Randomized K-Dimensional Binary Search Trees[*]

Amalia Duch[†] Vladimir Estivill-Castro[‡] Conrado Martínez[†]

September 28, 1998

Abstract

This paper introduces randomized K-dimensional binary search trees (randomized Kd-trees), a variant of K-dimensional binary trees. This data structure allows the efficient maintenance of multidimensional records for any sequence of insertions and deletions; and thus, is fully dynamic. We show that several types of associative queries are efficiently supported by randomized Kd-trees. In particular, a randomized Kd-tree with n records answers exact match queries in expected $\mathcal{O}(\log n)$ time. Partial match queries are answered in expected $\mathcal{O}(n^{1-f(s/K)})$ time, when s out of K attributes are specified, with $0 < f(s/K) < 1$ a real valued function of s/K. Nearest neighbor queries are answered on-line in expected $\mathcal{O}(\log n)$ time. Our randomized algorithms guarantee that their expected time bounds hold irrespective of the order and number of insertions and deletions.

KEYWORDS: Randomized Algorithms, Multidimensional Data Structures, Kd-trees, Associative Queries, Multidimensional Dictionaries.

1 Introduction

Many applications of Computer Science such as Geographical Information Systems (GIS), Databases or Computer Graphics [21] require the dynamic maintenance of a set (or *file*) of *multidimensional records*. Each multidimensional record contains an ordered K-tuple of values that constitute its K-dimensional (or multidimensional) key and associated information. The entries of a K-dimensional key are referred to as *attributes* (or *coordinates*). A file consisting of multidimensional records should support dictionary operations: insertions of new items, deletions, and exact searches, as well as several *associative queries*. An associative query specifies a certain condition to be satisfied by some attributes of the multidimensional keys and requires the retrieval of records in the file with keys satisfying the given condition. Examples of associative queries are intersection queries (which records do lie inside a given region of the search space?) and nearest neighbor queries (which is the record "closest" to the given query point?).

[*] This work is supported by the CONACyT grant # 89422, ESPRIT-LTR Project ALCOM-IT contract # 20244, CICYT TIC97-1475-CE, DGES PB95-0787 (KOALA) and CIRIT 1997SGR-00366 (SGR).

[†] Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, E-08034 Barcelona, Spain. E-mail: {duch,conrado}@lsi.upc.es

[‡] Department of Computer Science and Software Engineering, The University of Newcastle, Callaghan 2308, Australia. E-mail: vlad@cs.newcastle.edu.au

There exist several data structures that support associative queries [10, 21]. Such data structures offer different tradeoffs to the type of associative queries for which they are efficient. Their space requirements, worst-case and expected-case performance on a range of operations as well as their design make them more or less suitable for the dynamic maintenance of a file.

Multidimensional binary search trees, also known as K-dimensional search trees (Kd-trees, from now on) were first introduced by Bentley in 1975 [2] and support all conventional dictionary operations and the most frequent associative queries with relatively simple algorithms, offering reasonable compromises of time and space requirements.

One usual way to analyze the behavior of data structures like Kd-trees is by means of worst-case analysis. However, in many situations, the input sequence that forces the worst-case performance is unlikely. In contrast, expected-case analysis assumes some probability distribution of the input sequence and estimates the expected time of the algorithms over such a distribution. The expected-case analysis of operations for Kd-trees is made under the assumption that the Kd-tree is built by successive insertions of records with K-dimensional keys independently drawn from some continuous distribution [11]. This can be shown to be equivalent to the assumption that the keys are uniformly distributed random K-tuples in $[0,1]^K$. We say that a Kd-tree built in this way is a *randomly built* Kd-tree, or, for brevity, that the Kd-tree is *random*.

For random Kd-trees of size n the expected time for the insertion, the deletion or the exact search of a record is $\mathcal{O}(\log n)$ [2]. Different kinds of associative queries are also supported by this data structure [7, 8, 22]. However, the efficient expected case performance of these operations holds only under the assumption that the Kd-tree is random. Unfortunately, this assumption does not always hold. For instance, it fails when the keys to be inserted are sorted or nearly sorted w. r. t. one of the attributes. Moreover, an alternation of deletions and insertions over a random Kd-tree destroys the randomness of the tree, in the sense that the resulting tree is no longer a random Kd-tree [3, 5]. This happens even if every item of the file is equally likely to be deleted. After some updates (insertions and deletions), the tree may need to be rebuilt to preserve its efficient expected performance.

One possibility to overcome the problem of poor performance is the use of optimized Kd-trees [8, 22], assuming that the file of records is given a priori. Such Kd-trees are perfectly balanced and their logarithmic performance for dictionary operations is guaranteed. However, when insertions and deletions are to be performed, a reorganization of the whole tree is required. Thus, this alternative is not suitable unless updates occur rarely and most records in the file are known in advance, conditions that are not met in many practical situations.

Another approach is to introduce explicit constraints on the balancing of the trees, as in Kd-trees improved by local reorganizations, in dynamically balanced Kd-trees and in divided Kd-trees [4, 18, 23]. All update operations check whether a predefined balance criterion remains true after the insertion or the deletion of an element. If the balance constraint is violated then a complex reorganization of the tree is performed. Although these methods yield theoretically efficient searches (exact search and associative queries) and updates, and in some cases provide worst-case guarantees, these methods sacrifice the simplicity of the standard Kd-tree update algorithms and might be impractical in highly dynamic environments.

A third approach consists in the use of randomization to produce simple yet more robust algorithms, in the sense that the randomization guarantees efficient expected performance that no longer depends on assumptions about the input distribution [15]. Randomization has been successfully applied to the design of dictionaries by Aragon and Seidel [1], Pugh [19] and Martínez and Roura [14]. The approach of Aragon and Seidel has been applied to multidimensional data structures by Mulmuley [16, 17], where randomization is applied to data structures in the field of computational geometry (Voronoi diagrams for instance). However, in Mulmuley's work there is no support neither for dictionary operations nor for associative queries.

In this paper we draw upon the ideas of Martínez and Roura to design randomized Kd-trees, that is, trees built using randomized update algorithms that are guaranteed to always produce random trees, independently of the order of previous insertions and deletions and of the actual data. The main idea is to allow insertions and deletions in regions of Kd-trees other than the leaves, in such a way that the resulting tree is a random tree. As a consequence, the expected case performance of every operation holds irrespective of the order of update operations since it only depends on the random choices made by the randomized algorithms. A generalization of the approach of Aragon and Seidel to the case of Kd-trees is also possible, yielding similar algorithms to the ones in this work, and with similar performances.

Randomized K-dimensional binary search trees (*randomized Kd-trees* , for short), are thus a new type of K-dimensional binary trees that (1) support any sequence of update operations (insertions and deletions) while preserving the randomness of the tree, (2) do not demand preprocessing and (3) efficiently support exact match and associative queries.

A brief review of Kd-trees is presented in Section 2. Then, in order to introduce randomized Kd-trees, we present in Section 3 an extension of Kd-trees that we call *relaxed Kd-trees* Relaxed Kd-trees will provide the necessary flexibility, as the attributes examined along a path down the tree are not restricted to the usual cyclic ordering. We analyze in Section 4 the expected performance of partial match and nearest neighbor queries in random relaxed Kd-trees. Finally, in Section 5 we define randomized update operations over relaxed Kd-trees and we prove that these operations always produce random relaxed Kd-trees.

2 K-Dimensional Search Trees

Multidimensional binary search trees (K-dimensional search trees, Kd-trees) are a generalization of binary search trees to handle the case of multidimensional records.

For the sake of simplicity, in what follows, we identify a multidimensional record with its corresponding multidimensional key $\mathbf{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(K)})$, where each $x^{(j)}$, $1 \leq j \leq K$, refers to the value of the j^{th} attribute of the key $\mathbf{x}$. Each $x^{(j)}$ belongs to some totally ordered domain D_j, and $\mathbf{x}$ is an element of $D = D_1 \times D_2 \times \cdots \times D_K$.

