epsilon) x_{ij}, \tag{1}$$

i.e., in $\mathcal{F}_i$ all coordinates of x are non-negative and almost equal. Note that the family $(\mathcal{F}_i)$ is independent of the set S.

In Yao [26], it is shown how such a family consisting of $O((c/\epsilon)^{D-1})$ coordinate systems can be constructed. (Here, c is a constant.) In the planar case, such a family is easily obtained: Let $0 < \phi < \pi/4$ be such that $\tan\phi = \epsilon/(2+\epsilon)$. For $0 \leq i < 2\pi/\phi$, let X_i (resp. Y_i) be the directed line that makes an angle of $i \cdot \phi$ with the positive X-axis (resp. Y-axis). Then, $\mathcal{F}_i$ is the (X_iY_i)-coordinate system.

To prove that (1) holds for this family, let x be any point in the plane and let γ be the angle that the line segment $\vec{x}$ between the origin and x makes with the positive X-axis.

First assume that $\pi/4 \leq \gamma < 2\pi$. Let $i := \lfloor(\gamma - \pi/4)/\phi\rfloor$ and let γ_i be the angle between $\vec{x}$ and the positive X_i-axis. Then $\gamma_i = \gamma - i \cdot \phi$ and this angle lies in between $\pi/4$ and $\pi/4 + \phi$. Therefore, the coordinates x_{i1} and x_{i2} of x in $\mathcal{F}_i$ satisfy

$$x_{i2}/x_{i1} = \tan\gamma_i \geq \tan\pi/4 = 1$$

and

$$x_{i2}/x_{i1} = \tan\gamma_i \leq \tan(\pi/4+\phi) = \frac{1+\tan\phi}{1-\tan\phi} = 1+\epsilon.$$

If $0 \leq \gamma < \pi/4$, then we take $i := \lfloor (7\pi/4+\gamma)/\phi \rfloor$. In this case, $\gamma_i = 2\pi - i\cdot\phi + \gamma$ and this angle lies in between $\pi/4$ and $\pi/4+\phi$. This proves that x_{i2}/x_1 lies in between 1 and $1+\epsilon$.

Hence, the family $(\mathcal{F}_i)$ satisfies the assumptions made in (1). Moreover, the number of coordinate systems is at most

$$1+\frac{2\pi}{\phi} = 1+\frac{2\pi}{\arctan\epsilon/(2+\epsilon)} = O(1/\epsilon).$$

The data structure for approximate L_2-neighbor queries: For each index i, let S_i denote the set of points in S with coordinates in the system $\mathcal{F}_i$. The data structure consists of a collection of range trees; the i-th range tree stores the set S_i.
Finding an approximate L_2-neighbor: Let $p \in \mathbb{R}^D$ be a query point. For each index i, use the i-th range tree to find an L_∞-neighbor $q^{(i)}$ of p in S_i. Report an L_∞-neighbor that has minimal L_2-distance to p.

LEMMA 4.1. *The query algorithm reports a $(1+\epsilon)$-approximate L_2-neighbor of p.*

Proof. Consider an exact L_2-neighbor p^* of p. Let i be an index such that the coordinates of p^*-p in $\mathcal{F}_i$ satisfy

$$0 \leq p_{ij} - p^*_{ij} \leq p_{iD} - p^*_{iD} \leq (1+\epsilon)(p_{ij}-p^*_{ij}), \quad 1 \leq j \leq D.$$

Let q be the point that is reported by the algorithm. Since $d_2(p,q) \leq d_2(p,q^{(i)})$, it suffices to prove that $d_2(p,q^{(i)}) \leq (1+\epsilon)d_2(p,p^*)$.

Let B be the D-dimensional box centered at p with sides of length $2d_{i,\infty}(p,p^*)$ that are parallel to the axes of $\mathcal{F}_i$. Since $q^{(i)}$ is an L_∞-neighbor of p in S_i, this point must lie inside or on the boundary of B. It follows that

$$d_2(p,q^{(i)}) \leq \sqrt{D}\cdot d_{i,\infty}(p,p^*).$$

Note that $d_{i,\infty}(p,p^*) = p_{iD} - p^*_{iD}$. Moreover,

$$(d_2(p,p^*))^2 = \sum_{j=1}^{D}(p_{ij}-p^*_{ij})^2 \geq \sum_{j=1}^{D}\left(\frac{p_{iD}-p^*_{iD}}{1+\epsilon}\right)^2 = \frac{D}{(1+\epsilon)^2}(d_{i,\infty}(p,p^*))^2.$$

Hence,

$$d_2(p,q^{(i)}) \leq \sqrt{D}\cdot d_{i,\infty}(p,p^*) \leq \sqrt{D}\cdot\frac{1+\epsilon}{\sqrt{D}}\cdot d_2(p,p^*) = (1+\epsilon)d_2(p,p^*).$$

This proves the lemma. □

Hence, we have solved the $(1+\epsilon)$-approximate L_2-neighbor problem. The complexity of our solution follows immediately from Theorem 3.6. We have proved the following theorem.

THEOREM 4.2. *Let S be a set of n points in $\mathbb{R}^D$ and let ϵ be a positive constant. There exists a data structure of size $O(n(\log n)^{D-2})$ that, given a query point $p \in \mathbb{R}^D$, finds a $(1+\epsilon)$-approximate L_2-neighbor of p in S in $O((\log n)^{D-1})$ time.*

For the dynamic version of the problem, there is a data structure with a query time of $O((\log n)^{D-1} \log\log n)$ *and an amortized update time of* $O((\log n)^{D-1} \log\log n)$ *that uses space* $O(n(\log n)^{D-1})$.

In all complexity bounds, the constant factor is proportional to $(c/\epsilon)^{D-1}$ *for some fixed* c.

5. The containment problem for boxes of constant overlap. A *box* is a D-dimensional axes-parallel cube, i.e., it is of the form

$$[a_1 : a_1 + \delta] \times [a_2 : a_2 + \delta] \times \ldots \times [a_D : a_D + \delta],$$

for real numbers $a_1, a_2, \ldots, a_D$ and $\delta > 0$. The *center* of this box is the point $(a_1 + \delta/2, a_2 + \delta/2, \ldots, a_D + \delta/2)$.

Let S be a set of n boxes in $\mathbb{R}^D$ that are of *constant overlap*, i.e., there is an integer constant c_D—possibly depending on the dimension—such that each box B of S contains the centers of at most c_D boxes in its interior. (Here, we also count the center of B itself. Note that many boxes may have their centers on the boundary of B.)

We want to store these boxes in a data structure such that for any query point $p \in \mathbb{R}^D$, we can find all boxes that contain p. Note that we distinguish between "being contained in a box" and "being contained in the interior of a box". The following lemma shows that a point p can be contained in only a constant number of boxes.

LEMMA 5.1. *Any point* $p \in \mathbb{R}^D$ *is contained in at most* $2^{1+D^2} c_D$ *boxes of* S.

Proof. The proof is by induction on D. So let $D = 1$. Assume w.l.o.g. that $p = 0$. Let S_+ be the set of all boxes of S whose centers are positive. Let $b_1, b_2, \ldots, b_m$ be the centers of the boxes in S_+ that contain p. Assume w.l.o.g. that $b_1 = \max\{b_i : 1 \leq i \leq m\}$. The interior of the box having b_1 as its center contains all centers b_i, $1 \leq i \leq m$. Hence, the constant overlap property implies that $m \leq c_1$, i.e., point p is contained in at most c_1 boxes of S_+.

