A Dynamic Data Structure for Approximate Range Searching

Eunhui Park
Department of Computer Science
University of Maryland
College Park, Maryland
ehpark@cs.umd.edu

Abstract

In this paper, we introduce a simple, randomized dynamic data structure for storing multidimensional point sets. Our particular focus is on answering approximate geometric retrieval problems, such as approximate range searching. Our data structure is essentially a partition tree, which is based on a balanced variant of the quadtree data structure. As such it has the desirable properties that it forms a hierarchical decomposition of space into cells, which are based on hyperrectangles of bounded aspect ratio, each of constant combinatorial complexity.

In any fixed dimension d, we show that after $O(n \log n)$ preprocessing, our data structure can store a set of n points in $\mathbb{R}^d$ in space $O(n)$ and height $O(\log n)$. It supports insertion in time $O(\log n)$ and deletion in time $O(\log^2 n)$. It also supports ε-approximate spherical range (counting) queries in time $O\big(\log n + (\frac{1}{\varepsilon})^{d-1}\big)$, which is optimal for partition-tree based methods. All running times hold with high probability. Our data structure can be viewed as a multidimensional generalization of the treap data structure of Seidel and Aragon.

Keywords: Geometric data structures, dynamic data structures, quadtrees, range searching, approximation algorithms.

1 Introduction

Recent decades have witnessed the development of many data structures for efficient geometric search and retrieval. A fundamental issue in the design of these data structures is the balance between efficiency, generality, and simplicity. One of the most successful domains in terms of simple, practical approaches has been the development of linear-sized data structures for approximate geometric retrieval problems involving points sets in low dimensional spaces. In this paper, our particular interest is in dynamic data structures, which allow the insertion and deletion of points for use in approximate retrieval problems, and range searching in particular. A number of data structures have been proposed for approximate retrieval problems for points sets. The simplicity and general utility of these data structures is evident by the wide variety of problems to which they have been applied. We will focus on methods based on partition trees, and in particular, on those based on quadtrees and their multidimensional extensions (see, e.g., [18]). One nice feature of the quadtree, which makes it appropriate for approximate retrieval problems, is that it decomposes space into disjoint hypercubes, which are of constant combinatorial complexity and bounded aspect ratio, two important properties in geometric approximation.

Because the size and depth of a quadtree generally depends on the ratio of the maximum and minimum point distances, it is common to consider the compressed quadtree [5,9,16], which can store n points in space $O(n)$, irrespective of their distribution. A compressed quadtree may have height $\Theta(n)$, and so it is often combined with an auxiliary search structure for efficient access. This can be done by applying some form of centroid decomposition to the tree [10]or by imposing a general-purpose data structure, like a link-cut

tree [6, 20]. These approaches are rather inelegant, however, due to their reliance on relatively heavyweight auxiliary structures.

A much simpler and more elegant solution, called the *skip quadtree*, was proposed by Eppstein *et al.* [14]. It consists of a series of compressed quadtrees, each for a successively sparser random sample of the point set, together with links between corresponding nodes of consecutive trees. The result is a multidimensional analog of the well known skip list data structure [17]. The skip quadtree can be used to answer approximate nearest neighbor queries and approximate range-reporting queries. Another creative solution, due to Chan [7, 8], involves a simple in-place data structure based on the bit-interleaved ordering of point coordinates. He shows that this trivial "data structure," which is really just a sorted array of points, supports efficient nearest neighbor searching without explicitly storing any tree. The general approach of linearizing point sets through bit interleaving is well known, and is the basis of a structure called the linear quadtree [15]. Another approach for achieving balance is to modify the definition of the decomposition process. Examples of this include the BBD-tree [2–4], which uses an operation called centroid-shrinking to produce a balanced tree, and the BAR tree [13], which uses splitting planes that are not axis-aligned.

As mentioned above, our interest is in approximate range counting. Consider a set P of n points in $\mathbb{R}^d$, for some constant d. It is common to assume that each point $p \in P$ is associated with a numeric weight, $\text{wgt}(p)$. After preprocessing, we are given a query range Q, and the problem is to compute the sum of weights of the points lying within Q. It is generally assumed that the weights are drawn from a commutative semigroup. In some cases the semigroup is actually a group, and if so, subtraction of weights is also allowed. In *approximate range searching*, the range Q is assumed to be approximated by two convex shapes, an inner range Q^- and an outer range Q^+, where $Q^- \subset Q^+$. Points lying within the inner range must be counted and points lying outside the outer range must not. It is assumed that the boundaries of these two shapes are separated by a distance of at least $\varepsilon \cdot \text{diam}(Q)$. Space and query times are expressed as a function of both ε and n. The only other assumption imposed on the query shapes is the *unit-cost assumption*, which states that it is possible to determine whether a quadtree box lies entirely inside Q^+ or entirely outside Q^- in constant time. Arya and Mount [3] showed that, under the unit-cost assumption, approximate range queries could be answered with space $O(n)$ and query time $O\big(\log n + (\frac{1}{\varepsilon})^{d-1}\big)$. They showed that this is essentially optimal for methods based on partition trees. Although data structures exist with asymptotically better performance, these methods require ε to be fixed at construction time. The BBD-tree has the nice feature that preprocessing is independent of ε, and hence a single structure can be used to answer queries of all degrees of precision.

One significant shortcoming of the aforementioned data structures, is that none of them admits an efficient solution to the important problem of approximate range counting together with efficient insertion and deletion. The BBD-tree and BAR tree do not support efficient insertion and deletion (except in the amortized sense, through the process of completely rebuilding unbalanced subtrees [1, 12]. Chan's implicit data structure and the skip quadtree both support efficient insertion and deletion, but neither supports range-counting queries. Intuitively, to support range-counting queries, each internal node of a partition tree maintains the total weight of all the points residing in the subtree rooted at this node. With each insertion or deletion, these counts are updated from the point of insertion to the tree's root. In Chan's structure there is no tree, and hence no internal nodes. The skip quadtree is based on the compressed quadtree, which may have height $\Theta(n)$. (The skip quadtree does support efficient range-reporting queries, however, because even an unbalanced subtree of size k can be traversed in $O(k)$ time.)