Each multidimensional key may thus be viewed as a point in a K-dimensional space, and its j^{th} attribute can be viewed as the j^{th} coordinate of such a point. Without loss of generality, we assume that $D_j = [0, 1]$ for all $1 \leq j \leq K$, and hence that D is the hypercube $[0, 1]^K$ [2, 11].

Definition 2.1 *A* Kd-tree *for a set of K-dimensional records is a binary tree such that:*

1. *Each node contains a K-dimensional record and has an associated discriminant $j \in \{1, 2, \ldots, K\}$.*

2. *For every node with key* $\mathbf{x}$ *and discriminant* j*, the following invariant is true: any record in the left subtree with key* $\mathbf{y}$ *satisfies* $y^{(j)} < x^{(j)}$ *and any record in the right subtree with key* $\mathbf{y}$ *satisfies* $y^{(j)} > x^{(j)}$*.*

3. *The root node has depth* 0 *and discriminant* 1*. All nodes at depth* d *have discriminant* $(d \bmod K) + 1$.

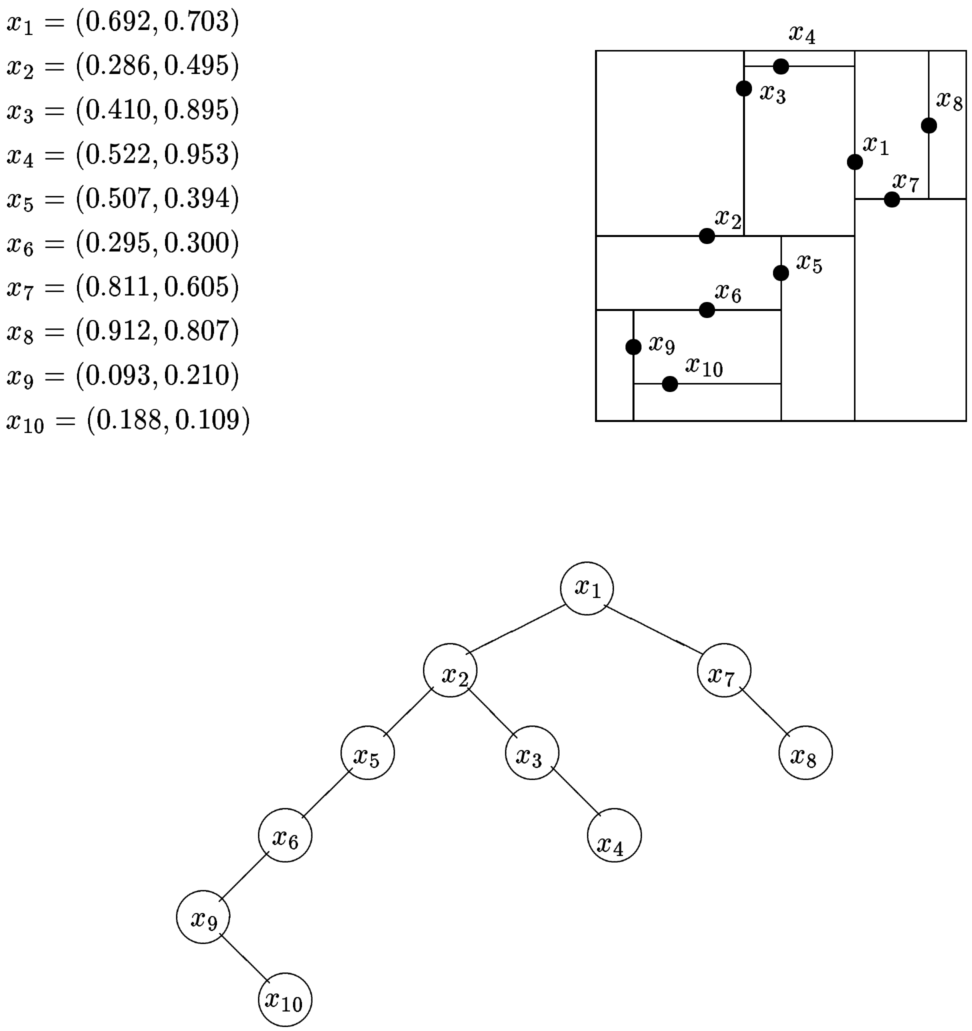
Note that the Definition 2.1 above does not take into account the case of equality of attributes. One usual way to handle this possibility is by means of a *collision list*. In this case, each node contains, in addition to its record $\mathbf{x}$ and its associated discriminant j a list of records with j^{th} attribute equal to $x^{(j)}$.

There are many implementations of Kd-trees. The one suggested by Definition 2.1 corresponds to *homogeneous* Kd-trees, but it is also possible to define *non-homogeneous* Kd-trees. Internal nodes in non-homogeneous Kd-trees contain only one attribute value, and pointers to its left and right subtrees, since all records are stored in external nodes (also known as *buckets*). Observe that in both cases, it is not required to explicitly store a field containing the discriminant of each node, as they are implicitly given by condition 3 of Definition 2.1. In this work, we only deal with homogeneous Kd-trees but, with minor modifications, all the conclusions here apply to non-homogeneous Kd-trees. Note that, if $K = 1$, then a Kd-tree is a binary search tree.

We say that a node contains $(\mathbf{x}, j)$ if its key is $\mathbf{x}$ and its discriminant is j and we say that a key $\mathbf{x}$ is *compatible* with a tree T if, for any j, its j^{th} attribute is different from the j^{th} attribute of any other key already in T. We assume that only insertions of compatible keys ever occur in Kd-trees and hence that any key to be inserted is different from any key already present in the tree (see the discussion above about collision lists).

A Kd-tree for a file F can be incrementally built by successive insertions into an initially empty Kd-tree as follows. The first key is put into a single node with two empty subtrees. The first attribute of the second key is compared then with the first attribute of the key at the root: if it is smaller, the second key is recursively inserted in the (empty) left subtree; otherwise, it is recursively inserted in the (empty) right subtree. Then the first attribute of the third key is compared with the first attribute of the key at the root, and recursively inserted into the left or right subtree, as before. But if that subtree were not empty because it contained the second key, we would compare the respective second attributes of the second and third keys and proceed as before. In general, when inserting a key $\mathbf{x}$, we compare the key to be inserted with some key $\mathbf{y}$ at the root of some subtree: if $\mathbf{y}$ is at level j, we compare $x^{((j \bmod K)+1)}$ and $y^{((j \bmod K)+1)}$, and recursively continue the insertion in the left or the right subtree of $\mathbf{y}$, until a leaf (empty subtree) is found —recall that we assume that the key $\mathbf{x}$ is compatible with the tree where it is being inserted. In Figure 1 we depict the 2d-tree that results from the insertion of ten keys into an initially empty tree, in the same order as they have been listed in the left part of the figure. The figure also shows the partition of $[0, 1]^2$ induced by the 2d-tree. It should be clear that if the same points were inserted in a different order they could yield a substantially different 2d-tree.

The standard model for the probabilistic analysis of Kd-trees is that a *randomly built* K*d-tree* or *random* K*d-tree* of size n is built by inserting n points independently drawn from some continuous probability distribution defined over $[0, 1]^K$. This is equivalent to assume that the



$x_1 = (0.692, 0.703)$

$x_2 = (0.286, 0.495)$

$x_3 = (0.410, 0.895)$

$x_4 = (0.522, 0.953)$

$x_5 = (0.507, 0.394)$

$x_6 = (0.295, 0.300)$

$x_7 = (0.811, 0.605)$

$x_8 = (0.912, 0.807)$

$x_9 = (0.093, 0.210)$

$x_{10} = (0.188, 0.109)$

Figure 1: A $2d$-tree and its corresponding partition of the search space

probability that the $(n+1)^{\text{th}}$ insertion fails in a given leaf of a random Kd-tree of size n is the same for any of its $n+1$ leaves.

The expected cost of a single insertion in a random Kd-tree is $\mathcal{O}(\log n)$ time while the expected cost of building the whole tree is $\mathcal{O}(n \log n)$ [2]. The cost of deleting the root of a random Kd-tree with n nodes is $\mathcal{O}(n^{1-\frac{1}{K}})$. But, on the average, deletions have expected cost $\mathcal{O}(\log n)$ since the expected depth of the node to be deleted is logarithmic and the subtree beneath it has expected size $\mathcal{O}(\log n)$ [2].