By a symmetric argument, point p is contained in at most c_1 boxes of S whose centers are negative. It remains to consider the boxes having p as their centers. The constant overlap property directly implies that there are at most c_1 such boxes.

This proves that there are at most $3c_1$ boxes in the entire set S that contain p.

Now let $D \geq 2$ and assume the lemma holds for dimension $D - 1$. Again, we assume w.l.o.g. that p is the origin. Let S_+ be the set of all boxes of S whose centers have positive coordinates. We will show that there are at most c_D boxes in S_+ that contain p.

Let $b_1, b_2, \ldots, b_m$ be the centers of the boxes in S_+ that contain p. Assume w.l.o.g. that

$$\delta := d_\infty(p, b_1) = \max\{d_\infty(p, b_i) : 1 \leq i \leq m\}.$$

Let B be the box of S_+ having b_1 as its center. We claim that $d_\infty(b_1, b_i) < \delta$ for $1 \leq i \leq m$. Indeed, let $1 \leq j \leq D$. The j-th coordinate b_{ij} of b_i satisfies $0 < b_{ij} \leq \delta$. As a result, $|b_{1j} - b_{ij}| < \delta$. This proves the claim.

Since B contains p, it has sides of length at least 2δ. Therefore, since each center b_i has L_∞-distance less than δ to b_1, these centers are contained in the interior of B. Then, the constant overlap property implies that $m \leq c_D$.

This proves that p is contained in at most c_D boxes of S_+. By a symmetric argument, p is contained in at most c_D boxes of S that have their centers in a fixed D-dimensional quadrant. Hence, p is contained in at most $2^D c_D$ boxes whose centers have non-zero coordinates.

Let S_0 be the set of all boxes in S whose centers have zero as their first coordinate. Consider the set S_0' of $(D-1)$-dimensional boxes obtained from S_0 by deleting from each box its first coordinates. The constant overlap property for S_0 implies that the boxes of S_0' are also of constant overlap, with constant c_D. Hence, by the induction hypothesis, point $p' = (p_2, \ldots, p_D)$ is contained in at most $2^{1+(D-1)^2} c_D$ boxes of S_0'. These boxes of S_0' correspond exactly to the boxes of S_0 that contain p.

Hence, at most $2^{1+(D-1)^2} c_D$ boxes of S_0 contain p. It follows from a symmetric argument that for each $1 \leq i \leq D$, point p is contained in at most $2^{1+(D-1)^2} c_D$ boxes whose centers have zero as their i-th coordinate.

To summarize, we have shown that there are at most

$$2^D c_D + D\, 2^{1+(D-1)^2} c_D \leq 2^{1+D^2} c_D$$

boxes of S that contain p. This completes the proof. □

In the next two subsections, we shall give two solutions for the box containment problem.

5.1. A solution based on segment trees. We start with the one-dimensional case. Let S be a set of n intervals $[a_j : b_j]$, $1 \leq j \leq n$, that are of constant overlap with constant c.

The one-dimensional structure: We store the intervals in the leaves of a balanced binary search tree T, sorted by their right endpoints. The leaves of this tree are threaded in a doubly-linked list.

The query algorithm: Let $p \in \mathbb{R}$ be a query element. Search in T for the leftmost leaf containing a right endpoint that is at least equal to p. Starting in this leaf, walk along the leaves to the right and report all intervals encountered that contain p. Stop walking as soon as $4c$ intervals have been reported, or c intervals have been encountered that do not contain p.

LEMMA 5.2. *The above data structure solves the one-dimensional containment problem in a set of n intervals of constant overlap in $O(n)$ space with a query time of $O(\log n)$. Intervals can be inserted and deleted in $O(\log n)$ time.*

Proof. The complexity bounds are clear; we only have to prove that the query algorithm is correct. Clearly, all intervals that are reported contain p. It remains to show that all intervals containing p are reported.

Let v be the leaf of T in which the search for p ends. The algorithm starts in v and walks to the right. First note that all leaves to the left of v store intervals that do not contain p. We know from Lemma 5.1 that there are at most $4c$ intervals that contain p. Hence, after $4c$ intervals have been reported, the algorithm can stop. Now assume that in leaf w, we encounter the c-th interval that does not contain p. All c encountered intervals that do not contain p lie completely to the right of p. If there is a leaf to the right of w whose interval contains p, then this interval contains these c intervals. By the constant overlap property, such a leaf cannot exist. This proves that the algorithm can stop in leaf w, i.e., if it has encountered c intervals that do not contain p. □

We next consider the D-dimensional case, where $D \geq 2$. Let S be a set of n boxes in $\mathbb{R}^D$ that are of constant overlap with constant c_D. The data structure is a segment tree for the intervals of the first coordinates of the boxes. (See [13, 15].) The nodes of this tree contain appropriate associated structures.

If $B = B_1 \times B_2 \times \ldots \times B_D$ is a box in $\mathbb{R}^D$, then B' denotes the box $B_2 \times \ldots \times B_D$ in $\mathbb{R}^{D-1}$. Similarly, S' denotes the set $\{B' : B \in S\}$. Recall that we use a similar notation for points.

The D-dimensional structure for $D \geq 2$: Let $a_1 < a_2 < \ldots < a_m$, where $m \leq 2n$, be the sorted sequence of all distinct endpoints of the intervals of the first coordinates of the boxes in S. We store the *elementary intervals*

$$(-\infty : a_1), [a_1 : a_1], (a_1 : a_2), [a_2 : a_2], \ldots, (a_{m-1} : a_m), [a_m : a_m], (a_m : \infty)$$

in this order in the leaves of a balanced binary search tree, called the *main tree*. Each node v of this tree has associated with it an interval I_v being the union of the elementary intervals of the leaves in the subtree of v. Let S_v be the set of all boxes $B = B_1 \times B_2 \times \ldots \times B_D$ in S such that B_1 spans the interval associated with v but does not span the interval associated with its father node, i.e. $I_v \subseteq B_1$ and $I_{f(v)} \not\subseteq B_1$, where $f(v)$ is the father of v.

Each node v of the main tree contains (a pointer to) an *associated structure* for the set S_v: Partition this set into S_{vl}, S_{vc} and S_{vr}, consisting of those boxes of S_v whose centers lie to the left of the "vertical" slab $I_v \times \mathbb{R}^{D-1}$, in or on the boundary of this slab, and to the right of this slab, respectively.

The associated structure of v consists of three $(D-1)$-dimensional structures for the sets S'_{vl}, S'_{vc} and S'_{vr}, that are defined recursively.

Remark: The interval I_v may be open, half-open or closed. Therefore, the boundary of the slab $I_v \times \mathbb{R}^{D-1}$ does not necessarily belong to this slab.

The query algorithm: Let $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$ be a query point. Search in the main tree for the elementary interval that contains p_1. For each node v on the search path, recursively perform a $(D-1)$-dimensional query with the point p' in the three structures that are stored with v.

