# A Self-adjusting Data Structure for Multidimensional Point Sets* 

Eunhui Park and David M. Mount<br>University of Maryland, College Park MD 20742, USA<br>\{ehpark, mount\}@cs.umd.edu


#### Abstract

A data structure is said to be self-adjusting if it dynamically reorganizes itself to adapt to the pattern of accesses. Efficiency is typically measured in terms of amortized complexity, that is, the average running time of an access over an arbitrary sequence of accesses. The best known example of such a data structure is Sleator and Tarjan's splay tree. In this paper, we introduce a self-adjusting data structure for storing multidimensional point data. The data structure is based on a quadtree-like subdivision of space. Like a quadtree, the data structure implicitly encodes a subdivision of space into cells of constant combinatorial complexity. Each cell is either a quadtree box or the set-theoretic difference of two such boxes. Similar to the traditional splay tree, accesses are based on an splaying operation that restructures the tree in order to bring an arbitrary internal node to the root of the tree. We show that many of the properties enjoyed by traditional splay trees can be generalized to this multidimensional version.


Keywords: Geometric data structures, quadtrees, self-adjusting data structures, amortized analysis.

## 1 Introduction

The development of efficient data structures for geometric search and retrieval is of fundamental interest in computational geometry. The continued interest in geometric data structures is motivated by the tension between the various desirable properties such a data structure should possess, including efficiency (in space and query time), generality, simplicity, efficient handling of updates, and accuracy (when approximation is involved). Partition trees, including the quadtree and its many variants, are among the most popular geometric data structures. The quadtree defines a hierarchical subdivision of space into hypercube-shaped cells [24]. The simplicity and general utility of quadtree-based data structures is evident in the wide variety of problems to which they have been applied, both in theory and practice.

Our focus here stems from the study of dynamic data structures for storing multidimensional point sets. Through path compression (see e.g., [5, 9,16 ), compressed quadtrees can store an arbitrary set of $n$ points in $\mathbb{R}^{d}$ in linear space.

[^0](Throughout, we assume that $d$ is a constant.) Since a compressed quadtree may have height $\Theta(n)$, it is desirable to consider ways of maintaining balance within the tree. Examples of such approaches include the randomized skip quadtree of Eppstein et al. 14 and our own quadtreap data structure, which, respectively, generalize the skip list data structure [23] and the treap data structure [25] to a multidimensional setting. It also includes Chan's data structures [7, 8], which are based on ordering the points according to the Morton ordering. These data structures all support efficient insertion and deletion and can answer approximate proximity queries, such as approximate nearest neighbor searching and approximate range searching.

The above data structures guarantee $O(\log n)$ time (possibly randomized) to access each node of the tree, and so are efficient in the worst case. When access patterns may be highly skewed, however, it is of interest to know whether the data structure can achieve even higher degrees of efficiency by exploiting the fact that some nodes are much more likely to be accessed than others. A natural standard for efficiency in such cases is based on the entropy of the access distribution. Entropy-optimal data structures for point location have been proposed by Arya et al. [1, 2] and Iacono [19]. These data structures are static, however, and cannot readily be generalized to other types of search problems.

A commonly studied approach for adaptive efficiency for one-dimensional data sets involves the notion of a self-adjusting data structure, that is, a data structure that dynamically reorganizes itself to fit the pattern of data accesses. Although many self-adjusting data structures have been proposed, the best-known example is the splay tree of Sleator and Tarjan [26, 29]. This is a binary search tree that stores no internal balance information, but nonetheless provides efficient dynamic dictionary operations (search, insert, delete) with respect to amortized time, that is, the average time per operation over a sequence of operations. Sleator and Tarjan demonstrated that (or in some cases conjectured that) the splay tree adapts well to skewed access patterns. For example, rather than relating access time to the logarithm of the number of points in the set, it is related to a parameter that characterizes the degree of locality in a query. Examples of measures of locality include the working set bound [26, static and dynamic finger bounds [10, 11, 26], the unified bound [6, 17, 26], key-independent optimality [20], and the BST model [12]. Numerous papers have been devoted to the study of dynamic efficiency of data structures (see, e.g., [13, 15, 22, 28, 30]).