Exact match queries in Kd-tree obviously operate by following a path down the tree, exactly the same way as if we were inserting that key: either we find it and report success or we reach a leaf, reporting failure. Therefore, they also require expected $O(\log n)$ time. It has to be pointed out here that if the application only required exact match queries, Kd-trees should not be used, since treating the attributes of each record in lexicographical order and using unidimensional data structures would usually yield better performance.

It has been shown by Flajolet and Puech [7] that partial match queries are efficiently supported by random Kd-trees. The expected time of this operation in random Kd-trees is $\mathcal{O}\left(n^{1-\frac{s}{K}+\theta\left(\frac{s}{K}\right)}\right)$, where $|\theta(s/K)| \leq 0.07$.

It has also been shown by Friedman, Bentley and Finkel [8] that nearest neighbor queries are supported in $\mathcal{O}(\log n)$ time in optimized Kd-trees. They use a simplified model and make a few assumptions to do the analysis; using this simplified model and the same assumptions it turns out that nearest neighbor queries can also be answered in expected time $\mathcal{O}(\log n)$ in random Kd-trees. Furthermore, these theoretical although somewhat approximate results are strongly supported by the experiments that we have carried out.

3 Relaxed K-Dimensional Search Trees

Observe that the cyclic restriction of Kd-trees (condition 3 of Definition 2.1) forces update operations to be very laborious or located at the leaves of the tree, since a reorganization of the whole tree would be required in order to preserve the cyclic assignment of discriminants. But, as we shall show later, in our quest for randomized Kd-trees, we require the flexibility of performing operations in places of Kd-trees other than the leaves without major reorganization.

With this purpose in mind, we introduce *relaxed K-dimensional search trees* (also called *relaxed Kd-trees* , or simply r-Kd-trees). These are Kd-trees where condition 3 of Definition 2.1 is dropped and where each node explicitly stores its discriminant instead. So, the sequence of discriminants in a path from the root to any leaf is arbitrary.

Definition 3.1 *A* r-Kd-tree *for a set of K-dimensional records is a binary tree in which:*

1. *Each node contains a K-dimensional record and has associated a discriminant $j \in \{1, 2, \ldots, K\}$.*
2. *For every node with key $\mathbf{x}$ and discriminant j, the following invariant is true: any record in the right subtree with key $\mathbf{y}$ satisfies $y^{(j)} < x^{(j)}$ and any record in the left subtree with key $\mathbf{y}$ satisfies $y^{(j)} > x^{(j)}$.*

Notice that a r-1d-tree is a binary search tree. Moreover, as for Kd-trees, it is assumed that the attribute domains are continuous sets [2, 11]. Figure 2 shows one relaxed 2d-tree that results from the insertion of ten keys into an initially empty tree. Each node contains its corresponding (key, discriminant) pair. The figure also shows the partition of $[0,1]^2$ induced by the relaxed 2d-tree. It should be clear that if different discriminants had been chosen or if the same keys were inserted in a different order the relaxed 2d-tree could be substantially different.

We say that a r-Kd-tree of size n is *randomly built* or *random* if it is built by n insertions where the keys are independently drawn from some continuous distribution (for example, uniformly from $[0,1]^K$) and the discriminants are uniformly and independently drawn from $\{1,\ldots,K\}$.

In random r-Kd-trees, this assumption about the distribution of the input implies that the $n!^K K^n$ distinct configurations of input and discriminant sequences are equally likely (observe that for random Kd-trees the assumption is that all $n!^K$ distinct configurations of the input sequence are equallylikely) [7, 2, 11].

In particular, in a random r-Kd-tree each of the nK possible pairs (key, discriminant) are equally likely to appear in the root and once the root is fixed, the left and right subtrees are independent random r-Kd-trees. Thus, we obtain the same distribution for the shapes of random r-Kd-trees as for the shapes of random binary search trees and the shapes of random Kd-trees. Moreover, the distribution of random r-Kd-trees of size n can be nicely characterized in terms of itself, and the probability that the left subtree of a random r-Kd-tree of size n has size ℓ is $1/n$, for $0 \leq \ell < n$, independently of K.

Definition 3.2 *T is a random r-Kd-tree if and only if*

1. *$|T| = 0$ (i.e. T is empty) or,*
2. *$|T| = n$, its left and right subtrees are independent random r-Kd-trees, and for any $\mathbf{x} \in T$ and any discriminant j, $1 \leq j \leq K$,*

$$\mathbb{P}\{(\mathbf{x}, j) \text{ is the root of } T\} = \frac{1}{nK}.$$

Random Kd-trees satisfy a similar (at least in spirit) but more complex recursive characterization, as the discriminants in successive levels must follow the cyclic sequence: $1, 2, \ldots, K, 1, 2 \ldots$

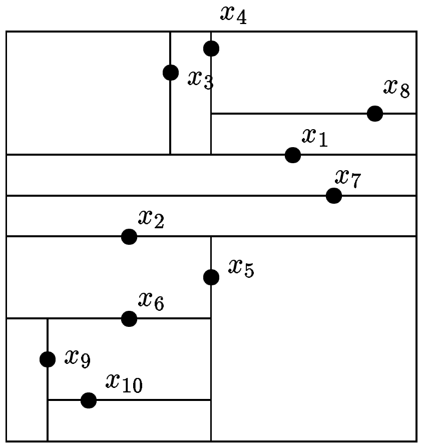
As we shall see the recursive definition above for random r-Kd-trees makes the analysis of partial match and other associative queries in r-Kd-trees much easier than in standard Kd-trees.

Parameters like the internal or the external path length (see [10, 11] for definitions) do not depend on the discriminants and hence their expected value is the same for random binary search trees and random r-Kd-trees.

Theorem 3.1 [10] *The expected internal and external path length of a random r-Kd-tree is $\mathcal{O}(n \log n)$.*

Therefore, random r-Kd-trees perform like random Kd-trees for exact match queries and insertions. These operations explore a path that, because of Theorem 3.1, is of logarithmic length, on the average.

$$x_1 = (0.692, 0.703)$$
$$x_2 = (0.286, 0.495)$$
$$x_3 = (0.410, 0.895)$$
$$x_4 = (0.522, 0.953)$$
$$x_5 = (0.507, 0.394)$$
$$x_6 = (0.295, 0.300)$$
$$x_7 = (0.811, 0.605)$$
$$x_8 = (0.912, 0.807)$$
$$x_9 = (0.093, 0.210)$$
$$x_{10} = (0.188, 0.109)$$



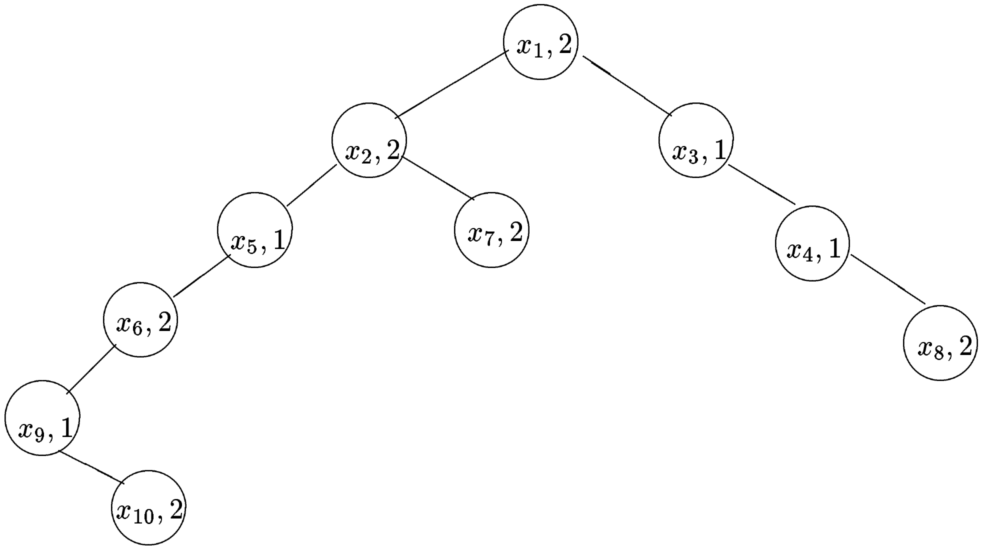


Figure 2: A relaxed 2d-tree and its corresponding partition of the search space.