At the last level of the recursion, a one-dimensional query is performed in a binary search tree. In this tree, the algorithm stops if it has reported $2^{1+D^2} c_D$ boxes of S that contain p, or if it has encountered c_D boxes of S that do not contain p. (If *in this tree* c_D boxes that do not contain p have been encountered, then the algorithm only stops at *this* level of the recursion. If, on the other hand, overall $2^{1+D^2} c_D$ boxes that contain p have been reported, then the *entire* query algorithm stops.)

The correctness proof of this algorithm uses the following lemma.

LEMMA 5.3. *Let x and y be real numbers, let $p \in \mathbb{R}^D$ and let i, $1 \leq i \leq D-1$, be an integer such that $x \leq p_i \leq y$. Let $A = [a_1 : a_1 + \alpha] \times \ldots \times [a_D : a_D + \alpha]$ and $B = [b_1 : b_1 + \beta] \times \ldots \times [b_D : b_D + \beta]$ be two boxes such that $a_i \leq x$, $a_i + \alpha \geq y$, $b_i \leq x$, $b_i + \beta \geq y$, $b_D > p_D$ and $b_D + \beta \leq a_D + \alpha$. Finally, assume that p is contained in A. Then,*

1. *$a_D < b_D + \beta/2 < a_D + \alpha$,*
2. *if one of the following three conditions holds,*
   (a) $x \leq a_i + \alpha/2 \leq y$ and $x \leq b_i + \beta/2 \leq y$,
   (b) $a_i + \alpha/2 < x$ and $b_i + \beta/2 < x$,
   (c) $a_i + \alpha/2 > y$ and $b_i + \beta/2 > y$,
   then $a_i < b_i + \beta/2 < a_i + \alpha$.

Proof. The right inequality of the first assertion follows easily: $b_D + \beta/2 < b_D + \beta \leq a_D + \alpha$. Since p is contained in A, we have $a_D \leq p_D$. Therefore, $a_D \leq p_D < b_D < b_D + \beta/2$. This proves 1.

Before we prove 2, observe that $b_D + \beta \leq a_D + \alpha < b_D + \alpha$. It follows that $\beta < \alpha$.

Assume that case 2(a) applies, i.e., $x \leq a_i + \alpha/2 \leq y$ and $x \leq b_i + \beta/2 \leq y$. Since $b_i + \beta/2 \leq y \leq a_i + \alpha$ and $b_i + \beta/2 \geq x \geq a_i$, we only have to show that $a_i \neq b_i + \beta/2$ and $b_i + \beta/2 \neq a_i + \alpha$.

Assume that $a_i = b_i + \beta/2$. Then, $x = a_i = b_i + \beta/2$. First note that $\alpha/2 \leq y - a_i = y - x$. Since $\beta/2 = b_i + \beta - (b_i + \beta/2) \geq y - (b_i + \beta/2) = y - x$, we infer that $\alpha \leq \beta$. This is a contradiction and, hence, $a_i < b_i + \beta/2$.

Next assume that $b_i + \beta/2 = a_i + \alpha$. Then, $y = b_i + \beta/2 = a_i + \alpha$ and in a similar way we can prove that $\alpha \leq \beta$. Therefore, $b_i + \beta/2 < a_i + \alpha$.

Next consider case 2(b), i.e., assume that $a_i + \alpha/2 < x$ and $b_i + \beta/2 < x$. Since $b_i + \beta/2 < x \leq y \leq a_i + \alpha$, we only have to show that $a_i < b_i + \beta/2$. We prove this by contradiction. So, assume that $a_i \geq b_i + \beta/2$. Then, $\alpha/2 = a_i + \alpha/2 - a_i < x - a_i \leq x - (b_i + \beta/2) = x - b_i - \beta/2$. Since $b_i + \beta \geq y \geq x$, we get $x - b_i \leq \beta$. Therefore, $\alpha/2 < x - b_i - \beta/2 \leq \beta - \beta/2 = \beta/2$, i.e., $\alpha < \beta$. This is a contradiction and, hence, we have proved that $a_i < b_i + \beta/2$.

Case 2(c) can be treated in the same way as case 2(b). ∎

LEMMA 5.4. *The query algorithm is correct.*

Proof. It is well known that the set of all boxes $B = B_1 \times \ldots \times B_D$ such that $p_1 \in B_1$ is exactly the union of all sets S_v, where v is a node on the search path to p_1. Hence, we only have to consider the nodes on this search path.

Let v be a node on the path to p_1. We know that p_1 is contained in the first interval of each box of S_v. Hence, we have to find all boxes in S_v whose last $D-1$ intervals contain $(p_2, \ldots, p_D)$. We claim that the recursive queries in the three structures stored with v find these boxes. By Lemma 5.1, there are at most $2^{1+D^2} c_D$ boxes that contain p. Hence, at the last level of the recursion, the query algorithm can stop as soon as it has reported this many boxes. It remains to prove that, at the last level, the query algorithm can stop if it has encountered c_D boxes that do not contain p.

Consider such a last level. That is, let $v_1, v_2, \ldots, v_{D-1}$ be nodes such that

1. $v_1 = v$,
2. v_i is a node of the main tree of one of the three structures that are stored with v_{i-1},
3. v_i lies on the search path to p_i.

The algorithm makes one-dimensional queries with p_D in the three structures—binary search trees—that are stored with v_{D-1}.

Consider one such query. The algorithm searches for p_D. Let r be the leaf in which this search ends. Starting in r, the algorithm walks along the leaves to the right. During this walk, it encounters boxes that do or do not contain p. Assume that in leaf s, the c_D-th box is encountered that does not contain p. We have to show that the algorithm can stop in s. That is, we must show that all leaves to the right of s store boxes that do not contain p.

Assume this is not the case. Then, there is a box

$$A = [a_1 : a_1 + \alpha] \times [a_2 : a_2 + \alpha] \times \ldots \times [a_D : a_D + \alpha]$$

that is stored in a leaf to the right of s and that contains p. Let

$$B^{(j)} = [b_{j1} : b_{j1} + \beta_j] \times [b_{j2} : b_{j2} + \beta_j] \times \ldots \times [b_{jD} : b_{jD} + \beta_j], 1 \leq j \leq c_D,$$

be the encountered boxes between r and s that do not contain p. We shall prove that A contains the centers of all these boxes in its interior. This will be a contradiction.

Let $1 \leq j \leq c_D$. Since the leaf of A lies to the right of the leaf of $B^{(j)}$, we know that $b_{jD} + \beta_j \leq a_D + \alpha$.

Let $1 \leq i \leq D-1$ and let $x \leq y$ be the boundary points of the interval I_{v_i}. Then, $x \leq p_i \leq y$, because $p_i \in I_{v_i}$. Moreover, we know that I_{v_i} is contained in the i-th interval of A and $B^{(j)}$, i.e., $a_i \leq x$, $a_i + \alpha \geq y$, $b_{ji} \leq x$ and $b_{ji} + \beta_j \geq y$. Finally, since the leaf of $B^{(j)}$ is equal to leaf r or lies to the right of it, we know that $p_D \leq b_{jD} + \beta_j$. Since $B^{(j)}$ does not contain p, this implies that $b_{jD} > p_D$. (Note that the i-th interval of $B^{(j)}$ contains p_i, $1 \leq i \leq D-1$.)