Given the fundamental importance of the splay-tree to the storage of 1 dimensional point sets and the numerous works it has inspired, it is remarkable that, after decades of study, there are no comparable self-adjusting data structures for storing multidimensional point sets. In this paper we propose such a data structure, called the splay quadtree. Like the splay tree, the splay quadtree achieves adaptability through a restructuring operation that moves an arbitrary internal node to the root of the tree. (In our tree, data is stored exclusively at the leaf nodes, and so it is not meaningful to bring a leaf node to the root.) Each node of our tree is associated with a region of space, called a cell, which is either a quadtree box or the set-theoretic difference of two quadtree boxes. As seen in
other variants of the quadtree, such as the BBD tree [3, 4] and quadtreap 21], this generalization makes it possible to store an arbitrary set of $n$ points in a tree of linear size and logarithmic height.

Although there are complexities present in the multidimensional context that do not arise in the 1-dimensional context, we show how to define a generalization of the splaying operation of [26] to the splay quadtree. We show that the fundamental properties of the standard splay tree apply to our data structure as well. In particular, we show that the amortized time of insertions and deletions into a tree of size $n$ is $O(\log n)$. We also present multidimensional generalizations of many of the results of [26, including the Balance Theorem, the Static Optimality Theorem, the Working Set theorem, and variants of the Static Finger Theorem for a number of different proximity queries, including box queries (which generalize point-location queries), approximate nearest neighbor queries, and approximate range counting queries.

The rest of the paper is organized as follows. In Section 2, we present some background material on the BD tree, the variant of the quadtree on which the splay quadtree is based. In Section 3 we present the splaying operation and analyze its amortized time. In Section 4, we present and analyze algorithms for point insertion and deletion. Finally, in Section 5, we present the various search and optimality results. Due to space limitations, a number of proofs and figures have been omitted. They will appear in the full version of the paper.

## 2 Preliminaries

We begin by recalling some of the basic elements of quadtrees and BD-trees. A quadtree is a hierarchical decomposition of space into $d$-dimensional hypercubes, called cells. The root of the quadtree is associated with a unit hypercube $[0,1]^{d}$, and we assume that (through an appropriate scaling) all the points fit within this hypercube. Each internal node has $2^{d}$ children corresponding to a subdivision of its cell into $2^{d}$ disjoint hypercubes, each having half the side length.

It will be convenient to consider a binary variant of the quadtree. Each decomposition step splits a cell by an axis-orthogonal hyperplane that bisects the cell's longest side. If there are ties, the side with the smallest coordinate index is selected. Although cells are not hypercubes, each cell has constant aspect ratio. Henceforth, we use the term quadtree box in this binary context.

If the point distribution is highly skewed, a compressed quadtree may have height $\Theta(n)$. One way to deal with this is to introduce a partitioning mechanism that allows the algorithm to "zoom" into regions of dense concentration. In [3, 4] a subdivision operation, called shrinking, was proposed to achieve this. The resulting data structure is called a box-decomposition tree (BD-tree). This is a binary tree in which the cell associated with each node is either a quadtree box or the set theoretic difference of two such boxes, one enclosed within the other. Thus, each cell is defined by an outer box and an optional inner box. Although cells are not convex, they have bounded aspect ratio and are of constant combinatorial complexity.

The spay quadtree is based on a decomposition method that combines shrinking and splitting at each step. We say that a cell of a BD-tree is crowded if it contains two or more points or if it contains an inner box and at
 least one point. Let $B$ denote this cell's outer box, and let $C$ be any quadtree box contained within $B$, such that, if the cell has an inner box, $C$ contains it. We generate a cell $O=B \backslash C$. We then split $C$ by bisecting its longest side forming two cells $L$ and $R$. By basic properties of quadtree boxes, the inner box (if it exists) lies entirely inside of one of the two cells. We make the convention, called the inner-left convention, that the inner box (if it exists) lies within $L$. We generate a node with three children called the left, right, and outer child, corresponding to the cells $L$, $R$, and $O$, respectively. We allow for the possibility that $C=B$, which implies that the outer child's cell is empty, which we call a vacuous leaf. Our algorithms will maintain the invariant that the left and right children separate at least two entities (either two points or a point and inner box). It follows from the analysis given in [21] that the size of the tree is linear in the number of points.