4 Associative Retrieval

The algorithms for associative queries in r-Kd-trees are essentially the same as for Kd-trees but their expected-case performances are not necessarily the same (for instance, partial match queries have not the same average complexity in Kd-trees and r-Kd-trees) since for random r-Kd-trees a path down the tree does not have a known cyclic pattern of discriminants but a random sequence of discriminants. In what follows, we analyze the average cost of performing partial match and nearest neighbor queries in r-Kd-trees.

4.1 Partial Match Queries

A *partial match query* is a query in which only s out of the K attributes of a given key are specified (with $0 < s < K$). The retrieval consists of finding all the records whose specified key attributes coincide with those of the given query.

More formally, given a query $q = (q_1, q_2, \ldots, q_K)$ where each q_j is either a value in D_j (specified) or $q_j = *$ (unspecified), the partial match algorithm returns the subset of records in the file F whose attributes coincide with the specified attributes of q, i.e. returns $\mathbf{x}$ if and only if, for any j, $q_j = *$ or $q_j = x^{(j)}$.

The partial match algorithm in a r-Kd-tree explores the tree in the following way. At each node it examines the corresponding discriminant. If that discriminant is specified in the query (which happens with probability s/K), then the algorithm recursively follows in the appropriate subtree, depending on the result of the comparison between the attribute of the key at the current node and the attribute of the query. Otherwise (with probability $1 - s/K$), the algorithm recursively follows the two subtrees.

The following theorem gives the expected performance of a partial match query in a random r-Kd-tree of size n.

Theorem 4.1 *The expected cost C_n (measured as the number of comparisons) of a partial match query with s out of K attributes specified in a random r-Kd-tree of size n is*

$$C_n = \beta n^{\delta} + \mathcal{O}(1),$$

where

$$\begin{aligned} \delta &= \delta(s/K) = -\frac{1}{2} + \frac{\Delta(s/K)}{2} \sim_{s/K \to 0} 1 - \frac{2}{3}\left(\frac{s}{K}\right) + \mathcal{O}\left(\left(\frac{s}{K}\right)^2\right), \\ \beta &= \frac{\Gamma(2\delta + 1)}{(1 - s/K)(\delta + 1)\Gamma^3(\delta + 1)}, \end{aligned}$$

with $\Delta(x) = \sqrt{9 - 8x}$ and $\Gamma(x)$ the Gamma function [9].

Proof. Let T be a random r-Kd-tree of size $n > 0$ with left subtree L and right subtree R, and let $C_n = C(T)$ be the average search cost of a partial match. Then, with probability $\frac{K-s}{K}$, the discriminant in the root of T corresponds to an unspecified attribute of the query. In this case, the search visits the root and then continues in both L and R. Otherwise, the discriminant in the root of T corresponds to some specified attribute, and then after visiting the root, the partial match retrieval continues into the appropriate subtree. It will continue in L with probability $\frac{\ell+1}{n+1}$ (where ℓ is the size of L) and in R with the complementary probability. Thus, the average search cost satisfies the relation

$$C(T||L| = l) = \frac{K-s}{K}\left(1 + C(L) + C(R)\right) + \frac{s}{K}\left(1 + \frac{\ell+1}{n+1}C(L) + \frac{n-\ell}{n+1}C(R)\right).$$

Unconditioning, since $\mathbb{P}\left\{|L| = \ell\right\} = 1/n$ for all ℓ, $0 \le \ell < n$ (because T is assumed to be random) we find that, for $n > 0$,

$$C_n = 1 + \frac{K-s}{K}\left(\frac{1}{n}\sum_{\ell=0}^{n-1}[C_\ell + C_{n-1-\ell}]\right) + \frac{s}{K}\left(\frac{1}{n}\sum_{\ell=0}^{n-1}\left[\frac{\ell+1}{n+1}C_\ell + \frac{n-\ell}{n+1}C_{n-1-\ell}\right]\right),$$

and by symmetry we obtain the recurrence relation

$$C_n = 1 + 2\left(\frac{K-s}{K}\right)\frac{1}{n}\sum_{\ell=0}^{n-1}C_\ell + 2\left(\frac{s}{K}\right)\frac{1}{n}\sum_{\ell=0}^{n-1}\frac{\ell+1}{n+1}C_\ell. \qquad (1)$$

Using the continuous master theorem by Roura [20] it is easy to derive an asymptotic estimate for C_n, namely, $C_n = \Theta(n^\delta)$, where $\delta = (\sqrt{9 - 8\frac{s}{K}} - 1)/2$.

The same result can be obtained using standard techniques such as generating function and singularity analysis [6]. The ordinary generating function of the sequence $\{C_n\}_{n\ge0}$, is $C(z) = \sum_{n\ge0} C_n z^n$, with $C(0) = 0$. We also find useful to define the generating function $D(z) = \sum_{n\ge0}(n+1)C_n z^n = zC'(z) + C(z)$, $D(0) = 0$. Multiplying Equation (1) by $(n+1)$ gives

$$(n+1)C_n = n + 1 + 2\left(\frac{K-s}{K}\right)\frac{n+1}{n}\sum_{\ell=0}^{n-1}C_\ell + 2\left(\frac{s}{K}\right)\frac{1}{n}\sum_{\ell=0}^{n-1}(\ell+1)C_\ell.$$

This recurrence translates to the following integral equation

$$D(z) = \frac{1}{(1-z)^2} - 1 + 2\frac{K-s}{K}\left(\frac{C(z)}{1-z} - C(z) + \int_0^z \frac{C(t)}{1-t}dt\right) + 2\frac{s}{K}\int_0^z \frac{D(t)}{1-t}dt.$$

Taking derivatives and expressing $D(z)$ in terms of $C(z)$ gives the second order non-homogeneous differential equation

$$C''(z) - 2\frac{(2z-1)C'(z)}{z(1-z)} - 2\frac{(2 - s/K - z)C(z)}{z(1-z)^2} - 2\frac{1}{z(1-z)^3} = 0. \qquad (2)$$

The homogeneous differential equation associated to Equation (2) has only z and $(1-z)^p$ as divisors for $p = 1, 2$. Thus, $C(z)$ has a single singularity at $z = 1$ and because p is integer, the function is meromorphic with a single pole at $z = 1$. Thus, the dominant contribution in the local expansion of $C(z)$, when $z \to 1$, is of the form $C(z) \sim \beta(1-z)^\alpha$, with α the smallest root of the indicial equation: $\alpha^2 + \alpha - 2(1 - s/K) = 0$, which results in $\alpha = \frac{-1-\Delta(s/K)}{2}$, with $\Delta(x) = \sqrt{9 - 8x}$. Standard singularity analysis [6], shows that

$$C_n = [z^n]C(z) \sim \beta n^\delta$$

for $\delta = -\alpha - 1$ and some constant β. Expanding $\delta(s/K)$ in Taylor series, we obtain the estimate $\delta(s/K) \sim 1 - 2/3(s/K) + \mathcal{O}((s/K)^2)$ given in the statement of the theorem.