The definition of our data structure implies that the i-th coordinates of the centers of A and $B^{(j)}$ are either both on the boundary or contained in I_{v_i}, or both are less than x, or both are larger than y. Therefore, all requirements of Lemma 5.3 are satisfied and we conclude that $a_i < b_{ji} + \beta_j/2 < a_i + \alpha$.

Since i was arbitrary between one and $D-1$, and since Lemma 5.3 also implies that $a_D < b_{jD} + \beta_j/2 < a_D + \alpha$, we have proved that the center of $B^{(j)}$ is contained in the interior of A. This completes the proof. □

We analyze the complexity of the D-dimensional structure. Let $M(n, D)$ denote the size of the data structure. It is well known that the first interval of a box in S is stored in the associated structure of $O(\log n)$ nodes of the main tree. This implies that $M(n, D) = O(M(n, D-1) \log n)$. Since $M(n, 1) = O(n)$, we get $M(n, D) = O(n(\log n)^{D-1})$. Using presorting, the structure can be built in $O(n(\log n)^{D-1})$ time.

Let $Q(n, D)$ denote the query time. Then $Q(n, 1) = O(\log n)$. The query algorithm performs $(D-1)$-dimensional queries in each of the $O(\log n)$ nodes on the search path. As a result, the query time satisfies $Q(n, D) = O(Q(n, D-1) \log n)$, which solves to $O((\log n)^D)$. By applying fractional cascading (see [6]) in the 2-dimensional case, we decrease $Q(n, 2)$ to $O(\log n)$. This improves the query time to $Q(n, D) = O((\log n)^{D-1})$ for $D \geq 2$.

By applying standard techniques, the data structure can be adapted to handle insertions and deletions of boxes. (See [13, 14].) Because of dynamic fractional cascading, the query time increases by a factor of $O(\log \log n)$, whereas the size only increases by a constant factor. The amortized update time is $O((\log n)^{D-1} \log \log n)$.

We summarize our result.

THEOREM 5.5. *Let S be a set of n boxes in $\mathbb{R}^D$ of constant overlap. There exists a data structure of size $O(n(\log n)^{D-1})$ such that for any point $p \in \mathbb{R}^D$, we can find all boxes of S that contain p in $O((\log n)^{D-1})$ time. This static data structure can be built in $O(n(\log n)^{D-1})$ time.*

For the dynamic version of the problem, the query time is $O((\log n)^{D-1} \log \log n)$ and the size of the structure is $O(n(\log n)^{D-1})$. Boxes can be inserted and deleted in $O((\log n)^{D-1} \log \log n)$ amortized time.

5.2. A solution based on skewer trees. In this section, we give a linear space solution to the box containment problem. This solution is based on skewer trees, introduced by Edelsbrunner et al. [8]. (See also Smid [21] for a dynamic version of this data structure.)

Let S be a set of n boxes in $\mathbb{R}^D$ that are of constant overlap with constant c_D. For $D = 1$, the data structure is the same as in the previous subsection.

The D-dimensional structure for $D \geq 2$: If S is empty, then the data structure is also empty. Assume that S is non-empty. Let γ be the median of the set

$$\{(a_1 + b_1)/2 : [a_1 : b_1] \times [a_2 : b_2] \times \ldots \times [a_D : b_D] \in S\},$$

and let ϵ be the largest element that is less than γ in the set of all elements a_1, $(a_1 + b_1)/2$ and b_1, where $[a_1 : b_1] \times \ldots \times [a_D : b_D]$ ranges over S. Let $\gamma_1 := \gamma - \epsilon/2$ and let σ be the hyperplane in $\mathbb{R}^D$ with equation $x_1 = \gamma_1$.

Let $S_<$, S_0 and $S_>$ be the set of boxes $[a_1 : b_1] \times \ldots \times [a_D : b_D]$ in S such that $b_1 < \gamma_1$, $a_1 \leq \gamma_1 \leq b_1$ and $\gamma_1 < a_1$, respectively. The D-dimensional data structure for the set S is an augmented binary search tree—called the *main tree*—having the following form:

1. The root contains the hyperplane σ.
2. The root contains pointers to its left and right sons, which are D-dimensional structures for the sets $S_<$ and $S_>$, respectively.
3. The root contains (a pointer to) an *associated structure* for the set S_0: Partition this set into S_{0l} and S_{0r}, consisting of those boxes in S_0 whose centers lie to the left of the hyperplane σ, and to the right of σ, respectively.
   The associated structure of the root consists of two $(D-1)$-dimensional structures for the sets S'_{0l} and S'_{0r}.

Remark: By our choice of γ_1, the hyperplane σ does not contain the center of any box in S. Moreover, each of the sets $S_<$ and $S_>$ has size at most $n/2$. The set S_0 may have size n. The *height* of the data structure is defined as the height of its main tree. It follows that the structure has height $O(\log n)$.

The query algorithm: Let $p = (p_1, p_2, \ldots, p_D) \in \mathbb{R}^D$ be a query point. Let $\sigma : x_1 = \gamma_1$ be the hyperplane stored in the root of the main tree. Recursively perform a $(D-1)$-dimensional query with the point p' in the two structures that are stored with the root.

If $p_1 < \gamma_1$ (resp. $p_1 > \gamma_1$), then recursively perform a D-dimensional query with p in the left (resp. right) subtree of the root, unless this subtree is empty, in which case the algorithm stops.

At the last level of the recursion, a one-dimensional query is performed in a binary search tree. In this tree, the algorithm stops if it has reported $2^{1+D^2}c_D$ boxes of S that contain p, or if it has encountered c_D boxes of S whose last intervals do not contain p_D, or if it has encountered $2^{1+D^2}c_D$ boxes of S that do not contain p but whose last intervals contain p_D.

The correctness proof is similar to the one of Subsection 5.1. Again, we start with a technical lemma.

LEMMA 5.6. *Let $p \in \mathbb{R}^D$ and let $\sigma_i : x_i = \gamma_i$, $1 \le i \le D-1$, be hyperplanes in $\mathbb{R}^D$. Let $A = [a_1 : a_1+\alpha] \times \ldots \times [a_D : a_D+\alpha]$ and $B = [b_1 : b_1+\beta] \times \ldots \times [b_D : b_D+\beta]$ be two boxes such that $b_D > p_D$ and $b_D + \beta \le a_D + \alpha$. Assume that p is contained in A. Finally, assume that $a_i + \alpha/2 < \gamma_i \le a_i + \alpha$ and $b_i + \beta/2 < \gamma_i \le b_i + \beta$ for all $1 \le i \le D-1$.*

Then $a_i < b_i + \beta/2 < a_i + \alpha$ for all $1 \le i \le D$.

Proof. We start with $i = D$. It follows directly that $b_D + \beta/2 < b_D + \beta \le a_D + \alpha$. Since p is contained in A, we have $a_D \le p_D$. Therefore, $a_D \le p_D < b_D < b_D + \beta/2$.