As shown in [21, the tree can be restructured through a local operation, called promotion, which is analogous to rotation in binary trees. The operation is given any non-root internal node $x$ and is denoted by $\operatorname{promote}(x)$. Let $y$ denote $x$ 's parent. There are three cases depending on which child $x$ is (see Fig. (1). Note that promotion does not alter the underlying subdivision of space.


Fig. 1. (a) Left and outer promotions (b) right promotion

Left Child: This makes $y$ the outer child of $x$, and $x$ 's old outer child becomes the new left child of $y$ (see Fig. [1(a)).
Outer Child: This is the inverse of the left-child promotion, where the roles of $x$ and $y$ are reversed (see Fig. [(a)).

Right Child: To maintain the convention that a cell has only one inner box, this operation is defined only if $x$ 's left sibling node does not have an inner box (node $D$ in Fig. (b)). Letting $y$ denote $x$ 's parent, promote( $(x)$ swaps $x$ with its left sibling and then performs $\operatorname{promote}(x)$ (which is now a left-child promotion).

It is easily verified that promotion can be performed in $O(1)$ time and preserves the inner-left convention. We say that a promotion is admissible, if it is either a left or outer promotion or if it is a right promotion under the condition that the left child has no inner box.

## 3 Self-adjusting Quadtrees

Like the standard splay tree [26], our self-adjusting quadtree, or splay quadtree, maintains no balance information and is based on an operation, called splaying, that moves an arbitrary internal node to the root through a series of promotions. This operation is described in terms of primitives, called basic splaying steps.

### 3.1 Basic Splaying Steps

Let us begin with some definitions. Let $x$ be internal, non-root node, and let $p(x)$ denote its parent. If $p(x)$ is not the root, let $g(x)$ denote $x$ 's grandparent. Define $\operatorname{rel}(p(x), x)$ to be one of the symbols $L, R$, or $O$ depending on whether $x$ is the left, right, or outer child of $p(x)$, respectively. If $p(x)$ is not the root, we also define $\operatorname{str}(x)$ to be a two-character string over the alphabet $\{L, R, O\}$ whose first character is $\operatorname{rel}(g(x), p(x))$ and whose second character is $\operatorname{rel}(p(x), x)$. The basic splaying step is defined as follows.

## Basic Splay $(x)$ :

$z i \boldsymbol{g}$ : If $p(x)$ is the root, perform $\operatorname{promote}(x)$ (see Fig. 2(a)).
zig-zag: If $p(x)$ is not the root, and $\operatorname{str}(x) \in\{L O, R O, O L\}$, do promote $(x)$, and then $\operatorname{promote}(x)$ (see Fig. 2(b) for the case $\operatorname{str}(x)=L O)$.
zig-zig: Otherwise, perform $\operatorname{promote}(p(x))$ and then perform promote $(x)$ (see Fig. 2(c) for the case $\operatorname{str}(x)=L L$ ).

Note that basic splaying may not be applicable to a node because it would result in an inadmissible promotion. We say that a basic splaying step is admissible if all of its promotions are admissible. We will show later that our splaying algorithm performs only admissible promotions.

### 3.2 Splaying Operation

As mentioned earlier, the splay quadtree is based on an operation that modifies the tree so that a given internal node becomes the root. We start with a definition. Let $T$ be a splay quadtree, let $u$ be a node of $T$, and let $t$ be $T$ 's root.