The approach that we have sketched above for the analysis of partial match can be further explored. In [13] it has been shown that the exact solution of Equation (2) is

$$C(z) = \frac{1}{1-s/K}\left({}_2F_1\left(\left. \begin{matrix} a, b \\ 2 \end{matrix} \right| z \right) (1-z)^{\alpha} - \frac{1}{1-z} \right), \tag{3}$$

where ${}_2F_1\left(\left. \begin{matrix} a,b \\ c \end{matrix} \right| z \right)$ is the hypergeometric function [9], and $a = 2+\alpha$ and $b = 1+\alpha$. Then, we can study the asymptotic behavior of $C(z)$ when $z \to 1$ to get not only the precise order of magnitude of C_n but the coefficient of the main order term and the magnitude of the lower order terms. The second term of $C(z)$ makes a contribution which is $\mathcal{O}(1)$ and the hypergeometric function is analytic at $z = 1$. Therefore,

$$\beta = \frac{{}_2F_1\left(\left. \begin{matrix} a,b \\ 2 \end{matrix} \right| 1 \right)}{(1-s/K)\Gamma(-\alpha)} = \frac{\Gamma(2\delta+1)}{(1-s/K)(\delta+1)\Gamma^3(\delta+1)}.$$

□

It is interesting to point out that, although Theorem 3.1 is valid only if $0 < \frac{s}{K} < 1$, it provides meaningful information for the limiting cases—at least, to some extent. If $\frac{s}{K} \to 0$, that is, no attribute is specified, then $C_n = n$. Indeed, $\delta \to 1$ and $\beta \to 1$ as $\frac{s}{K} \to 0$. On the other hand, in an exact match all attributes are specified and $s = K$. In this case, we know that $C_n = \mathcal{O}(\log n)$. And we have that $\delta \to 0$ and $\beta \to \infty$ if $\frac{s}{K} \to 1$, which is the best approximate way to say with a formula like βn^{δ} that C_n grows with n, but slower than any function of the type n^{ϵ}, for real positive ϵ.

In Figures 3 and 4 we plot respectively the value of the exponent δ in the average cost of partial match queries in random Kd-trees and random r-Kd-trees, and the excess of the corresponding exponents with respect to $1 - \frac{s}{K}$ ($\Theta(n^{1-s/K})$ is the best known upper bound for partial match queries). In Figure 5 we plot the value of β as a function of the ratio $\rho = s/K$.

Observe that the expected cost of partial match queries in random r-Kd-trees is slightly higher than the one given by Flajolet and Puech [7] for random Kd-trees. It is possible to show that the difference in the exponent of n of these costs is at most 0.08, though —and for extreme values of s/K the difference is much smaller. Notice also that the constant β in the main order term of the expected cost of partial match queries in r-Kd-trees is independent of the pattern of specified/unspecified attributes, whereas for standard Kd-trees such a constant is dependent on the particular pattern of the query [7].

4.2 Nearest Neighbor Queries

A request for the closest record in a file to a given query record, under a determined distance function (usually called dissimilarity function), is called a *nearest neighbor* query. More generally, given a multidimensional record $\mathbf{q}$, a distance d over D, and a r-Kd-tree T of n records, an m-nearest neighbor query requires finding the m points in T closest to $\mathbf{q}$; in order words, to find the m points $\mathbf{y}$ in T such that $d(\mathbf{q}, \mathbf{y}) \leq d(\mathbf{q}, \mathbf{x})$, for all $\mathbf{x}$ but at most $m-1$ of the records in T. The nearest neighbor query is the case $m = 1$.

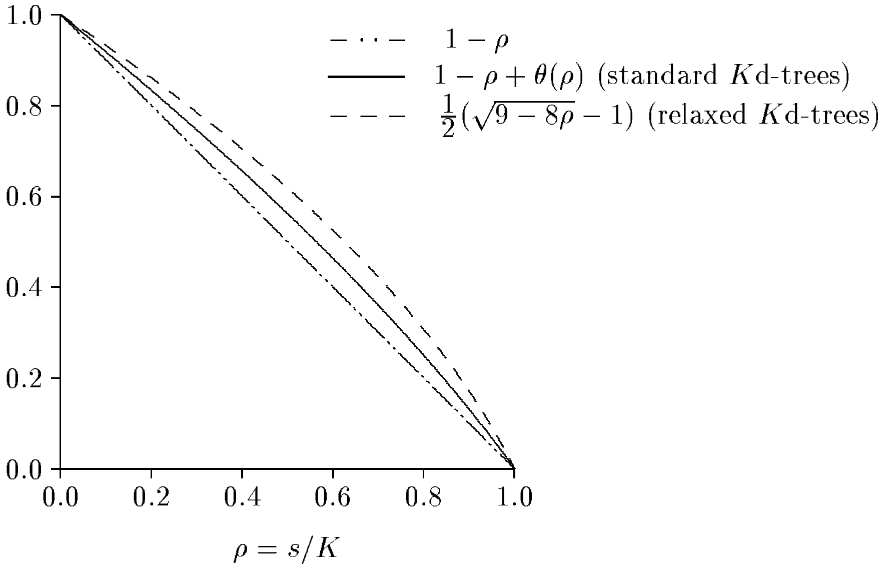


Figure 3: The value of the exponent in the average cost of partial match in standard and relaxed Kd-trees.

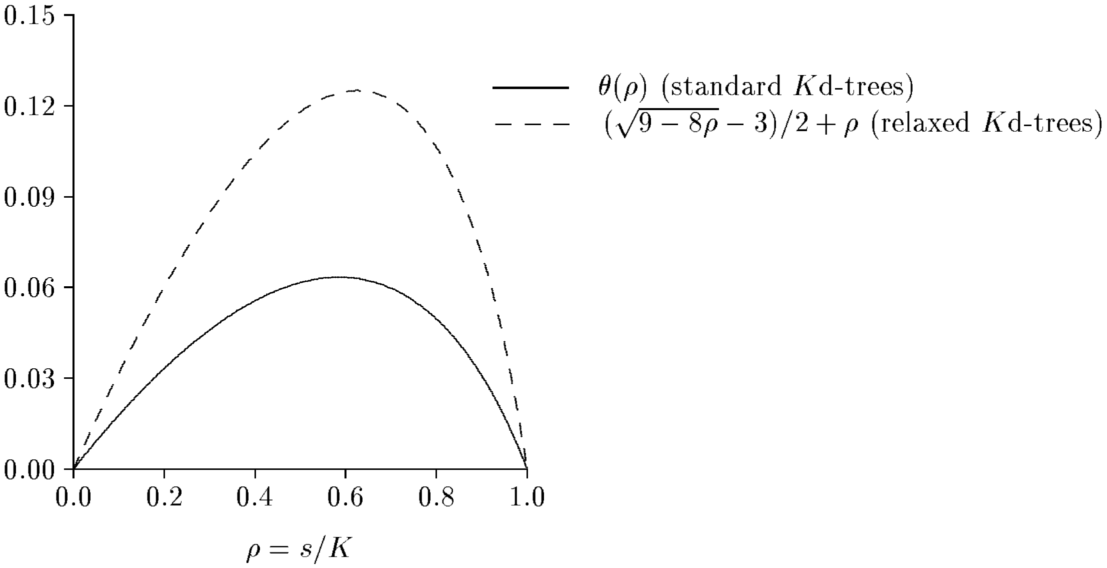


Figure 4: Excess (with respect to $1-\rho$) of the exponent in the average cost of partial match

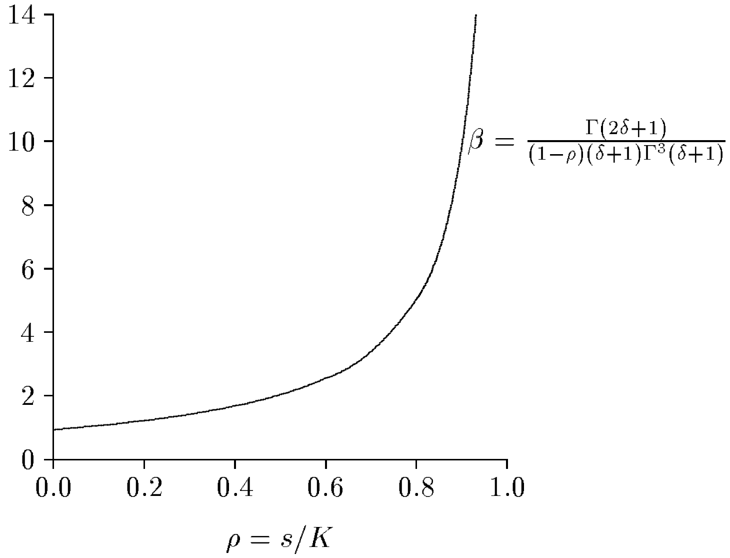


Figure 5: The value of the constant β in the average cost of partial match in relaxed Kd-trees.