Let $1 \le i \le D-1$. Then $b_i + \beta/2 < \gamma_i \le a_i + \alpha$. It remains to show that $a_i < b_i + \beta/2$. Assume that $b_i + \beta/2 \le a_i$. Then, $\beta/2 = b_i + \beta - (b_i + \beta/2) \ge \gamma_i - (b_i + \beta/2) \ge \gamma_i - a_i > \alpha/2$, i.e., $\beta > \alpha$. But, since $b_D + \beta \le a_D + \alpha < b_D + \alpha$, we also have $\beta < \alpha$. This is a contradiction. □

LEMMA 5.7. *The query algorithm is correct.*

Proof. It is clear that the algorithm branches correctly. Hence, we only have to consider the last level of the recursion. Since there are at most $2^{1+D^2}c_D$ boxes that contain p, the algorithm can stop as soon as it has reported this many boxes. It remains to prove that, at the last level, the query algorithm can stop if it has encountered c_D boxes whose last intervals do not contain p_D, or if it has encountered $2^{1+D^2}c_D$ boxes that do not contain p but whose last intervals contain p_D.

Consider such a last level. That is, let $v_1, v_2, \ldots, v_{D-1}$ be nodes such that

1. v_1 is a node of the main tree,

2. v_i is a node of the main tree of one of the two structures that are stored with v_{i-1},
3. v_i lies on the search path to p_i.

Let $\sigma_i : x_i = \gamma_i$ be the hyperplane that is stored with v_i, $1 \leq i \leq D-1$.

The algorithm makes one-dimensional queries with p_D in the two structures—binary trees—that are stored with v_{D-1}. Consider one such query. The algorithm searches for p_D. Let r be the leaf in which this search ends. Starting in r, the algorithm walks along the leaves to the right. Assume that in leaf s, the c_D-th box is encountered whose last interval does not contain p_D, or the $(2^{1+D^2}c_D)$-th box is encountered that does not contain p but whose last interval contains p_D. We will prove that all leaves to the right of s store boxes that do not contain p.

Assume this is not the case. Then, there is a box

$$A = [a_1 : a_1 + \alpha] \times [a_2 : a_2 + \alpha] \times \ldots \times [a_D : a_D + \alpha]$$

that is stored in a leaf to the right of s and that contains p. Let

$$B^{(j)} = [b_{j1} : b_{j1} + \beta_j] \times [b_{j2} : b_{j2} + \beta_j] \times \ldots \times [b_{jD} : b_{jD} + \beta_j]$$

be the encountered boxes between r and s that do not contain p.

The definition of our data structure implies that for each $1 \leq i \leq D-1$, the centers of A and the $B^{(j)}$'s lie on the same side of the hyperplane σ_i. Moreover, these boxes intersect σ_i. We assume w.l.o.g. that for all $1 \leq i \leq D-1$, $a_i + \alpha/2 < \gamma_i \leq a_i + \alpha$, and for all j and all $1 \leq i \leq D-1$, $b_{ji} + \beta_j/2 < \gamma_i \leq b_{ji} + \beta_j$.

There are two possible cases. First assume that the last intervals of c_D boxes $B^{(j)}$ do not contain p_D. Consider such a box $B^{(j)}$. Since the leaf of this box is equal to leaf r or lies to the right of it, we must have $p_D \leq b_{jD} + \beta_j$. Hence, $p_D < b_{jD}$. Also, since the leaf of A lies to the right of the leaf of $B^{(j)}$, we have $b_{jD} + \beta_j \leq a_D + \alpha$. Hence, all requirements of Lemma 5.6 are satisfied. We conclude that the center of $B^{(j)}$ is contained in the interior of A. This proves that the interior of A contains more than c_D centers, namely its own center and the centers of c_D boxes $B^{(j)}$. This is a contradiction.

The second case is where $2^{1+D^2}c_D$ boxes $B^{(j)}$ do not contain p but their last intervals contain p_D. We claim that the point $q := (\gamma_1, \gamma_2, \ldots, \gamma_{D-1}, p_D)$ is contained in A and in all these $B^{(j)}$'s. To prove this, first note that $a_D \leq p_D \leq a_D + \alpha$, because A contains p. Consider any of these boxes $B^{(j)}$. By our assumption, $b_{jD} \leq p_D \leq b_{jD} + \beta_j$. Our assumptions also imply that $a_i \leq \gamma_i \leq a_i + \alpha$ and $b_{ji} \leq \gamma_i \leq b_{ji} + \beta_j$ for all $1 \leq i \leq D-1$. Hence, there are more than $2^{1+D^2}c_D$ boxes that contain q. This contradicts Lemma 5.1. □

The complexity analysis is similar to that of the previous subsection. Since the main tree has height $O(\log n)$, the query time is bounded by $O((\log n)^D)$. In the planar case, we can apply fractional cascading to improve the query time to $O(\log n)$. Then, the query time for the D-dimensional case is improved to $O((\log n)^{D-1})$. It is easy to see that the data structure has size $O(n)$ and that it can be built in $O(n \log n)$ time. (See [8] for details.)

We can adapt the data structure such that it can also handle insertions and deletions of boxes. Since the algorithms and their running times are exactly the same as in [21], we refer the reader to that paper for the details. Because of dynamic fractional cascading, the query time increases by a factor of $O(\log \log n)$, whereas the size only increases by a constant factor. The amortized update time is $O((\log n)^2 \log \log n)$.

We summarize our result.

THEOREM 5.8. *Let S be a set of n boxes in $\mathbb{R}^D$ of constant overlap. There exists a data structure of size $O(n)$ such that for any point $p \in \mathbb{R}^D$, we can find all boxes of S that contain p in $O((\log n)^{D-1})$ time. This static data structure can be built in $O(n \log n)$ time.*

For the dynamic version of the problem, the query time is $O((\log n)^{D-1} \log\log n)$ and the size of the structure is $O(n)$. Boxes can be inserted and deleted in amortized time $O((\log n)^2 \log\log n)$.

6. Maintaining the closest pair. In this section, we apply the results obtained so far to maintain a closest pair of a point set under insertions and deletions. Let S be a set of n points in $\mathbb{R}^D$ and let $1 \leq t \leq \infty$ be a real number. We denote the L_t-distance between any two points p and q in $\mathbb{R}^D$ by $d(p,q)$. The pair $P, Q \in S$ is called a *closest pair* of S if

$$d(P,Q) = \min\{d(p,q) : p,q \in S, p \neq q\}.$$

We introduce the following notations. For any point $p \in \mathbb{R}^D$, $box(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S \setminus \{p\}$. In other words, the side length of $box(p)$ is twice the L_∞-distance between p and its $(1+(2D+2)^D)$-th (resp. $(2D+2)^D$-th) L_∞-neighbor, if $p \in S$ (resp. $p \notin S$).

Let $N(p)$ be the set of points of $S \setminus \{p\}$ that are contained in the interior of $box(p)$. Note that $N(p)$ has size less than $(2D+2)^D$. In fact, $N(p)$ may even be empty.

Our data structure is based on the following lemma.

LEMMA 6.1. *The set $\{(p,q) : p \in S, q \in N(p)\}$ contains a closest pair of S.*

Proof. Let (P,Q) be a closest pair of S. We have to show that $Q \in N(P)$. Assume this is not the case. Let δ be the side length of $box(P)$. Since Q lies outside or on the boundary of this box, we have $d(P,Q) \geq \delta/2$.

Partition $box(P)$ into $(2D+2)^D$ subboxes with sides of length $\delta/(2D+2)$. Since $box(P)$ contains at least $1+(2D+2)^D$ points of S, one of these subboxes contains at least two points. These two points have distance at most $D \cdot \delta/(2D+2) < \delta/2$, which is a contradiction, because (P,Q) is a closest pair of S. □

The set $\{box(p) : p \in S\}$ is of constant overlap: each box contains the centers of at most $(2D+2)^D$ boxes in its interior. These centers are precisely the points of $N(p) \cup \{p\}$. This fact and the above lemma suggest the following data structure.