In this paper, we introduce a simple dynamic data structure, which supports efficient approximate range counting and approximate nearest neighbor searching. This tree structure is similar in spirit to the BBD-tree, but whereas the BBD-tree is static, our data structure employs a dynamic rebalancing method based on rotations. Similar to the skip quadtree, which is based on combining quadtrees and the 1-dimensional skip list, our data structure can be viewed as a combination of the quadtree and the treap data structure of Seidel and Aragon [19]. A *treap* is a randomized data structure for storing 1-dimensional keys, which is based on a combination of a binary search tree and a heap. A treap assigns random priorities to the keys, and the tree behaves at all times as if the points had been inserted in order of increasing priority. It relies on the fact that, if nodes are inserted in random order into a binary tree, then the tree's height is $O(\log n)$ with

high probability. Our structure also assigns priorities to points, and maintains the tree as if the points had been inserted in this order. Our tree structure maintains something akin to the heap property of the treap (but each node of our structure is associated with two priority values, not one). Because of its similarity to the treap, we call our data structure the *quadtreap*. Our main result is stated below.

Theorem 1.1 *Given a set of n points in $\mathbb{R}^d$, a quadtreap storing these points has space $O(n)$. The tree structure is randomized and, with high probability, it has height $O(\log n)$. Letting h denote the height of the tree:*

(*i*) *It supports point insertion in time $O(h)$.*

(*ii*) *It supports point deletion in worst-case time $O(h^2)$, and in expected-case time (averaged over all the points of the tree) $O(h)$.*

(*iii*) *Approximate range-counting queries can be answered in time:*

- *$O\big(h + (\frac{1}{\varepsilon})^{d-1}\big)$, when the point weights are drawn from a commutative group,*
- *$O\big(h \cdot (\frac{1}{\varepsilon})^{d-1}\big)$, when the point weights are drawn from a commutative semigroup.*

(*iv*) *Approximate range-reporting queries can be answered in time $O\big(h + (\frac{1}{\varepsilon})^{d-1} + k\big)$, where k is the output size.*

(*v*) *Approximate nearest neighbor queries can be answered in time $O\big(h + (\frac{1}{\varepsilon})^{d-1}\big)$.*

For approximate range-counting, it is assumed that the inner and outer ranges are convex, and satisfy the unit-cost assumption. (See the definition given above).

The rest of the paper is organized as follows. In Section 2 we introduce the BD-tree data structure and present a simple incremental algorithm for its construction. In Section 3 we present our dynamic data structure and show how insertions and deletions are performed. In Section 4 we present the range query algorithm. We do not explicitly present approximate nearest neighbor searching or range reporting, but these problems are substantially simpler than range searching, and the techniques for solving them are straightforward extensions of existing methods.

2 Incremental Construction of a BD-Tree

To motivate our data structure, 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. Given a set P of points, this decomposition process is applied until each cell contains at most one point, and these terminal cells form the leaves of the trees. A *compressed quadtree*, is obtained by replacing all maximal chains of nodes that have a single non-empty child by a single node associated with the coordinates of the smallest quadtree box containing the data points. The size of a compressed quadtree is $O(n)$, and it can be constructed in time $O(n \log n)$ time (see, e.g., [9]).

If the dimension of the space is much larger than a small constant, it is impractical to split each internal node into 2^d cells, many of which may contain no points. For this reason, we 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. Each cell has constant aspect ratio, and so the essential packing properties of cells still hold. Henceforth, we use the terms *compressed quadtree* and *quadtree box* in this binary context. As with standard quadtree boxes, it is easy to prove that two quadtree boxes in this binary context are either spatially disjoint or one is nested within the other. Assuming a model of computation that supports exclusive bitwise-or and base-2 logarithms on point coordinates, the smallest quadtree box enclosing a pair of points (or more generally a pair of quadtree boxes) can be computed in constant time.

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*. The *size* of a cell is defined to be the maximum side length of its outer box. Although cells are not convex, they have bounded aspect ratio and are of constant combinatorial complexity.

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. Each crowded cell is partitioned into two smaller cells by one of two partitioning operations (see Fig. 1). The first operation, called a *split*, partitions a cell by an axis-orthogonal hyperplane in the same manner described above for the binary quadtree. The associated node has two children, one associated with each of these cells, called the *left child* and *right child*. In contrast to traditional partition-tree definitions, we do not assume that the coordinates of points in the left node are smaller than those of the the right node. Instead (for reasons to be discussed later), it will be convenient to assume that, if the original cell contains an inner box, this inner box lies within the left cell. We call this the *inner-left convention*. Otherwise, the nodes may be labeled left and right arbitrarily.

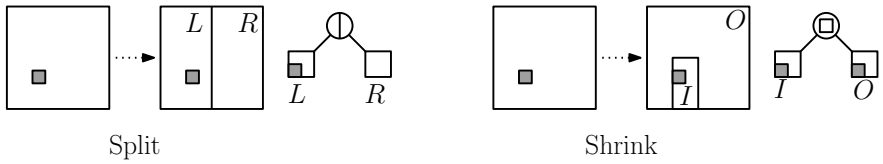

Fig. 1: Splitting (left) and shrinking (right). In each case the initial cell contains an inner box, which is indicated by a small gray square.

The second operation, called a *shrink*, partitions a cell by a quadtree box (called the *shrinking box*), which lies within the cell. It partitions the cell into two parts, one lying within the shrinking box (called the *inner cell*), and the other being the set-theoretic difference of the original cell and the shrinking box (called the *outer cell*). The associated child nodes are called the *inner child* and *outer child*, respectively. (In our figures, the inner child will always be on the left side.)

We will maintain the invariant that, whenever a shrink operation is applied to a cell that contains an inner box, the shrinking box (properly) contains the inner box. This implies that a cell never has more than one inner box. We allow a degenerate shrink to occur, where the shrinking box is equal to the cell's outer box. In this case, the outer box is said to be a *vacuous leaf* since it has no area, and hence no points can be inserted into it. In summary, there are two types of internal nodes in a BD-tree, *split nodes* and *shrink nodes*, depending on the choice of the partitioning operation.

The decomposition process terminates when the cell associated with the current node contains either a single point or a single inner box. The resulting node is a leaf, and the associated point or inner box is called *contents*.

2.1 Incremental Construction Algorithm

In [3,4], a bottom-up method for constructing a BD-tree of height $O(\log n)$ was proposed. The starting point for our discussion is a simple incremental algorithm for constructing a BD-tree through repeated insertion. This construction does not guarantee logarithmic height, but in Section 2.3 we will show that, if the points are inserted in random order, the height will be $O(\log n)$ with high probability. Ignoring the trivial case, where the tree is empty, we will maintain the invariant that the cell associated with each leaf node contains either a single point or a single inner box, which we call its *contents*.