The nearest neighbor search algorithm over a r-Kd-tree begins by calculating the distance between the query and the root of the tree, giving a current nearest neighbor. The search continues along either the left or the right subtree according to whether the query record lies on the left or right side of the root, until an empty subtree is reached. At each level of the recursion, we calculate the distance between the query and the root of the current subtree. If the new distance is smaller than the previous one, then, the current nearest neighbor is updated. At the end of the recursion, we have a current nearest neighbor $\mathbf{y}$. If there is a record in the tree, that is closer to the query than $\mathbf{y}$, then, it must lie within the region Q centered at $\mathbf{q}$ with radius $d(\mathbf{q}, \mathbf{y})$. Remember that, in geometric terms, a r-Kd-tree induces a partition of the K-dimensional space in which the file of records lies. So, if the region Q is totally included in the cell of the partition corresponding to $\mathbf{q}$, then, the search is finished at this point and the nearest-neighbor is the record $\mathbf{y}$. Otherwise, all the cells intersected by the region Q must be explored.

Theorem 4.2 *The expected cost (measured as the number of visited nodes) of a nearest neighbor query in a random r-Kd-tree of size n is $\mathcal{O}(\log n)$.*

Proof. The proof is similar to the proof given by Friedman, Bentley and Finkel [8]. It is only required to observe that the assumptions and the simplified model used there for the analysis may also be applied to study nearest neighbor queries in r-Kd-trees. In particular, the expected volume of a cell in the partition induced by a random r-Kd-tree is $1/(n+1)$ and that the expected side-length of any cell along any coordinate is $1/(n+1)^{1/K}$, as was the case for optimized Kd-trees. Observe, also, that to explore a cell corresponds to follow a path of the tree, thus, the average case analysis of the nearest neighbor search algorithm depends on the expected number of cells intersected by the minimum region centered at the query $\mathbf{q}$ and with radius equal to the distance from $\mathbf{q}$ to the current nearest neighbor. Since this number does not depend on the size of the tree [8], the average expected cost of nearest neighbor queries is proportional to $\log n$. □

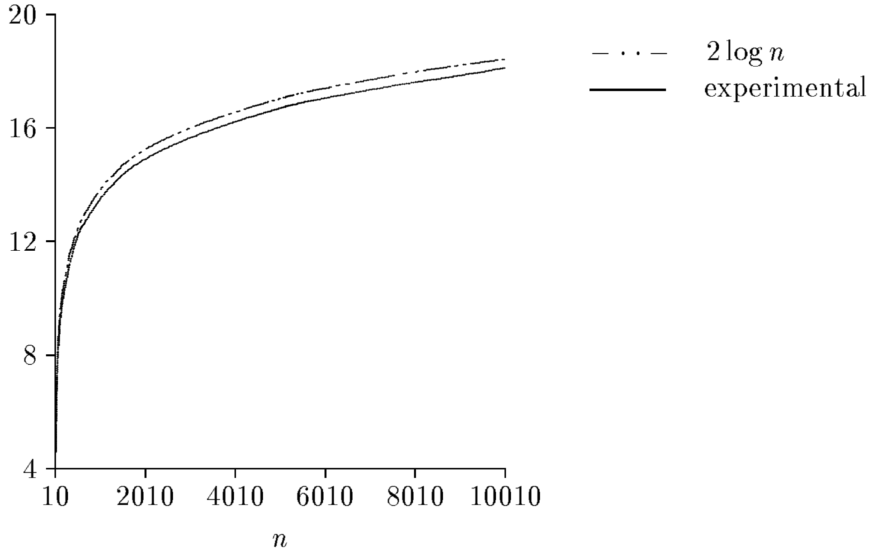


Figure 6: The average cost of nearest neighbor in relaxed Kd-trees.

Experimentally, we obtain that the average cost of nearest neighbor queries under the Euclidean distance is approximately $2\log n$ independent of the dimension. Our experiments consisted on building r-Kd-trees of sizes going from $n = 10$ to $n = 10000$ records (we generated 10000 trees of each size), and in generating one thousand nearest neighbor queries for each generated tree. The experiments were repeated for dimensions $K = 1$ to $K = 25$, but no significant variation was observed from one dimension to another. The graph of Figure 6 summarizes the results of our experiments. It shows a good match with the theoretical predictions: the average path length if $2\log n + \mathcal{O}(1)$ in random binary search trees and in random r-Kd-trees, and the nearest neighbor cost is basically: 1) following a path stopping at some leaf; 2) exploring bottom-up a small subtree that contains that leaf.

It is also interesting (on-going work) to obtain experimental results for other metric distances as well as to empirically study the variance of these costs. Preliminary results show that for the Euclidean distance the value of this variance is approximately $\mathcal{O}(\log^2 n)$.

5 Randomized Kd-trees

Relaxed Kd-trees have been shown to efficiently support dictionary operations and associative queries in the expected-case. However, this expected performance is based upon the assumption that the input sequences of keys from which the tree is built corresponds to a random permutation (that is, the $n!^K$ possible configurations for the input sequence are equally likely). In what follows we will guarantee the expected performance, irrespective of any assumption about the input sequence[1], using randomized algorithms. The randomized algorithms presented here require that each node stores the size of the subtree beneath it [14],

[1] Except that we assume that all keys are compatible or equivalently, that the probability of two keys having the same value for the i^{th} attribute, $i = 1, \dots, K$, is insignificantly small.

because the behavior of our randomized operations depends on the size of the (sub)trees to which they are applied.

We say that a r-Kd-tree is a *randomized Kd-tree* if it is the result of a sequence of update operations performed by means of the randomized algorithms introduced below, applied to an initially empty tree. We shall show in this section that any tree obtained this way is a random r-Kd-tree.

Without loss of generality, we assume in what follows that randomized algorithms have free access to a source of random bits and that the cost of generating a random number of $\mathcal{O}(\log n)$ bits is constant [15].

5.1 Randomized Insertions and Deletions

Informally, in order to produce a random r-Kd-tree two properties must hold (because of Definition 3.2):

1. A new inserted key should have some probability of becoming the root of the tree, or the root of one of the subtrees of the root, and so forth, and,
2. Every discriminant should have the same probability of being the discriminant of the new node.

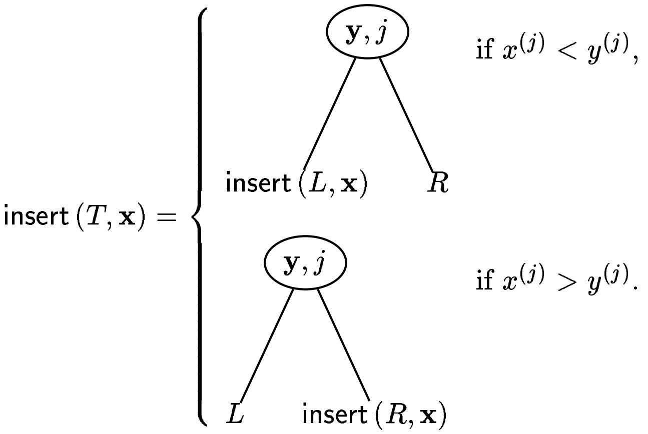
The insertion of $\mathbf{x}$ in a r-Kd-tree T, begins by generating uniformly a random integer, say i, from the set $\{1, \ldots, K\}$. This step corresponds to the assignment of a discriminant to the new node. Once the discriminant is assigned the insertion algorithm proceeds as follows.

1. If the tree T is the empty tree, then the algorithm insert produces a tree with root node $(\mathbf{x}, i)$ and empty left and right subtrees.
2. If the tree T is not empty, then, with probability $\frac{1}{n+1}$ the (key, discriminant) pair $(\mathbf{x}, i)$ must be placed at the root of the new tree using the $\mathsf{insert\_at\_root}$ algorithm (since the new tree will have size $n+1$). Otherwise, we insert the pair $(\mathbf{x}, i)$ recursively in either the left or the right subtree of T depending on $\mathbf{x}$'s order relation with the root of T.