The closest pair data structure:

1. The points of S are stored in a range tree.
2. The distances of the multiset $\{d(p,q) : p \in S, q \in N(p)\}$ are stored in a heap. With each distance, we store the corresponding pair of points. (Note that both $d(p,q)$ and $d(q,p)$ may occur in the heap.)
3. The points of S are stored in a dictionary. With each point p, we store a list containing the elements of $N(p)$. For convenience, we also call this list $N(p)$. With each point q in $N(p)$, we store a pointer to the occurrence of $d(p,q)$ in the heap.
4. The set $\{box(p) : p \in S\}$ is stored in the dynamic data structure of Theorem 5.5. This structure is called the *box tree*.

It follows from Lemma 6.1 that the pair of points that is stored with the minimal element of the heap is a closest pair of S. The update algorithms are rather straightforward.

The insertion algorithm: Let $p \in \mathbb{R}^D$ be the point to be inserted. Assume w.l.o.g. that $p \notin S$.

1. Using the range tree, find the $(2D+2)^D$ L_∞-neighbors of p in S. The point among these neighbors having maximal L_∞-distance to p determines $box(p)$. The neighbors that are contained in the interior of $box(p)$ form the list $N(p)$.
2. Insert p into the range tree and insert the distances $d(p,q)$, $q \in N(p)$ into the heap. Then, insert p—together with the list $N(p)$—into the dictionary. With each point q in $N(p)$, store a pointer to $d(p,q)$ in the heap. Finally, insert $box(p)$ into the box tree.
3. Using the box tree, find all boxes that contain p. For each reported element $box(q)$, $q \neq p$, that contains p in its interior, do the following:
   (a) Search in the dictionary for q. Insert p into $N(q)$, insert $d(q,p)$ into the heap and store with p a pointer to $d(q,p)$.
   (b) If $N(q)$ has size less than $(2D+2)^D$, then the insertion algorithm is completed. Otherwise, if $N(q)$ has size $(2D+2)^D$, let $r_1, \ldots, r_l$ be all points in $N(q)$ that have maximal L_∞-distance to q. For each $1 \leq i \leq l$, delete r_i from $N(q)$ and delete $d(q,r_i)$ from the heap. Finally, delete $box(q)$ from the box tree and insert the box centered at q having r_1 on its boundary as the new $box(q)$.

The deletion algorithm: Let $p \in S$ be the point to be deleted.

1. Delete p from the range tree. Search for p in the dictionary. For each point q in $N(p)$, delete the distance $d(p,q)$ from the heap. Then, delete p and $N(p)$ from the dictionary. Finally, delete $box(p)$ from the box tree.
2. Using the box tree, find all boxes that contain p. For each reported element $box(q)$, do the following:
   (a) If p lies in the interior of $box(q)$, then search in the dictionary for q, delete p from $N(q)$ and delete $d(q,p)$ from the heap.
   (b) Using the range tree, find the $1+(2D+2)^D$ L_∞-neighbors of q. Let $box_0(q)$ be the smallest box centered at q that contains these neighbors. If $box(q) = box_0(q)$, then the deletion algorithm is completed.
   (c) Otherwise, if $box(q) \neq box_0(q)$, let $r_1, \ldots, r_l$ be all points that are contained in the interior of $box_0(q)$ but that do not belong to $N(q) \cup \{q\}$. For each $1 \leq i \leq l$, insert r_i into $N(q)$, insert $d(q,r_i)$ into the heap and store with r_i a pointer to $d(q,r_i)$. Finally, delete $box(q)$ from the box tree and insert $box_0(q)$, being the new $box(q)$.

It is easy to verify that these update algorithms correctly maintain the closest pair data structure. During these algorithms, we perform a constant number of query and update operations in the range tree, the box tree, the heap and the dictionary. Therefore, by Theorems 3.6 and 5.5, the amortized update time of the entire data structure is bounded by $O((\log n)^{D-1} \log\log n)$. We have proved the following result.

THEOREM 6.2. *Let S be a set of n points in $\mathbb{R}^D$ and let $1 \leq t \leq \infty$. There exists a data structure of size $O(n(\log n)^{D-1})$ that maintains an L_t-closest pair of S in $O((\log n)^{D-1} \log\log n)$ amortized time per insertion and deletion. The constant factor in the space (resp. update time) bound is of the form $O(D)^D$ (resp. $2^{O(D^2)}$).*

Consider again our data structure. The box $box(p)$ that is associated with point p contains an L_∞-neighbor of p in $S \setminus \{p\}$. Therefore, the data structure can easily be adapted such that it maintains for each point in S its L_∞-neighbor.

COROLLARY 6.3. *Let S be a set of n points in $\mathbb{R}^D$. There exists a data structure of size $O(n(\log n)^{D-1})$ that maintains an L_∞-neighbor of each point in S. This data structure has an amortized update time of $O((\log n)^{D-1} \log\log n)$.*

7. A transformation for reducing the space complexity. The closest pair data structure of the previous section uses more than linear space. This raises the question if the same update time can be obtained using only linear space. In this section, we show that for $D \geq 3$, this is indeed possible. For $D = 2$, we will obtain a family of closest pair data structures.

Note that we have a linear space solution for maintaining the set $\{box(p) : p \in S\}$. (See Theorem 5.8.) For the L_∞-neighbor problem, however, no linear space solution having polylogarithmic query and update times is known. Hence, in order to reduce the space complexity, we should avoid using the range tree.

We will give a transformation that, given *any* dynamic closest pair data structure having more than linear size, produces another dynamic closest pair structure that uses less space.

The transformed data structure is composed on two sets A and B that partition S. The set B is contained in a dynamic data structure. To reduce space, B is a subset of the entire set and contains points involved in $o(n)$ updates only.

Let DS be a data structure that maintains a closest pair in a set of n points in $\mathbb{R}^D$ under insertions and deletions. Let $S(n)$ and $U(n)$ denote the size and update time of DS, respectively. The update time may be worst-case or amortized. We assume that $S(n)$ and $U(n)$ are non-decreasing and *smooth* in the sense that $S(\Theta(n)) = \Theta(S(n))$ and $U(\Theta(n)) = \Theta(U(n))$. Finally, let $f(n)$ be a non-decreasing smooth integer function. such that $1 \leq f(n) \leq n/2$.

Let $S \subseteq \mathbb{R}^D$ be the current set of points. The cardinality of S is denoted by n. Our transformed data structure will be completely rebuilt after a sufficiently long sequence of updates. Let S_0 be the set of points at the moment of the most recent rebuilding and let n_0 be its size at that moment.

As in the previous section, for each $p \in \mathbb{R}^D$, $box(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S \setminus \{p\}$. The set of all points of $S \setminus \{p\}$ that are in the interior of this box is denoted by $N(p)$. If $p \in S_0$, then $box_0(p)$ denotes the smallest box centered at p that contains at least $(2D+2)^D$ points of $S_0 \setminus \{p\}$.