Fig. 2. The basic splaying operations: (a) zig, (b) zig-zag, and (c) zig-zig, where (before promotion) $y=p(x)$ and $z=g(x)$

Letting $\left\langle u=u_{1}, u_{2}, \ldots, u_{n}=t\right\rangle$ denote the path from $u$ to $t$, the path sequence $s e q_{T}(u)$ is defined

$$
\operatorname{seq}_{T}(u)=\operatorname{rel}\left(u_{n}, u_{n-1}\right) \ldots \operatorname{rel}\left(u_{3}, u_{2}\right) \operatorname{rel}\left(u_{2}, u_{1}\right) .
$$

Let $\varepsilon$ denote the empty string. Using the terminology of regular expressions, this sequence can be thought of as a string in the language $\{L, R, O\}^{*}$. Thus, after splaying $u$, we have $\operatorname{seq}_{T}(u)=\varepsilon$.
In the standard binary splay tree [26], rotations are performed bottom-up along the access path. In our case, we will need to take care to avoid inadmissible promotions. We perform the splaying operation in three phases, each of which transforms the path sequence into successively more restricted form. Due to space limitations, the details of the algorithm and its analysis are presented in the full version. Here is a high-level overview of the process.

Phase 1: Initially, the path sequence is arbitrary, that is, $\operatorname{seq}_{T}(u) \in\{L, R, O\}^{*}$, which can be represented by $\left\{\{O, L\}^{*} R\right\}^{*}\{O, L\}^{*}$. This phase compresses each substring of the form $\{O, L\}^{*}$ to a substring consisting of at most one such character by an appropriate sequence of zig-zig or zig-zag steps. The output has the form $\{\{O, L, \varepsilon\} R\}^{*}\{O, L, \varepsilon\}$.

Phase 2: In this phase, we transform $s e q_{T}(u)$ such that the sibling node of any right child on the path does not have an inner box. After Phase 1, we have $\operatorname{seq}_{T}(u) \in\{\{O, L, \varepsilon\} R\}^{*}\{O, L, \varepsilon\}$, which can be equivalently represented as $\{O R, L R, R\}^{*}\{O, L, \varepsilon\}$. Here, we consider the case where the sequence starts with $O R$, that is, $\operatorname{str}(x)=O R$. Thus, since $x$ 's parent, $p(x)$ is an outer child, and the left child of $p(x)$, which is a sibling node of $x$, has an inner box, we cannot perform a right-child promotion at $x$. To solve this, we transform $s e q_{T}(u)$ to $\{L R, R\}^{*}\left\{O L^{*}, O, L, \varepsilon\right\}$ by removing the $O R$ substrings through zig-zig steps.

Fig. 3 shows an example of a zig-zig step for such an $O R$ substring. Let $y$ denote $x$ 's parent. First, we perform $\operatorname{promote}(y)$. Before promote (y), $x$ 's left sibling (labeled $w_{1}$ in Fig. (3) may have an inner box. But, after promote(y), $x$ 's new sibling node, $z$, does not have an inner box. Thus, now promote $(x)$ is admissible. We perform $\operatorname{promote}(x)$. By the right-child property, prior to this $z i g$-zig step, $z$ 's children, $A$ and $B$, and $x$ 's children, $E$ and $F$ do not have inner boxes. After the operation, the right-child property still holds since $x$ 's children, $E$ and $F$, and $z$ 's children, $A$ and $B$, do not have inner boxes. The inner-left convention is still preserved, since prior to the operation, the inner box consisting of the union of $A$ and $B$ 's cells in Fig 3 lies in $y$ 's left child (labeled $w_{1}$ in Fig. (3), and after the operation, the inner box consisting of the union of $E$ and $F$ 's cells lies in $y$ 's left child (labeled $w_{2}$ in Fig. (3).


Fig. 3. A zig-zig splaying for an $O R$ substring

Note that $x$ 's old outer child (labeled $w_{2}$ in Fig. (3) becomes to the left child of $x$ 's new outer child, $y$ after the zig-zig step. Thus, if $\operatorname{str}(x)$ is $O R$, and $u$ is $x$ 's outer child, the value of $\operatorname{str}(u)$ after zig-zig step is $O L$. This occurs recursively. Therefore, $\operatorname{seq}(u)=\{O R, L R, R\}^{*}\{O, L, \varepsilon\}$ is transformed to a sequence of the form $\{L R, R\}^{*}\left\{O L^{*}, O, L, \varepsilon\right\}$ by removing $O R$.