Let T have root $(\mathbf{y}, j)$, left subtree L and right subtree R. Symbolically we have,

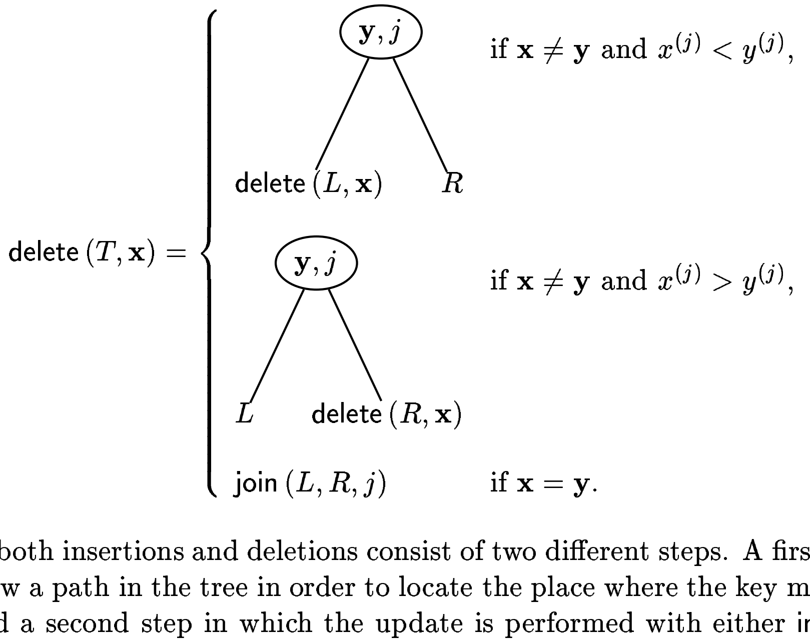
1. With probability $\frac{1}{n+1}$, $\mathsf{insert}\,(T, \mathbf{x}) = \mathsf{insert\_at\_root}\,(T, \mathbf{x}, i)$, and,

2. With probability $\frac{n}{n+1}$,



The algorithm insert requires the possibility of inserting the pair $(\mathbf{x}, i)$ at the root of any subtree of a r-Kd-tree T. If T is empty, then, insert_at_root $(T, \mathbf{x}, i)$ gives as a result a tree with root node $\mathbf{x}$, discriminant i and empty left and right subtrees. When T is not empty, by definition, insert_at_root $(T, \mathbf{x}, i) = T'$ where the root of T' is $(\mathbf{x}, i)$, its left subtree consists of all those elements of T with i^{th} attribute smaller than $x^{(i)}$ and its right subtree contains those elements of T with i^{th} attribute than $x^{(i)}$. To obtain the left and right subtrees of T' we use the split algorithm which we present later.

The deletion of a record from a random r-Kd-tree consists of searching in the tree for the key $\mathbf{x}$ to be deleted and then to join its corresponding left and right subtrees. In order to keep a random r-Kd-tree after the deletion, all the nodes in the two subtrees of the deleted node should have some probability of taking the place of the deleted node. This is achieved by the join algorithm (which we introduce later). Let $\mathbf{x}$ be the record to be deleted from the random r-Kd-tree T. Let T have root $(\mathbf{y}, j)$, left subtree L and right subtree R. Symbolically,


delete $(T, \mathbf{x}) =$
$\mathbf{y}, j$
if $\mathbf{x} \neq \mathbf{y}$ and $x^{(j)} < y^{(j)}$,
delete $(L, \mathbf{x})$
R
$\mathbf{y}, j$
if $\mathbf{x} \neq \mathbf{y}$ and $x^{(j)} > y^{(j)}$,
L
delete $(R, \mathbf{x})$
join (L, R, j)
if $\mathbf{x} = \mathbf{y}$.

Observe that both insertions and deletions consist of two different steps. A first step in which one must follow a path in the tree in order to locate the place where the key must be inserted or deleted and a second step in which the update is performed with either insert_at_root or join.

The expected cost of insertions and deletions is $\mathcal{O}(\log n)$, since both the insertion at the root of any element and the deletion of any element occur near the leaves on the average. In other words, the reconstruction step, although expensive —but less expensive than in standard Kd-trees— occurs in subtrees of expected size $\mathcal{O}(\log n)$ [2, 12].

5.2 The Split and Join Algorithms

The insertion of a pair $(\mathbf{x}, i)$ at the root of a tree T (the task that $\mathsf{insert\_at\_root}\,(T, \mathbf{x}, i)$ performs) is accomplished in two phases. First, the tree T is partitioned with respect to the i^{th} attribute of $\mathbf{x}$ to produce two trees, $T_{<_i}$ and $T_{>_i}$, that contains all keys of T whose i^{th} attribute is smaller than $x^{(i)}$ and all the keys of T whose attribute is greater than $x^{(i)}$, respectively. Second, the two trees of the previous step are attached to the root $(\mathbf{x}, i)$. But the main bulk of $\mathsf{insert\_at\_root}\,(T, \mathbf{x}, i)$ clearly lies in the partitioning or splitting $(T_{<_i}, T_{>_i}) = \mathsf{split}(T, \mathbf{x}, i)$. To simplify the description of split, we will see it as pair of functions $\mathsf{split}_<$ and $\mathsf{split}_>$, with $T_{<_i} = \mathsf{split}_<\,(T, \mathbf{x}, i)$ and $T_{>_i} = \mathsf{split}_>\,(T, \mathbf{x}, i)$. In practice, both $T_{<_i}$ and $T_{>_i}$ can be simultaneously computed.

The algorithm $\mathsf{split}_<$ works in the following way. When T is the empty tree, $\mathsf{split}_<$ returns the empty tree. Otherwise, let T have root $(\mathbf{y}, j)$, left subtree L, and right subtree R. We have four cases to consider:

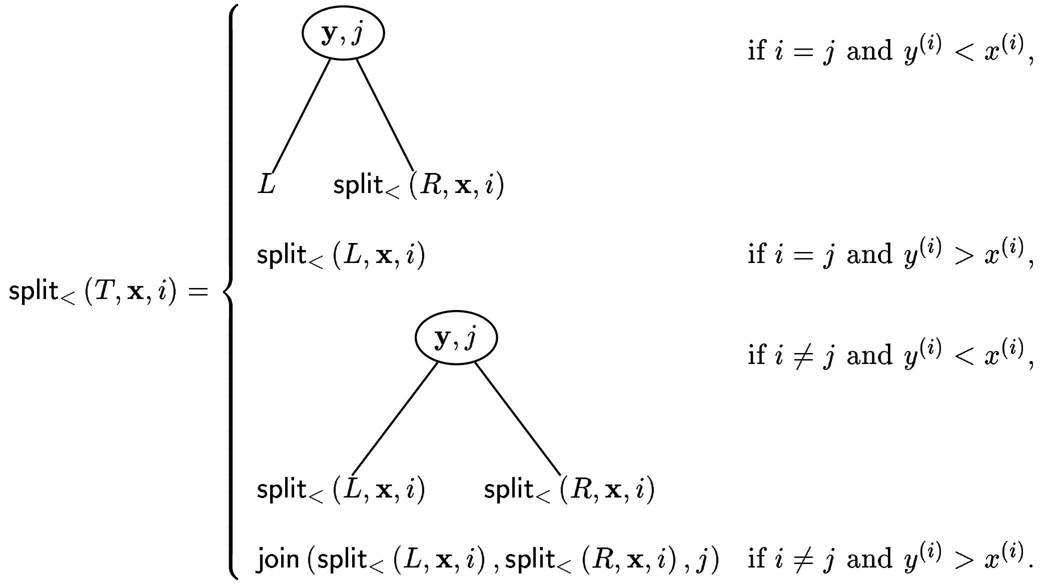
1. If $i = j$ and $y^{(i)} < x^{(i)}$, then $\mathbf{y}$ belongs to $T_{<_i}$, all the elements of L do as well (since each i^{th} attribute value in L is smaller than $y^{(i)}$ and thus smaller than $x^{(i)}$) and the operation proceeds recursively in R to complete the result;