Given a point set $P = \{p_1, \ldots, p_n\}$, the incremental construction algorithm begins with an empty tree, whose only cell is the unit hypercube, and it proceeds by repeatedly inserting each point $p \in P$ into the tree as follows. First, through a top-down descent, we find the leaf node u that contains p (see Fig. 2). Let b denote u's contents (either a point or an inner box), and let C denote u's outer box. We compute the minimum quadtree box enclosing both p and b, denoted E. By the minimality of E and basic properties of quadtree boxes, applying a split to E will produce two cells, one containing b and the other containing p.

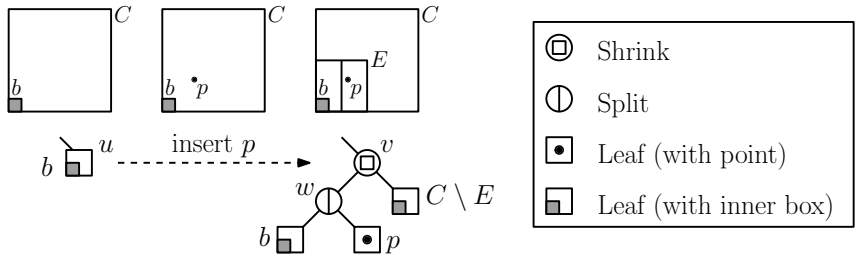

Fig. 2: Insertion algorithm

We replace u with a pair of internal nodes v and w. The node v shrinks from u's outer box down to E. The outer child of v is a leaf node with outer box C and inner box E. (Note that E might be equal to u's outer box C), in which case the outer child is a vacuous leaf.) The inner child of v is a split node w that separates b from p. The children of w are two leaves. By the inner-left convention, the left child has b as its contents, and the right child contains p. (Recall that the left child does not necessarily correspond to points with smaller coordinates.) Observe that this insertion process generates nodes in pairs, a shrink node whose inner child is a split node. Irrespective of the insertion order, the following invariants are preserved after creation:

- Each cell contains at most one inner box.
- Each leaf cell contains either a single point or a single inner box (unless the tree is empty, in which case it contains nothing).
- The inner child of each shrink node is a split node.

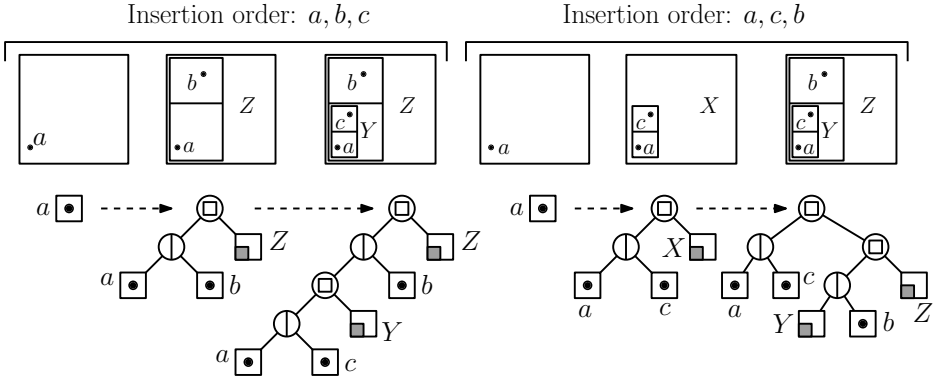

Fig. 3: The tree on the left has been created with the sequence $\langle a, b, c\rangle$ and the one on the right with $\langle a, c, b\rangle$. Uppercase letters identify the outer cells of shrink nodes.

An example showing two different insertion orders is given in Fig. 3. Although the tree structure depends on the insertion order, the following lemma shows that the subdivision induced by the tree does not. Due to space limitations, the proofs of all the results in the paper appear in the appendix.

Lemma 2.1 *Let T be a BD-tree resulting from applying the incremental algorithm to a set P of points. The subdivision induced by the leaf cells of T is independent of the insertion order.*

2.2 Node Labels and the Heap Property

In this section we show that, if points are labeled with their insertion times, then it is possible to assign labels to the internal nodes of the tree based on the labels of its descendant leaves, so that these labels satisfy a heap ordering. This observation is analogous to the heap condition in the treap data structure of Seidel and Aragon [19].

To motivate this labeling, observe that each newly created internal node is added by the incremental algorithm in order to separate two entities, the newly inserted point and the existing contents of the leaf node in which the new point resides. If we were to label every point with its insertion time (and also provide an appropriate label value to each inner box), it follows that the creation time of any internal node is the *second smallest* label over all the points in the subtree rooted at this node. This suggests a labeling approach based on maintaining the smallest and second smallest labels of the descendant points.

To make this more formal, we define the following *node-labeling rule*. First, we define two symbols, $\downarrow$ and $\uparrow$ denote, respectively, numeric values smaller and larger than any insertion times. We create a labeled pair for each node of the BD-tree. First, each leaf that contains a point is labeled with the pair $(t, \uparrow)$, where t is the insertion time of the associated point. Each leaf that contains only an inner box is given the label $(\downarrow, \uparrow)$. For each internal node u, let v and w be its two children, and let $(v^{[1]}, v^{[2]})$ and $(w^{[1]}, w^{[2]})$ denote the labels of its children. If u is a split node it has the label

$$\big(u^{[1]}, u^{[2]}\big) \;\leftarrow\; \big(\min\big(v^{[1]}, w^{[1]}\big), \max\big(v^{[1]}, w^{[1]}\big)\big).$$

If u is a shrink node, it is given the label of its inner child (the split node). An example of such a labeling is shown in Fig. 4. The following lemma establishes some useful observations regarding the relationships between the labels of nodes in the tree and the insertion times of the points in their associated subtrees.

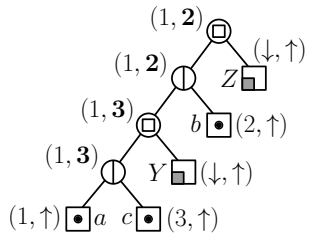

Fig. 4: Node labeling for the insertion order $\langle a, b, c\rangle$.

Lemma 2.2 *Consider the labeled BD-tree resulting from the incremental insertion algorithm. For any node u in this tree:*

- *(i) If u's cell has an inner box, then $u^{[1]} = \downarrow$, and in particular, this holds for the outer child of any shrink node. Otherwise $u^{[1]}$ is equal to the earliest insertion time of any point in u's subtree.*
- *(ii) If u is an internal node, then $u^{[2]}$ is equal to the node's creation time, and in particular, $u^{[2]} \notin \{\downarrow, \uparrow\}$.*
- *(iii) If u is a shrink node, then for every node v in its outer subtree $v^{[2]} > u^{[2]}$.*

With the aid of this observation, the node labels satisfy the following properties. The second property is the heap property.