Phase 3: After Phase 2, the path sequence has been modified so that all rightchild transitions precede outer-child transitions. Since inner boxes are generated only by outer-child transitions, it follows that promotions performed on any right-child nodes of the current path are admissible. This phase completes the process (thus brining $u$ to the root) by a series of basic splaying steps (zig-zag if $\operatorname{str}(x) \in\{R O, O L\}$ and zig-zag otherwise).

Let us now analyze the amortized time of splaying. As in [26, we will employ a potential-based argument. We assign each node $x$ a non-negative weight $w(x)$. We define $x$ 's size, $s(x)$, to be the sum of the weights of its descendants, including $x$ itself, and we define its rank, $r(x)$, to be $\log s(x)$. (Unless otherwise specified, all logs are taken base 2.) We also define the potential $\Phi$ of the tree to be the sum of the ranks of all its nodes. The amortized time, $a$, of an operation is defined to be $t+\Phi^{\prime}-\Phi$, where $t$ is the actual time of the operation, $\Phi$ is the potential before the operation, and $\Phi^{\prime}$ is the potential after the operation. It is easily verified that our splay operations satisfy the general splay properties described by Subramanian [27], and therefore, by the analysis presented there we have:

Lemma 1. The amortized time to splay a tree with root $t$ at a node $u$ is at most $O(r(t)-r(u))=O(\log s(t) / s(u))$.

## 4 Updates to Splay Quadtrees

The insertion of a point $q$ into a splay quadtree can be performed by a straightforward modification of the incremental insertion process presented in 21]. This algorithm first determines the leaf node $x$ containing $q$ by a simple descent of the tree. It then generates a new node $u$ to separate $q$ from $x$ 's current contents and replaces $x$ with $u$. The leaf containing $q$ becomes the right child of $u$. After insertion, we splay $u$.

The deletion process is complimentary but more complex. We first find the leaf node containing $q$. Let $u$ be this node's parent. As in [21], we apply a restructuring procedure to the subtree rooted at $u$ so that the tree structure would be the same as if the point $q$ had been the last point to be inserted. After restructuring, we delete $q$ by simply "undoing" the insertion process. We then splay $q$ 's parent node. The analysis is similar to that of [26]. Details will appear in the full version.
Lemma 2. Given a splay quadtree $T$ containing an n-element points set, insertion or deletion of a point can be performed in $O(\log n)$ amortized time.

## 5 Search and Optimality Results

In this section, we present theorems related to the complexity of performing various types of geometric searches on splay quadtrees. We provide generalizations to a number of optimality results for the standard splay tree, including the Balance Theorem, the Static Optimality Theorem, the Working Set theorem, and variants of the Static Finger Theorem. As in [26], our approach is based on applying Lemma 1 under an appropriate assignment of weights to the nodes.

We first discuss theorems related to the search time for accessing points. These theorems are essentially the same as those proved in [26] for 1-dimensional splay trees. Given a query point $q$, an access query returns the leaf node of $T$ whose cell contains $q$. (Thus, it can be thought of as a point location query for the subdivision of space induced by $T$.) An access query is processed by traversing the path from the root to the leaf node containing $q$, and splaying the parent of this leaf.

The working set property states that once an item is accessed, accesses to the same item in the near future are particularly efficient [26]. Through the application of an appropriate weight assignment, similar to one used in [26] we obtain the following.
Theorem 1. (Working Set Theorem) Consider a set $P$ of $n$ points in $\mathbb{R}^{d}$. Let $q_{1}, \ldots, q_{m}$ be a sequence of access queries. For each access $q_{j}(1 \leq j \leq m)$, let $t_{j}$ be the number of different cells accessed before $q_{j}$ since its previous access, or since the beginning of the sequence if this is $q_{j}$ 's first access. The total time to answer $m$ queries is $O\left(\sum_{j=1}^{m} \log \left(t_{j}+1\right)+m+n \log n\right)$.