The transformed closest pair data structure:

1. The set S is partitioned into sets A and B such that $A \subseteq \{p \in S : p \in S_0 \wedge box(p) \subseteq box_0(p)\}$.
2. The distances of the multiset $\{d(p,q) : p \in A, q \in N(p)\}$ are stored in a heap. With each distance, we store the corresponding pair of points.
3. The boxes of the set $\{box_0(p) : p \in S_0\}$ are stored in a list, called the *box list.* With each element $box_0(p)$ in this list, we store a bit having value *true* if and only if $p \in A$. Moreover, if $p \in A$, we store with $box_0(p)$ the box $box(p)$.
4. The boxes of the set $\{box_0(p) : p \in S_0\}$ are stored in the *static* data structure of Theorem 5.8. This structure is called the *box tree.* With each box in this structure, we store a pointer to its occurrence in the box list.
5. The points of S are stored in a dictionary. With each point p, we store a bit indicating whether p belongs to A or B. If $p \in A$, then we store with p
   (a) a pointer to the occurrence of $box_0(p)$ in the box list, and
   (b) a list containing the elements of $N(p)$. For convenience, we also call this list $N(p)$. With each point q in $N(p)$, we store a pointer to the occurrence of $d(p,q)$ in the heap.
6. The set B is stored in the dynamic data structure DS. This structure is called the *B-structure.*

First we prove that this data structure indeed enables us to find a closest pair of the current set S in $O(1)$ time.

LEMMA 7.1. *Let δ be the minimal distance stored in the heap and let δ' be the distance of a closest pair in B. Then, $\min(\delta, \delta')$ is the distance of a closest pair in the set S.*

Proof. Let (P, Q) be a closest pair in S. We distinguish two cases.

Case 1: At least one of P and Q is contained in A.

Assume w.l.o.g. that $P \in A$. Since $box(P)$ contains at least $1 + (2D+2)^D$ points of S, it follows in the same way as in the proof of Lemma 6.1 that Q is contained in the interior of this box. Hence, $Q \in N(P)$ and, therefore, the distance $d(P,Q)$ is stored in the heap. Clearly, the heap only contains distances of the current set S. Therefore, $\delta = d(P,Q)$. Moreover, since $d(P,Q)$ is the minimal distance in S, we have $d(P,Q) \leq \delta'$. This proves that $d(P,Q) = \min(\delta, \delta')$.

Case 2: Both P and Q are contained in B.

Since $d(P,Q)$ is the minimal distance in S and since the heap only stores distances between points of the current set S, we have $\delta' = d(P,Q)$ and $d(P,Q) \leq \delta$. Therefore, $d(P,Q) = \min(\delta, \delta')$. □

Initialization: At the moment of initialization, $S = S_0 = A$ and $B = \emptyset$. Using Vaidya's algorithm [24], compute for each point p in S its $1 + (2D+2)^D$ L_∞-neighbors. The point among these neighbors having maximal L_∞-distance to p determines $box(p) = box_0(p)$. The neighbors (except p itself) that are contained in the interior of this box form the list $N(p)$. It is clear how the rest of the data structure can be built. Note that each element $box_0(p)$ in the box list has a bit with value *true*.

Now we can give the update algorithms. If a point p of S_0 is deleted, it may be inserted again during some later update operation. If this happens, p is assumed to be a new point, i.e., it is assumed that p does not belong to S_0 again. In this way, an inserted point always belongs to B.

The insertion algorithm: Let $p \in \mathbb{R}^D$ be the point to be inserted. Assume w.l.o.g. that $p \notin S$.

1. Insert p into the dictionary and store with p a bit saying that p belongs to B. Then, insert p into the B-structure.
2. Using the box tree, find all boxes that contain p. For each reported element $box_0(q)$, follow the pointer to its occurrence in the box list. If the bit of $box_0(q)$ has value *true*, then check if p is contained in the interior of $box(q)$. If so, do the following:
   (a) Search in the dictionary for q. Insert p into $N(q)$, insert $d(q,p)$ into the heap and store with p a pointer to $d(q,p)$.
   (b) If $N(q)$ has size less than $(2D+2)^D$, then the insertion algorithm is completed. Otherwise, let $r_1, \ldots, r_l$ be all points of $N(q)$ that are at maximal L_∞-distance from q. For each $1 \leq i \leq l$, delete r_i from $N(q)$ and delete $d(q, r_i)$ from the heap. Finally, replace $box(q)$—which is stored with $box_0(q)$ in the box list—by the box centered at q having r_1 on its boundary, being the new $box(q)$.

It is easy to verify that this algorithm correctly maintains the data structure. Note that since $A \subseteq \{p \in S : p \in S_0 \wedge box(p) \subseteq box_0(p)\}$, all boxes $box(q)$, $q \in A$, that contain p are found in Step 2.

The deletion algorithm: Let $p \in S$ be the point to be deleted.

1. Search for p in the dictionary. If $p \in B$, then delete p from this dictionary and from the B-structure.

Otherwise, if $p \in A$, follow the pointer to $box_0(p)$ in the box list and set its bit to *false*. Moreover, for each point q in $N(p)$, delete the distance $d(p, q)$ from the heap. Then, delete p from the dictionary.

2. Using the box tree, find all boxes that contain p. For each reported element $box_0(q)$, follow the pointer to its occurrence in the box list. If the bit of $box_0(q)$ has value *true*, then check if p is contained in $box(q)$. If so, do the following:
Set the bit of $box_0(q)$ to *false*. Search in the dictionary for q. For each point r in $N(q)$, delete $d(q, r)$ from the heap. Then, delete the pointer from q to $box_0(q)$, delete the list $N(q)$, and store with q a bit saying that it belongs to B. Finally, insert q into the B-structure.

This concludes the description of the update algorithms. In order to guarantee a good space bound, we occasionally rebuild the data structure:

Rebuild: Recall that n_0 is the size of S at the moment we initialize the structure. After $f(n_0)$ updates have been performed, we discard the entire structure and initialize a new data structure for the current S.

We analyze the complexity of the transformed data structure. First note that the initial and current sizes n_0 and n are proportional: Since $f(n_0) \leq n_0/2$, we have $n \leq n_0 + f(n_0) \leq 3n_0/2$ and $n \geq n_0 - f(n_0) \geq n_0/2$.

The total size of the heap, the box list, the box tree and the dictionary is bounded by $O(n + n_0) = O(n)$. Consider the B-structure. Initially, this structure is empty. With each insertion, we insert one point into it, whereas with each deletion at most a constant number of points are inserted. (See Lemma 5.1 for the constant. Note that $c_D = (2D+2)^D$.) Therefore, the B-structure stores $O(f(n_0)) = O(f(n))$ points and it has size $O(S(f(n)))$.

During each update operation, we perform a constant number of queries and updates in the various parts of the structure. Therefore, by Theorem 5.8, $O((\log n)^{D-1} + U(f(n)))$ time is spent per update, in case the data structure is not rebuilt.