Lemma 2.3 *The labeled BD-tree resulting from the incremental insertion algorithm satisfies the following properties:*

- *(i) If v and w are left and right children of a split node, respectively, then $v^{[1]} < w^{[1]}$.*
- *(ii) Given any pair consisting of a parent u and child v, we have $u^{[2]} \le v^{[2]}$. If u is a split node, the inequality is proper.*

2.3 Randomized Incremental Construction

In this section we show that, if points are inserted in random order into the BD-tree by the above incremental algorithm, with high probability, the resulting tree has $O(\log n)$ height. We begin by showing that the search depth of any query point is $O(\log n)$ in expectation (over all possible insertion orders). The proof is a standard application of backwards analysis [11]. It is based on the observations that (1) the subdivision induced by the tree is independent of the insertion order (as shown in Lemma 2.1), and (2) the existence of each cell of the subdivision depends only on a constant number of points. Thus, the probability that any one insertion affects the leaf cell containing the query point is inversely proportional to the number of points inserted thus far.

Lemma 2.4 *Consider the BD-tree resulting from incrementally inserting an n-element point set P in random order. Given an arbitrary point q, the expected depth in the tree (averaged over all insertion orders) of the leaf containing q is $O(\log n)$.*

The following theorem strengthens the above result by showing that the height bound holds with high probability. A bound on the construction time and space follow immediately. Again, this is a straightforward generalization of analyses of other randomized incremental algorithms (see, e.g., [11]).

Theorem 2.1 *Consider the BD-tree resulting from incrementally inserting an n-element point set in random order. The tree has space $O(n)$ (unconditionally). With high probability, the height of the tree is $O(\log n)$, and the time needed to construct the tree is $O(n \log n)$.*

3 The Quadtreap

In this section we define our dynamic data structure, which we call the *quadtreap*. Before presenting the algorithm, we discuss the basic rebalancing operation upon which the data structure is based.

3.1 Pseudo-nodes and Rotation

The BD-tree depends on the insertion order, but we will show in this section that it can be rebalanced through a simple local operation called a *rotation*. This operation is much like the familiar rotation operation defined on binary search trees, but some modifications will be needed in the context of BD-trees. This operation will be a central primitive in maintaining balance in our data structure as points are inserted and deleted.

Throughout this section we assume that the BD-tree satisfies the property that the internal nodes of the BD-tree can be partitioned into pairs, such that each pair consists of a shrink node as a parent and a split node as its inner child. We call this the *shrink-split property*. As mentioned in Section 2, the BD-tree resulting from our incremental construction algorithm satisfies this property.

This assumption allows us to view each shrink-split combination as single decomposition operation, which subdivides a cell into three subcells. Thus, we can conceptually merge each shrink-split pair into a single node, called a *pseudo-node* (see Fig. 5), which has three children. The first two children, called the *left* and *right child*, correspond to the children of the split node, and the third child, called the *outer child*, corresponds to the outer child of the shrink node. Note that this is a conceptual device and does not involve any structural changes to the tree.

Let us now define rotation on pseudo-nodes. A rotation is specified by giving a node y of the tree that is either a left child or an outer child. Let x denote y's parent. If y is a left child, the operation promote(y) makes x the outer child of y, and y's old outer child becomes the new left child of x. Observe that the inner-left convention is preserved, since y's left child is unchanged, and x's left child (labeled w in the figure) contains the inner box (consisting of the union of A and B's cells in the figure).

Next, let us consider the promotion of an outer child x. Let y denote its parent. The operation promote(x) is the inverse of the previously described left-child promotion. It makes y the left child of x, and x's old left child becomes the new outer child of y. Again, the inner-left convention is preserved, since prior to the

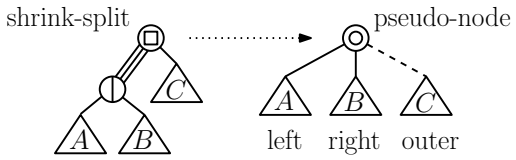

Fig. 5: The pseudo-node corresponding to a shrink-split pair.

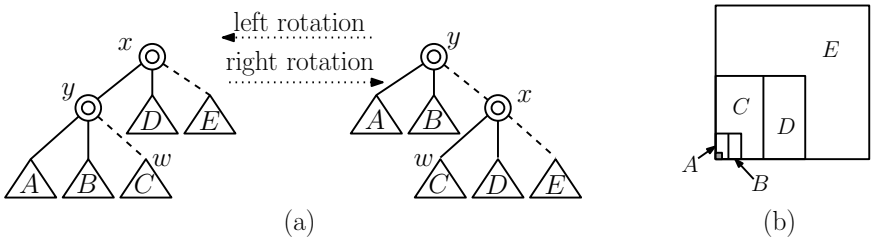

Fig. 6: Rotation operation on pseudo-nodes (a). The associated subdivision is shown in (b). (The order of left-right children is based on the assumption that the earliest insertion time (or inner box, if present) lies within the cell associated with subtree A.)

rotation, if y's cell had an inner box it must lie in its left child A, and so after rotation it lies in x's left child y. (The reason for the inner-left convention is to allow us to define one rotation rule. Otherwise, there would need to be a third rotation rule involving the right-child edge.)

Based on the above descriptions, it is easy to see that the essential properties of the BD-tree are preserved after applying either rotation operation. Since it is a purely local operation, rotation can be performed in $O(1)$ time.

3.2 Point Insertion

We have seen that, if the points are inserted in random order, the height of the BD-tree is $O(\log n)$ with high probability, but a particularly bad insertion order could result in a tree of height $\Omega(n)$. In order to make the tree's height independent of the insertion order, we assign each point a random numeric *priority* at the time it is inserted, and we maintain the tree *as if* the points had been inserted in increasing priority order. This is essentially the same insight underlying the treap data structure [19]. Intuitively, since the priorities are not revealed to the user of the data structure, the tree effectively behaves as if the insertion order had been random.

The tree's structure is determined by the node-labeling rule described in Section 2.2 and the properties of Lemmas 2.2 and 2.3. Recall that the labeling procedure of that section gives both nodes of each shrink-split pair the same label. We define the label of each pseudo-node to be this value. Whenever a new point is inserted into or deleted from the tree, we need to restructure the tree so that its structure is consistent with the heap property. Thus, the tree behaves *as if* the nodes of the tree had been inserted in priority order. We begin by presenting the insertion algorithm.