Two additional properties follow as consequences of the above theorem (see, e.g., Iacono [18]):

Theorem 2. (Balance Theorem) Consider a set $P$ of $n$ points in $\mathbb{R}^{d}$ and a splay quadtree $T$ for $P$. Let $q_{1}, \ldots, q_{m}$ be a sequence of access queries. The total time to answer these queries is $O((m+n) \log n+m)$.

Theorem 3. (Static Optimality Theorem) Given a subdivision $Z$, and the empirical probability $p_{z}$ for each cell $z \in Z$, the total time to answer $m$ access queries is $O(m \cdot \operatorname{entropy}(Z))$.

In traditional splay trees, the static finger theorem states that the running times of a series of accesses can be bounded by the sum of logarithms of the distances of each accessed item to a given fixed key, called a finger. In the 1-dimensional context, the notion of distance is based on the number of keys that lie between two items [26]. Intuitively, this provides a measure of the degree of locality among a set of queries with respect to a static key. Generalizing this to a geometric setting is complicated by the issue of how to define an appropriate notion of distance based on betweenness. In this section, we present a number of static finger theorems for different queries in the splay quadtree.

In general, our notion of distance to a static finger is based on the number of relevant objects (points or quadtree cells) that are closer to the finger than the accessed object(s). Define the distance from a point $p$ to a geometric object $Q$, denoted $\operatorname{dist}(p, Q)$, to be the minimum distance from $p$ to any point of $Q$. Let $b(f, r)$ denote a ball of radius $r$ centered at a point $f$. Our first static-finger result involves access queries.

Theorem 4. Consider a set $P$ of $n$ points in $\mathbb{R}^{d}$, for some constant d. Let $q_{1}, \ldots, q_{m}$ be a sequence of access queries, where each $q_{j}(1 \leq j \leq m)$ is a point of $P$. If $f$ is any fixed point in $\mathbb{R}^{d}$, the total time to access points $q_{1}, \ldots, q_{m}$ is $O\left(\sum_{j=1}^{m} \log N_{f}\left(q_{j}\right)+m+n \log n\right)$, where $N_{f}\left(q_{j}\right)$ is the number of points of $P$ that are closer to $f$ than $q_{j}$ is.

This theorem extends the static finger theorem for the traditional splay trees of [26] to a multidimensional setting. However, it is limited to the access of points that are in the given set. In a geometric setting, it is useful to extend this to points lying outside the set. The simplest generalization is to point location queries in the quadtree subdivision, that is, determining the leaf cell that contains a given query point. More generally, given a quadtree box $Q$, a box query returns the smallest outer box of any node of a splay quadtree that contains $Q$. (Point location queries arise as a special case corresponding to an infinitely small box.)

Our next result establishes a static finger theorem to box queries (and so applies to point location queries as well). Given a query box $Q$, starting from the root of the tree, we descend to the child node that contains $Q$ until no such child exists. We then return the outer box of the current node, and if the current node is an internal node, we splay it, otherwise, we splay its parent node.

The following theorem establishes the static finger theorem for box queries.

Theorem 5. Consider a set $P$ of $n$ points in $\mathbb{R}^{d}$ and a splay quadtree $T$ for $P$. Let $Q_{1}, \ldots, Q_{m}$ be a sequence of box queries. If $f$ is any fixed point in $\mathbb{R}^{d}$, the total time to answer these queries is $O\left(\sum_{j=1}^{m} \log N_{f}\left(Q_{j}\right)+m+n \log n\right)$, where $N_{f}\left(Q_{j}\right)$ is the number of leaf cells overlapping the ball $b\left(f, \operatorname{dist}\left(f, Q_{j}\right)\right)$ in the subdivision induced by $T$.