Consider the initialization. We can compute for each point its $1 + (2D+2)^D$ L_∞-neighbors in $O(n_0 \log n_0)$ time. (See [24].) By Theorem 5.8, the static box tree can also be built in $O(n_0 \log n_0)$ time. It is clear that the rest of the data structure can be built within these time bounds. Hence, the entire initialization takes $O(n_0 \log n_0)$ time. Since we do not rebuild during the next $f(n_0)$ updates, the initialization adds

$$O\left(\frac{n_0 \log n_0}{f(n_0)}\right) = O((n \log n)/f(n))$$

to the overall amortized update time. This proves:

THEOREM 7.2. *Let DS be any data structure for the dynamic closest pair problem. Let $S(n)$ and $U(n)$ denote the size and update time of DS, respectively. Let $1 \leq f(n) \leq n/2$ be a non-decreasing integer function. Assume that $S(n)$, $U(n)$ and $f(n)$ are smooth.*

We can transform DS into another data structure for the dynamic closest pair problem having

1. *size $O(n + S(f(n)))$, and*
2. *an amortized update time of $O((\log n)^{D-1} + U(f(n)) + (n \log n)/f(n))$.*

COROLLARY 7.3. *Let S be a set of n points in $\mathbb{R}^D$, $D \geq 3$, and let $1 \leq t \leq \infty$. There exists a data structure of size $O(n)$ that maintains an L_t-closest pair of S in $O((\log n)^{D-1} \log\log n)$ amortized time per insertion and deletion.*

Proof. We apply Theorem 7.2 twice. Let DS be the data structure of Theorem 6.2, i.e., $S(n) = O(n(\log n)^{D-1})$ and $U(n) = O((\log n)^{D-1} \log\log n)$. Moreover, let $f(n) =$

$n/((\log n)^{D-2} \log\log n)$. Then, Theorem 7.2 gives a closest pair structure DS' of size $S'(n) = O(n \log n / \log\log n)$ having $U'(n) = O((\log n)^{D-1} \log\log n)$ amortized update time. Applying Theorem 7.2 to DS' with $f'(n) = n \log\log n / \log n$ proves the corollary. □

COROLLARY 7.4. *Let S be a set of n points in the plane and let $1 \leq t \leq \infty$. For any non-negative integer constant k, there exists a data structure*

1. *of size $O(n \log n / (\log\log n)^k)$ that maintains an L_t-closest pair of S at a cost of $O(\log n \log\log n)$ amortized time per insertion and deletion,*
2. *of size $O(n)$ that maintains an L_t-closest pair of S in $O((\log n)^2 / (\log\log n)^k)$ amortized time per insertion and deletion.*

Proof. The proof is by induction on k. For $k = 0$, the result claimed in 1. follows from Theorem 6.2. Let $k \geq 0$ and let DS be a closest pair data structure having size $S(n) = O(n \log n / (\log\log n)^k)$ and update time $U(n) = O(\log n \log\log n)$. Applying Theorem 7.2 with $f(n) = n(\log\log n)^k / \log n$ gives a closest pair structure of size $O(n)$ having $O((\log n)^2 / (\log\log n)^k)$ amortized update time.

On the other hand, applying Theorem 7.2 to DS with $f(n) = n / \log\log n$ gives a closest pair structure of size $O(n \log n / (\log\log n)^{k+1})$ with an amortized update time of $O(\log n \log\log n)$. □

8. Concluding remarks. We have given new techniques for solving the dynamic approximate nearest-neighbor problem and the dynamic closest pair problem. Note that for the static version of the first problem an optimal solution—having logarithmic query time and having linear size—is known. (See [2].) It would be interesting to solve the dynamic problem within the same complexity bounds and with a logarithmic update time.

For the dynamic closest pair problem, we obtained several new results. We first gave a data structure that improved the best structures that were known. Then we applied a general transformation to improve this solution even further. Note that if we apply this transformation several times, as we did, then we maintain a hierarchy of data structures similar to the logarithmic method for decomposable searching problems. (See [3].) It would be interesting to know if the ideas of this transformation can be applied to other problems. Finally, we leave as an open problem to decide whether there is an $O(n)$ space data structure that maintains the closest pair in $O(\log n)$ time per insertion and deletion.

REFERENCES

[1] S. ARYA AND D. M. MOUNT, *Approximate nearest neighbor queries in fixed dimensions*, in Proc. of 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 271–280.
[2] S. ARYA, D. M. MOUNT, N. S. NETANYAHU, R. SILVERMAN, AND A. WU, *An optimal algorithm for approximate nearest neighbor searching*, in Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 573–582.
[3] J. L. BENTLEY, *Decomposable searching problems*, Inform. Proc. Lett., 8 (1979), pp. 244–251.
[4] J. L. BENTLEY AND M. I. SHAMOS, *Divide-and-conquer in multidimensional space*, in Proc. of 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 220–230.
[5] M. BERN, *Approximate closest-point queries in high dimensions*, Inform. Proc. Lett., 45 (1993), pp. 95–99.
[6] B. CHAZELLE AND L. J. GUIBAS, *Fractional cascading I: A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.
[7] K. L. CLARKSON, *A randomized algorithm for closest-point queries*, SIAM J. Comput., 17 (1988), pp. 830–847.
[8] H. EDELSBRUNNER, G. HARING, AND D. HILBERT, *Rectangular point location in d dimensions with applications*, The Computer Journal, 29 (1986), pp. 76–82.

[9] M. Golin, R. Raman, C. Schwarz, and M. Smid, *Randomized data structures for the dynamic closest-pair problem*, in Proc. of 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 301–310.

[10] ———, *Simple randomized algorithms for closest pair problems*, in Proc. of 5th Canadian Conference on Computational Geometry, 1993, pp. 246–251.

[11] D. T. Lee, *Two-dimensional Voronoi diagrams in the L_p-metric*, J. ACM, 27 (1980), pp. 604–618.

[12] G. S. Lueker, *A data structure for orthogonal range queries*, in Proc. of 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 28–34.

[13] K. Mehlhorn, *Data Structures and Algorithms, Volume 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[14] K. Mehlhorn and S. Näher, *Dynamic fractional cascading*, Algorithmica, 5 (1990), pp. 215–241.

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

[16] J. S. Salowe, *Enumerating interdistances in space*, International Journal of Computational Geometry & Applications, 2 (1992), pp. 49–59.

[17] C. Schwarz, *Data structures and algorithms for the dynamic closest pair problem*, Ph.D. thesis, Department of Computer Science, Universität des Saarlandes, Saarbrücken, 1993.

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

[19] M. I. Shamos and D. Hoey, *Closest-point problems*, in Proc. of 16th Annual IEEE Symposium on Foundations of Computer Science, 1975, pp. 151–162.

[20] M. Smid, *Maintaining the minimal distance of a point set in less than linear time*, Algorithms Review, 2 (1991), pp. 33–44.

[21] ———, *Dynamic rectangular point location, with an application to the closest pair problem*, Information and Computation, to appear.

[22] ———, *Maintaining the minimal distance of a point set in polylogarithmic time*, Discrete Comput. Geom., 7 (1992), pp. 415–431.

[23] K. J. Supowit, *New techniques for some dynamic closest-point and farthest-point problems*, in Proc. of 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 84–90.

[24] P. M. Vaidya, *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem*, Discrete Comput. Geom., 4 (1989), pp. 101–115.

[25] D. E. Willard and G. S. Lueker, *Adding range restriction capability to dynamic data structures*, J. ACM, 32 (1985), pp. 597–617.

[26] A. C. Yao, *On constructing minimum spanning trees in k-dimensional spaces and related problems*, SIAM J. Comput., 11 (1982), pp. 721–736.