A newly inserted point is assigned a random priority $0 < t < 1$. It is then inserted into the tree, by the incremental algorithm. The newly created nodes (three leaves, one split node, and one shrink node) generated by the incremental algorithm are labeled as if the insertion took place at time t. The resulting tree may violate the heap properties, however, and the tree needs to be restructured to restore this property. The algorithm for restoring the tree structure is presented in Algorithm 1. Recall that the label of a node u is denoted by $(u^{[1]}, u^{[2]})$. The restructuring procedure makes use of the following simple utility functions:

- adjust-left-right(u): Let v and w be the left and right children of u, respectively. If $w^{[1]} < v^{[1]}$, swap

these children. (This enforces property (i) of Lemma 2.3.)

- update-label(u): Let w_1 and w_2 denote u's left and right children, respectively, Set

$$(u^{[1]}, u^{[2]}) \leftarrow (w_1{}^{[1]}, w_2{}^{[1]}).$$

Finally, the operation update(u) invokes both of these functions on u.

Algorithm 1: Tree restructuring after inserting a point in a new leaf node v.

```
1  while v ≠ root(T) do
2  |  u ← parent(v);
3  |  if v is a left or right child then
4  |  |  update(u);
5  |  |  if v^[2] < u^[2] then
6  |  |  |  promote(v);  update(u);
7  |  |  else v ← u;
8  |  else if v^[2] < u^[2] then
9  |  |  promote(v);  update(v);
10 |  else break
```

An example of the insertion and subsequent restructuring is shown in Fig. 7. (For ease of illustration, we assume that priorities are positive integers.) Consider the insertion of a point with priority 2, which lies within the leaf node indicated in Fig. 7(a). This insertion creates a pseudo-node v with label $(\downarrow, 2)$, which violates Lemma 2.3(ii) with respect to its parent u (see (b)), thus causing a right rotation. After the rotation, there is a violation of Lemma 2.3(ii) between v and its new parent, whose label is $(\downarrow, 6)$ (see (c)), resulting in another right rotation. There is now violation of the heap property with the root node (see (d)), causing a left rotation. At this point v is the root, and we obtain the final tree (see (e)).

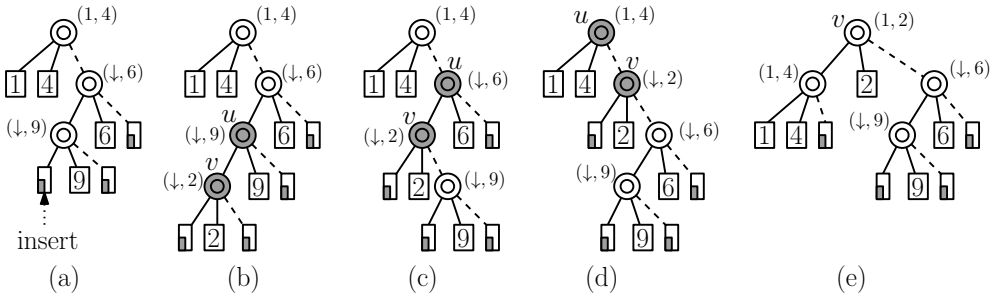

Fig. 7: Example of quadtreap restructuring after insertion.

The following lemmas establish the correctness and running time of this procedure. The proof involves a straightforward induction involving a case analysis of the relationships between the labels of nodes before and after each rotation. Together, they establish Theorem 1.1(i).

Lemma 3.1 *After inserting a point and applying Algorithm 1, the quadtreap satisfies Lemma 2.3.*

Lemma 3.2 *The time to insert a point in a quadtreap of size n and height h is $O(h)$. Thus, the insertion time is $O(\log n)$ with high probability.*

Proof: The rotation and update of priority values can be performed in $O(1)$ time per node. The time needed to locate the point to be inserted is $O(h)$. The time to insert the point is $O(1)$. Since the algorithm traces the path from the insertion point to the root, the time to perform the restoration of the heap properties by

Algorithm 1 is $O(h)$. Thus, the entire insertion time is $O(h)$. By Theorem 2.1, $h = O(\log n)$ time with high probability. □

3.3 Point Deletion

Next, we consider the deletion of a point, p. We first determine the leaf node v containing this point. We handle deletion by reversing the insertion process. In particular, we effectively set the deleted point's priority to $\uparrow$ by assigning v a label of $(\uparrow, \uparrow)$. We then restructure the tree, described below, so that the properties of Lemmas 2.3 are satisfied. Because of p's high priority, the tree's structure would be the same as if p had been the last point to be inserted. When viewed from the perspective of pseudo-nodes, when a point is inserted into the tree, it is placed within a leaf cell that is the right child of its parent, and its two siblings are both leaves. To establish this claim, observe first that v cannot be an outer child because the cell associated with an outer child has an inner box. Since v contains a point, it cannot also contain an inner box. Also, v cannot be a left child since this would violate Lemma 2.3(i). Therefore, v is the right child of its parent. To show that v's siblings are leaves, note that by the node-labeling rule, v's parent's second label component is $\uparrow$. If its siblings were not leaves, their second label components would be strictly smaller than this, thus violating property (ii) of Lemma 2.3. Therefore, after restructuring, we can delete p by simply "undoing" the insertion procedure by replacing the subtree rooted at v's parent with a single leaf node. All that remains is to describe the above restructuring process. This is given in Algorithm 2.

Algorithm 2: Restructuring the tree as part of the deletion of a point in a leaf node v.

```
1  label(v) ← (↑, ↑);
2  r ← parent(v);
3  while r ≠ null do
4      u ← r;  r ← parent(r);
5      update(u);
6      w ← left(u);  x ← outer(u);
7      while w[2] < u[2] or x[2] < u[2] do
8          if w[2] < x[2] then
9              promote(w);  update(u);
10             w ← left(u);
11         else
12             promote(x)  update(x);
13             x ← outer(u);
```

An example of the restructuring for the deletion process is shown in Fig. 8. (For ease of illustration, let us assume that priorities are positive integers.) The point to be deleted lies in node v (see (a)). We set its label to $(\uparrow, \uparrow)$, set u to v's parent and then invoke the update procedure on line 5 of Algorithm 2. As a result, u's left and right children are swapped, and u's label is updated (see (b)). A sequence of rotations is then applied to restore the heap properties (see (c)–(e)). Finally, the subtree rooted at v's parent is replaced by a single leaf node (see (f)).

The following lemmas establish the correctness of this procedure. As with insertion, the proof involves a straightforward induction involving a case analysis of the relationships between the labels of nodes before and after each rotation.