Theorem 5 expresses the time for a series of box queries based on the number of leaf cells. In some applications, the induced subdivision is a merely a byproduct of a construction involving a set of $P$ of points. In such case, it is more natural to express the time in terms of points, not quadtree cells. Given a set of box queries that are local to some given finger point $f$, we wish to express total access time as the function of the number of points of $P$ in an appropriate neighborhood of $f$. As before, our analysis is based on quadtree leaf cells that are closer to $f$ than the query box. The problem is that such leaf cells may generally contain points are very far from $f$, relative to the query box. To handle this, we allow a constant factor expansion in the definition of local neighborhood. Consider a ball containing the closest points of all given boxes from $f$ (see Fig. [4(a)). Given a point set $P$, a finger point $f$, any constant $c$, and the box queries $Q_{1}, \ldots, Q_{m}$, the working set $W_{\text {box }}$ with respect to $P, f, c$, and the box queries is defined:
$W_{\text {box }}\left(P, f, c, Q_{1}, \ldots, Q_{m}\right)=\left\{p: p \in b\left(f, r_{\text {box }}(1+c)\right) \cap P, r_{\text {box }}=\max _{j} \operatorname{dist}\left(f, Q_{j}\right)\right\}$.


Fig. 4. (a) Working set, (b) the leaf cells overlapping a ball

Theorem 6. Let $Q_{1}, \ldots, Q_{m}$ be the sequence of box queries. For any point $f \in \mathbb{R}^{d}$ and any positive constant $c$, the total time to answer these queries is $O\left(m \log \left(\left|W_{\text {box }}\right|+\left(\frac{1}{c}\right)^{d-1}\right)+n \log n\right)$.

We will also analyze the time for answering approximate nearest neighbor. First, we consider approximate nearest neighbor queries using splay quadtrees. Let $q$ be the query point, and let $\varepsilon>0$ be the approximating factor. We apply a variant of the algorithm of [4]. We use a priority queue $U$ and maintain the closest point
$p$ to $q$. Initially, we insert the root of the splay quadtree into $U$, and set $p$ to a point infinitely far away. Then we repeatedly carry out the following process. First, we extract the node $v$ with the highest priority from the queue, that is, the node closest to the query point. If the distance from $v$ 's cell to $q$ exceeds $\operatorname{dist}(q, p) /(1+\varepsilon)$, we stop and return $p$. Since no subsequent point in any cell to be encountered can be closer to $q$ than $\operatorname{dist}(q, p) /(1+\varepsilon), p$ is an $\varepsilon$-approximate nearest neighbor. Otherwise, we descend $v$ 's subtree to visit the leaf node closest to the query point. As we descend the path to this leaf, for each node $u$ that is visited, we compute the distance to the cell associated with $u$ 's siblings and then insert these siblings into $U$. If the visited leaf node contains a point, and the distance from $q$ to this point is closer than $p$, we update $p$ to this point. Finally, we splay the parent of this leaf node.

Consider any sequence of $m$ approximate nearest neighbor queries, $q_{1}, \ldots, q_{m}$. The working set for these queries can be defined to be

$$
\begin{aligned}
& W_{\mathrm{ann}}\left(P, f, c, q_{1}, \ldots, q_{m}\right) \\
& \quad=\left\{p: p \in b\left(f, r_{\mathrm{ann}}(1+c)\right) \cap P, r_{\mathrm{ann}}=\max _{j}\left(\operatorname{dist}\left(f, q_{j}\right)+\operatorname{dist}\left(q_{j}, \mathrm{NN}\left(q_{j}\right)\right)\right\} .\right.
\end{aligned}
$$

The following theorem shows that the time to answer these sequence of queries can be related to the size of this working set.

Theorem 7. Let $q_{1}, \ldots, q_{m}$ be a sequence of $\varepsilon$-approximate nearest neighbor queries. For any point $f \in \mathbb{R}^{d}$ and any positive constant $c$, the total time to answer all these queries is $O\left(m\left(\frac{1}{\varepsilon}\right)^{d} \log \left(\left|W_{\mathrm{ann}}\right|+\left(\frac{1}{c}\right)^{d-1}\right)+n \log n\right)$.