Theorem 3.1 *After deleting a point and applying the restructuring procedure of Algorithm 2, the quadtreap satisfies Lemma 2.3.*

Deletion requires $O(\log^2 n)$ time. Intuitively, the reason that deletion is more complicated than insertion is because the deleted node's priority may exist among the label components along the entire search path

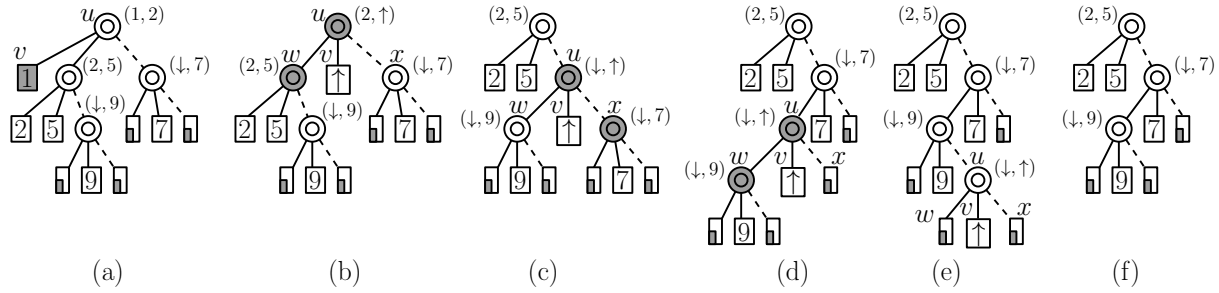

Fig. 8: Example of deletion from a quadtreap.

to the root. (Consider, for example, the deletion of the point with the lowest priority in the tree.) Deleting the point involves updating the label of each such node. Each update may result in $O(\log n)$ rotations.

Lemma 3.3 *The time to delete a point from a quadtreap of size n and height h is $O(h^2)$. Thus, the deletion time is $O(\log^2 n)$ with high probability.*

Proof: The rotation and update of each label can be performed in time $O(1)$. Each iteration of the inner loop of Algorithm 2 takes time proportional to the height of node u. This inner loop may be repeated for each ancestor of the leaf v containing the deleted point. Therefore, the total deletion time is at most $O(h^2)$. Since the depth of the tree by Theorem 2.1 is $O(\log n)$ with high probability, the deletion time is $O(\log^2 n)$, also with high probability. □

The worst case scenario for point deletion is rather pessimistic. The following lemma shows that the expected time to delete a random point of the tree is linear in the tree's height. The expectation is over the choices of which point to delete, whereas the high-probability bound is over the assignment of priorities. This establishes Theorem 1.1(ii).

Lemma 3.4 *The expected time to delete a random point from a quadtreap T of size n and height h is $O(h)$. Thus, the expected deletion time is $O(\log n)$ with high probability*

Proof: Before giving the proof, we begin with a useful observation. Consider a node u visited in an arbitrary iteration of the outer while loop of Algorithm 2 during the deletion of some point p. We claim that the inner loop will be entered only if p's priority is among the two smallest in the subtree rooted at u. First, observe u is an ancestor of the leaf containing p. The second component of u's label is equal to the second smallest priority in u's subtree, which implies that this components value can be changed only if p has one of the two smallest priorities in u's subtree. If this observation holds for p and u, p applies a charge to u.

Define an indicator variable, $\chi(u, p)$ as follows.

$$\chi(u,p) \;=\; \begin{cases} 1 & \text{if } p \text{ charges } u, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the expected deletion time t is

$$E(t) \;\le\; \frac{1}{n} \sum_{p \in P} \sum_{u \in T} height(u) \cdot \chi(u,p).$$

Each node is charged by at most two points, and thus,

$$\begin{aligned} E(t) \;&\le\; \frac{1}{n} \sum_{u \in T} 2 \cdot height(u) \;\le\; \frac{1}{n} \cdot n \cdot 2h \\ &=\; O(h) \;=\; O(\log n), \end{aligned}$$

as desired. □

4 Approximate Range Queries

In this section we present an algorithm for answering approximate range-counting queries using the quadtreap. Recall the problem description from Section 1. Because rotations will not be needed here, most of the description here will be based on the simpler representation of the tree as a BD-tree with split and shrink nodes.

In order to provide efficient response to queries, we will need to add a couple of enhancements to the basic data structure. Before discussing these enhancements, let us recall that we are given a set P of n points $\mathbb{R}^d$, for fixed d. Let $\text{wgt}(p)$ denote the weight of point p. Recall that a query is presented in the form of two convex shapes, an inner range and outer range, Q^- and Q^+, respectively, whose boundaries are separated by a distance of at least $\varepsilon \cdot \text{diam}(Q)$. Let T be a BD-tree storing these points, and let h denote its height. For the sake of generality, we will express running times (deterministically) in terms of n, h, and ε, but as shown in the previous sections, if the BD-tree is a quadtreap, then h will be $O(\log n)$ with high probability. Our best results are for range counting where the weights are drawn from a commutative group, but we will also consider the case of commutative semigroups and range reporting.

The first enhancement to the tree is to maintain at each node v the total weight of the points in the associated subtree. Given a node v, let P_v denote the points of P lying within v's subtree, and let $\text{wgt}(v)$ denote their total weight. Observe that this sum can be updated as points are inserted or deleted, without altering the construction time or space bounds.

The second enhancement is a structural addition to the tree. To motivate the addition, it is useful to recall how approximate range queries are answered in partition trees. We apply a standard top-down recursive algorithm (see for example [3]). Starting at the root, let v denote the current node being visited. We compare v's cell C to the range. If the cell lies outside the inner range or inside the outer range or is a leaf, we can process it in $O(1)$ time. Otherwise, we recurse on its children. (See the code block.)

Procedure range(v, Q)

```
1  C ← cell(v);
2  if C ∩ Q⁻ = ∅ then return 0;
3  else if C ⊆ Q⁺ then return wgt(v);
4  else if v is a leaf then
5  |   if v contains a point in Q then return wgt(p);
6  |   else return 0;
7  else if v is a split node then
8  |   return range(left(v), Q) + range(right(v), Q);
9  else // v is a shrink node
10 |   return range(inner(v), Q) + range(outer(v), Q);
```

The key issue in analyzing the algorithm is determining the number of internal nodes that are *expanded*, which means that a recursive call is made to the node's two children. Recall that the *size* of a cell in the tree is the maximum side length of its outer box. The analysis classifies expanded nodes as being of two types: *large nodes* are those whose cell size is at least $2 \cdot \text{diam}(Q)$, and remainder are called *small nodes*. To understand the issue, recall the analysis of the small expanded nodes given in [3]. It relies on an crucial property of the BBD-tree, namely that whenever the search descends a constant number of levels, the size of the associated cells decreases by a constant factor. This property holds for the binary quadtree, since with each d levels of descent in the tree, every side is bisected and so the cell size is halved. However, this generally does not hold for BD-tree. The problem arises from the fact that a the outer child of a shrink node has the same outer box as its parent. Thus, there may arbitrarily long chains of shrink nodes, for which the

cell size does not decrease. We define a *shrink chain* to be a maximal sequence of shrink nodes, where each node is the outer child of its predecessor.

We fix this problem by adding a special pointer, called a *link*, which points from the first node of each shrink chain to its last node (see Fig. 9). We also assume that each node of the tree stores a pointer to its parent. A BD-tree with all the above enhancements (internal weights, parent pointers and links) is called an *augmented BD-tree*. A quadtreap based on such a tree is called an *augmented quadtreap*.

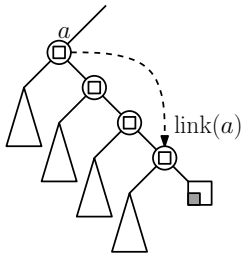

Fig. 9: A link.

In Section 3.1 we discussed rotations. It is easy to modify the rotation process to maintain node weights and links. To see this, let us briefly return to the pseudo-node view of the tree. Consider the operation promote(y), where y is the left child of some node x (recall Fig. 6). Let w be y's outer child. The associated weights wgt(x) and wgt(y) can be updated by assigning each the sum of the weights of their new children. The links are updated as follows. Prior to the rotation, x and y may each have been the head of some shrink chain. Therefore, each may have a link. If y was the first node of a shrink chain prior to the rotation, w is now the head of this chain, and so w is assigned y's old link. If x was the first node of a shrink chain prior to the rotation, y now becomes the first node of the chain after the rotation, and so y is assigned x's old link. The modifications for the promotion of an outer child are symmetrical. (In particular, x is assigned y's old link, and y is assigned w's old link.)

Let us see how the link is used in answering range searching. Given an augmented tree, when the search arrives at the start of a shrink chain and the node is small, rather than descending the chain from top to bottom, as the above search algorithm would, it will traverse the chain from bottom to top. Why does this work? Recall from Section 2 our invariant that, if the cell of a shrink node has an inner box, the shrinking box properly contains this inner box. This implies that, when considered from bottom to top, the sizes of the inner boxes decreases monotonically. This is exactly the property we desire when visiting small nodes.

There is a simple and elegant way in which to implement the modified search algorithm. The algorithm is exactly as described above, but whenever we visit a small shrink node, we "redefine" the meanings of the various tree primitives for each node: left, right, cell, and weight. To motivate this modification, observe that it is possible to transform each shrink chain, into one in which each node in the transformed sequence is the inner child of its parent (see Fig. 10). Let T^* denote the resulting transformed tree.[1] Note that this transformed tree exists merely for conceptual purposes (we do not actually modify the tree).

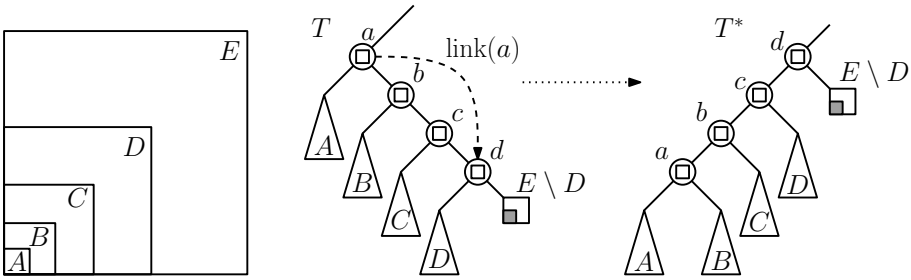

Fig. 10: A shrink chain and the transformed tree.

As mentioned above, this transformation can be achieved "on the fly" by appropriately redefining the primitive tree operations as the range search algorithm is running. The algorithm proceeds as before, but whenever a shrink node is first encountered, we save this node, which is called the *head* of the chain (labeled

[1]The conversion of each shrink chain can be viewed as arising from a series of rotations to the nodes of the shrink chain, but, since we are not considering nodes at the level of pseudo-nodes here, the resulting transformed tree does not satisfy the shrink-split property. Nonetheless, it is equivalent to the original tree from the perspective of the subdivision of space that it defines, and thus it is a valid BD-tree. This is the only property that Procedure range requires.

"a" in Fig. 10). We then immediately follow its link to obtain the next node to be visited by the search. As long as we continue to visit shrink nodes, we apply the alternate definitions of the various tree primitives given in Fig. 11. On returning to a leaf or split node, we return to the standard definitions. Note that the modified weight function requires the use of a subtraction operator.

$$\begin{aligned}
\text{inner}^*(v) &\equiv (v = \textit{head})\ ?\ \text{inner}(v) : \text{parent}(v) \\
\text{outer}^*(v) &\equiv (\text{outer}(v) \text{ is a leaf})\ ?\ \text{outer}(v) : \text{inner}(\text{outer}(v)) \\
\text{cell}^*(v) &\equiv (\text{outer}(v) \text{ is a leaf})\ ?\ \text{cell}(\textit{head}) : \text{cell}(\textit{head}) \setminus \text{cell}(\text{outer}(\text{outer}(v))) \\
\text{wgt}^*(v) &\equiv (\text{outer}(v) \text{ is a leaf})\ ?\ \text{wgt}(\textit{head}) : \text{wgt}(\textit{head}) - \text{wgt}(\text{outer}(\text{outer}(v)))
\end{aligned}$$

Fig. 11: Redefined tree primitives.

For example, in Fig. 10, in the tree T^* the outer child of b is C, which is equivalent to inner(outer(b)) in the original tree T, and this matches the definition of outer$^*(b)$.

Because the search algorithm does not rely on the shrink-split property, establishing correctness of the modified range-search algorithm involves showing that the modified primitives achieve the desired transformation.

Lemma 4.1 *Given a set of points with weights drawn from a commutative group and an augmented BD-tree storing these points (which need not satisfy the shrink-split property), the modified query algorithm correctly returns a count that includes all the points lying inside the inner range and excludes all the points lying outside the outer range.*

The transformed tree T^* satisfies the property that the cell sizes decrease by a constant factor with every constant number of levels of descent. The original tree T is of height h. It follows from a straightforward generalization of the analysis of [3] (using T for the large expanded nodes and T^* for the small expanded nodes) that the query time is $O(h+(1/\varepsilon)^{d-1})$. Because the use of wgt* requires subtraction, this query time applies only if the weights are drawn from a group. In the case of semigroup weights, we perform the entire query in T. In the absence of the size-decreasing property, the query time that results is the product (not the sum) of h and $(1/\varepsilon)^{d-1}$. (Answering an approximate range query can be reduced to $O((1/\varepsilon)^{d-1})$ cells queries in T, each of which takes $O(h)$ time.)