## References

1. Arya, S., Malamatos, T., Mount, D.M.: A simple entropy-based algorithm for planar point location. ACM Trans. Algorithms, 3, article: 17 (2007)
2. Arya, S., Malamatos, T., Mount, D.M., Wong, K.-C.: Optimal expected-case planar point location. SIAM J. Comput. 37, 584-610 (2007)
3. Arya, S., Mount, D.M.: Approximate range searching. Comput. Geom. Theory Appl. 17, 135-163 (2001)
4. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor searching. J. Assoc. Comput. Mach. 45, 891-923 (1998)
5. Bern, M.: Approximate closest-point queries in high dimensions. Inform. Process. Lett. 45, 95-99 (1993)
6. Bădoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A unified access bound on comparison-based dynamic dictionaries. Theo. Comp. Sci. 382, 86-96 (2007)
7. Chan, T.: A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions (2006),
http://www.cs.uwaterloo.ca/~tmchan/pub.html (unpublished manuscript)
8. Chan, T.M.: Closest-point problems simplified on the ram. In: Proc. 13th Annu. ACM-SIAM Sympos. Discrete Algorithms, pp. 472-473 (2002)
9. Clarkson, K.L.: Fast algorithms for the all nearest neighbors problem. In: Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci., pp. 226-232 (1983)
10. Cole, R.: On the dynamic finger conjecture for splay trees. Part 2 The proof. SIAM J. Comput. 30, 44-85 (2000)
11. Cole, R., Mishra, B., Schmidt, J., Siegel, A.: On the dynamic finger conjecture for splay trees. Part 1: Splay sorting log $n$-block sequences. SIAM J. Comput. 30, 1-43 (2000)
12. Demaine, E.D., Harmon, D., Iacono, J., Pătraşcu, M.: The geometry of binary search tree. In: Proc. 20th Annu. ACM-SIAM Sympos. Discrete Algorithms, pp. 496-505 (2009)
13. Elmasry, A.: On the sequential access theorem and deque conjecture for splay trees. Theo. Comp. Sci. 314, 459-466 (2004)
14. Eppstein, D., Goodrich, M.T., Sun, J.Z.: The skip quadtree: A simple dynamic data structure for multidimensional data. Internat. J. Comput. Geom. Appl. 18, 131-160 (2008)
15. Georgakopoulos, G.F.: Chain-splay trees, or, how to achieve and prove $\log \log N$ competitiveness by splaying. Inform. Process. Lett. 106, 37-43 (2008)
16. Har-Peled, S.: Geometric Approximation Algorithms. American Mathematical Society, Providence (2011)
17. Iacono, J.: Alternatives to splay trees with $O(\log n)$ worst-case access times. In: Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algorithms, pp. 516-522 (2001)
18. Iacono, J.: Distribution-sensitive data structures. PhD thesis, Rutgers, The state University of New Jersey, New Brunswick, New Jersey (2001)
19. Iacono, J.: Expected asymptotically optimal planar point location. Comput. Geom. Theory Appl. 29, 19-22 (2004)
20. Iacono, J.: Key-independent optimality. Algorithmica 42, 3-10 (2005)
21. Mount, D.M., Park, E.: A dynamic data structure for approximate range searching. In: Proc. 26th Annu. Sympos. Comput. Geom., pp. 247-256 (2010)
22. Pettie, S.: Splay trees, Davenport-Schinzel sequences, and the deque conjecture. In: Proc. 19th Annu. ACM-SIAM Sympos. Discrete Algorithms, pp. 1115-1124 (2008)
23. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33, 668-676 (1990)
24. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann, San Francisco (2006)
25. Seidel, R., Aragon, C.: Randomized search trees. Algorithmica 16, 464-497 (1996)
26. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. J. Assoc. Comput. Mach. 32, 652-686 (1985)
27. Subramanian, A.: An explanation of splaying. J. Algorithms 20(3), 512-525 (1996)
28. Sundar, R.: On the deque conjecture for the splay algorithm. Combinatorica 12 , 95-124 (1992)
29. Tarjan, R.E.: Sequential access in splay trees takes linear time. Combinatorica 5, 367-378 (1985)
30. Wang, C.C., Derryberry, J., Sleator, D.D.: $o(\log \log n)$-competitive dynamic binary search trees. In: Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms, pp. 374-383 (2006)

[^0]:    * This work has been supported by the National Science Foundation under grant CCF-1117259 and the Office of Naval Research under grant N00014-08-1-1015.