The above query algorithm (for the group case) can be adapted to answer range reporting queries. Whenever we encounter a node whose cell lies entirely inside Q^+, rather than returning the sum of the weights of the points in this node, the algorithm traverses the associated subtree and enumerates all its points.

Summarizing all of these observations, we have the following result, which establishes Theorem 1.1(iii) and (iv).

Theorem 4.1 *Consider a set of n points in $\mathbb{R}^d$ that have been stored in an augmented BD-tree of height h. Then, for any $\varepsilon > 0$ and any convex range Q satisfying the unit-cost assumption:*

(*i*) *If the point weights are drawn from a commutative group, it is possible to answer ε-approximate range counting queries in time $O(h + (1/\varepsilon)^{d-1})$.*

(*ii*) *If the point weights are drawn from a commutative semigroup, it is possible to answer ε-approximate counting queries in time $O(h \cdot (1/\varepsilon)^{d-1})$.*

(*iii*) *It is possible to answer ε-approximate range reporting queries in time $O(h + (1/\varepsilon)^{d-1} + k)$, where k is the number of points reported.*

5 Nearest Neighbor Queries

In this section we briefly outline how to answer approximate nearest neighbor queries in BD-trees. To do so, we assume that the tree is augmented with the link pointers of the previous section. We also assume that each internal node u of the tree stores an arbitrary one of its descendant points, called its *representative*. (A convenient choice is the point that provides the value of $u^{[2]}$.) This point can be maintained throughout our update procedures.

Let T denote the BD-tree, and let h denote its height. Let q denote the query point, and let ε denote the desired approximation factor. Observe that approximate nearest neighbor searching can be reduced to approximate spherical range emptiness queries. By invoking the algorithm of [4] on the BD-tree, it is possible to compute a factor-2 approximation $\widehat{r}$ to the nearest neighbor distance in $O(h)$ time. (There is no need to deal with the transformed tree T^*, since only large nodes are visited.) From this, it is possible to compute an ε-approximate nearest neighbor by employing a bisecting search over the interval $[\widehat{r}/2, \widehat{r}]$. The resulting query time is $O((h + (1/\varepsilon)^{d-1}) \log(1/\varepsilon))$.

We can eliminate the additional factor of $O(\log(1/\varepsilon))$ by employing a more careful search. The idea is to run a type of breadth-first search based on the sizes of the cells involved. Given the value of $\widehat{r}$, it is possible in $O(\log n)$ time to determine a constant number of maximal nodes (depending on dimension) of size at most $2\widehat{r}$ whose cells form a disjoint cover of the ball of radius $\widehat{r}$ centered at q. These form an initial set of *active nodes*. We maintain a variable r, which holds the distance from q to the closest representative so far. Initially, $r = \widehat{r}$. We keep the active nodes in a priority queue, sorted by decreasing cell sizes. We process each active node as follows. First, if its representative is closer to q than r, we update the value of r. If the distance from q to the node's cell is greater than $r/(1 + \varepsilon)$, we discard it. Otherwise, we expand the node by placing its two children in the priority queue of active cells. As in the previous section, all shrink node expansions are performed in the transformed tree T^* (thus, all nodes are treated as if they are small nodes). We apply this until no active nodes remain.

As shown in [4], the search is guaranteed to stop when the sizes of the cells being processed is smaller than $\varepsilon r/\sqrt{d}$, which is as least $\varepsilon\widehat{r}/2\sqrt{d}$. There are $O(\log(1/\varepsilon))$ different quadtree cell size possible in the range from $\widehat{r}$ down to $\varepsilon\widehat{r}/2\sqrt{d}$. We implement the priority queue as an array of lists, which allows us to perform each priority queue operation in $O(1)$ time (with a one-time additive cost of $O(\log(1/\varepsilon))$. Although space does not permit, it can be shown that (even with the dynamically varying radius r) the total number of nodes expanded is $O(1/\varepsilon^{d-1})$. We obtain the following result, which establishes Theorem 1.1(v).

Theorem 5.1 *Consider a set of n points in $\mathbb{R}^d$ that have been stored in an augmented BD-tree of height h. Then, for any $\varepsilon > 0$, it is possible to answer ε-approximate nearest neighbor queries in time $O(h+(1/\varepsilon)^{d-1})$.*

Acknowledgments

We would like to thank Guilherme da Fonseca for suggesting the original motivating application for this work and the reviewers for their many helpful suggestions.

References

[1] A. Andersson. General balanced trees. *J. Algorithms*, 30:1–18, 1999.

[2] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In *Data Compression Conference*, pages 381–390. IEEE Press, 1993.

[3] S. Arya and D. M. Mount. Approximate range searching. *Comput. Geom. Theory Appl.*, 17:135–163, 2001.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *J. Assoc. Comput. Mach.*, 45:891–923, 1998.

[5] M. Bern. Approximate closest-point queries in high dimensions. *Inform. Process. Lett.*, 45:95–99, 1993.

[6] S. N. Bespamyatnikh. Dynamic algorithms for approximate neighbor searching. In *Proc. Eighth Canad. Conf. Comput. Geom.*, pages 252–257, 1996.

[7] T. Chan. A minimalist's implementation of an approximate nearest neighbor algorithm in fixed dimensions. (See `http://www.cs.uwaterloo.ca/- ~tmchan/pub.html`), 2006.

[8] T. M. Chan. Closest-point problems simplified on the ram. In *Proc. 13th Annu. ACM-SIAM Sympos. Discrete Algorithms*, pages 472–473, 2002.

[9] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 226–232, 1983.

[10] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 2005.

[11] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 3rd edition, 2008.

[12] C. A. Duncan. *Balanced Aspect Ratio Trees*. PhD dissertation, Johns Hopkins University, Department of Computer Science, 1999.

[13] C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. *J. Algorithms*, 38:303–333, 2001.

[14] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: A simple dynamic data structure for multidimensional data. In *Proc. 21st Annu. Sympos. Comput. Geom.*, pages 296–305, 2005.

[15] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25:905–910, 1982.

[16] S. Har-Peled. Notes on geometric approximation algorithms. (`http://valis.cs.uiuc.edu/~sariel/-teach/notes/aprx/`), 2009.

[17] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, 1990.

[18] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.

[19] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[20] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Sys. Sci.*, 26:362–391, 1983.