2. If $i = j$ and $y^{(i)} > x^{(i)}$, then the algorithm proceeds recursively in L (since $\mathbf{y}$ and all the i^{th} attributes in R are greater than $x^{(i)}$);

3. If $i \neq j$ and $y^{(i)} < x^{(i)}$, then $\mathbf{y}$ belongs to $T_{<_i}$ and the algorithm proceeds recursively in L and in R and attach to $\mathbf{y}$ the results of splitting L and R;

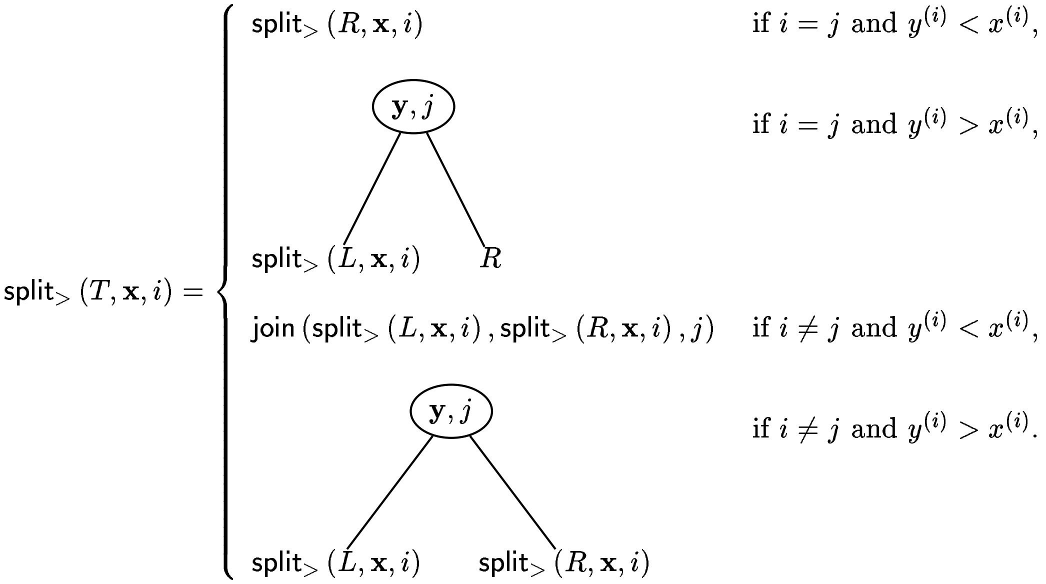
4. If $i \neq j$ and $y^{(i)} > x^{(i)}$, then $\mathbf{y}$ is not in the tree and the algorithm must proceed recursively in L and R, but now it has to join the trees resulting after the split of L and R.

Symbolically, if T is not empty,



$\mathsf{split}_<(T, \mathbf{x}, i) =$
$\mathbf{y}, j$
L
$\mathsf{split}_<(R, \mathbf{x}, i)$
if $i = j$ and $y^{(i)} < x^{(i)}$,
$\mathsf{split}_<(L, \mathbf{x}, i)$
if $i = j$ and $y^{(i)} > x^{(i)}$,
$\mathbf{y}, j$
$\mathsf{split}_<(L, \mathbf{x}, i)$
$\mathsf{split}_<(R, \mathbf{x}, i)$
if $i \neq j$ and $y^{(i)} < x^{(i)}$,
$\mathsf{join}(\mathsf{split}_<(L, \mathbf{x}, i), \mathsf{split}_<(R, \mathbf{x}, i), j)$
if $i \neq j$ and $y^{(i)} > x^{(i)}$.

The $\mathsf{split}_>$ operation is defined analogously. The symbolic equations are, if T is not empty,



$\mathsf{split}_>(T, \mathbf{x}, i) =$
$\mathsf{split}_>(R, \mathbf{x}, i)$
if $i = j$ and $y^{(i)} < x^{(i)}$,
$\mathbf{y}, j$
$\mathsf{split}_>(L, \mathbf{x}, i)$
R
if $i = j$ and $y^{(i)} > x^{(i)}$,
$\mathsf{join}(\mathsf{split}_>(L, \mathbf{x}, i), \mathsf{split}_>(R, \mathbf{x}, i), j)$
if $i \neq j$ and $y^{(i)} < x^{(i)}$,
$\mathbf{y}, j$
$\mathsf{split}_>(L, \mathbf{x}, i)$
$\mathsf{split}_>(R, \mathbf{x}, i)$
if $i \neq j$ and $y^{(i)} > x^{(i)}$.

It is not difficult to see that both $\mathsf{split}_<(T, \mathbf{x}, i)$ and $\mathsf{split}_>(T, \mathbf{x}, i)$ compare $\mathbf{x}$ against the same keys in T as if we were performing a partial match with one attribute—the i^{th} attribute—specified. But the additional cost of the join algorithm must be taken into account.

We now describe the algorithm join, whose input is a pair of r-Kd-trees A and B and a discriminant i. By definition this algorithm is applied only when the i^{th} attribute of all keys in A is smaller than the i^{th} attribute value of any key in B. As we have already pointed out, in order to produce random r-Kd-trees, each node of A and each node of B must have some probability of becoming the root of the new tree $T = \mathsf{join}(A, B, i)$ (because of Definition 3.2).

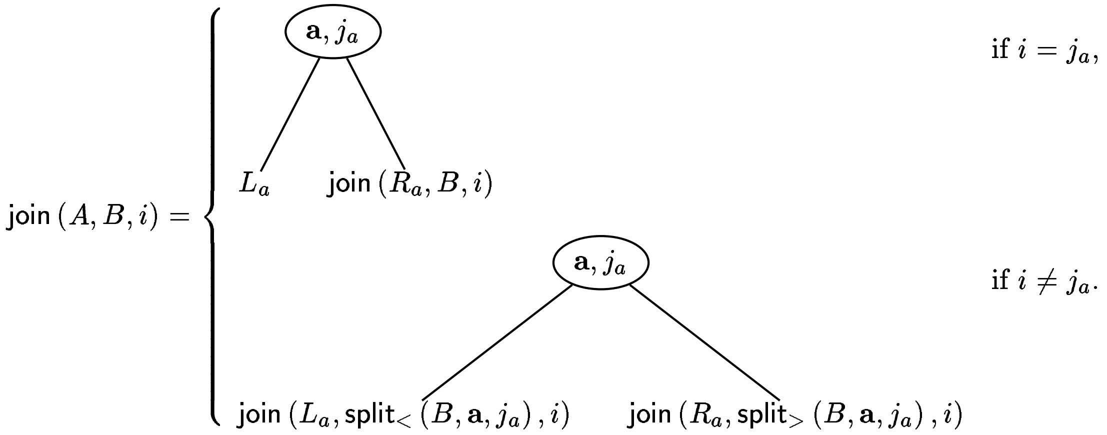
If A and B have sizes n and m respectively, then $T = \mathsf{join}(A, B, i)$ has size $n + m$. Hence, if $n > 0$ and $m > 0$ the join algorithm selects with probability $\frac{n}{n+m}$ the root $(\mathbf{a}, j_a)$ of A as the root of T and with complementary probability the root $(\mathbf{b}, j_b)$ of B. We have the following three cases to consider:

1. If A and B are both empty, then, T is the empty tree;
2. If only one of them is empty, then T is equal to the non-empty one (either A or B);
3. Let A and B be both non-empty trees with roots $(\mathbf{a}, j_a)$, $(\mathbf{b}, j_b)$, left subtrees L_a, L_b and right subtrees R_a and R_b, respectively. In this situation, if $(\mathbf{a}, j_a)$ is selected to become the root of T there are two sub-cases:
   (a) If $i = j_a$ then the left subtree of T is L_a and its right subtree is the result of joining R_a with B (since all keys in B have i^{th} attributes greater than $a^{(j_a)} = a^{(i)}$);
   (b) If $i \neq j_a$ then the left subtree of T is the result of joining L_a with $\mathsf{split}_<(B, \mathbf{a}, j_a)$ and the right subtree of T is the result of joining R_a with $\mathsf{split}_>(B, \mathbf{a}, j_a)$.

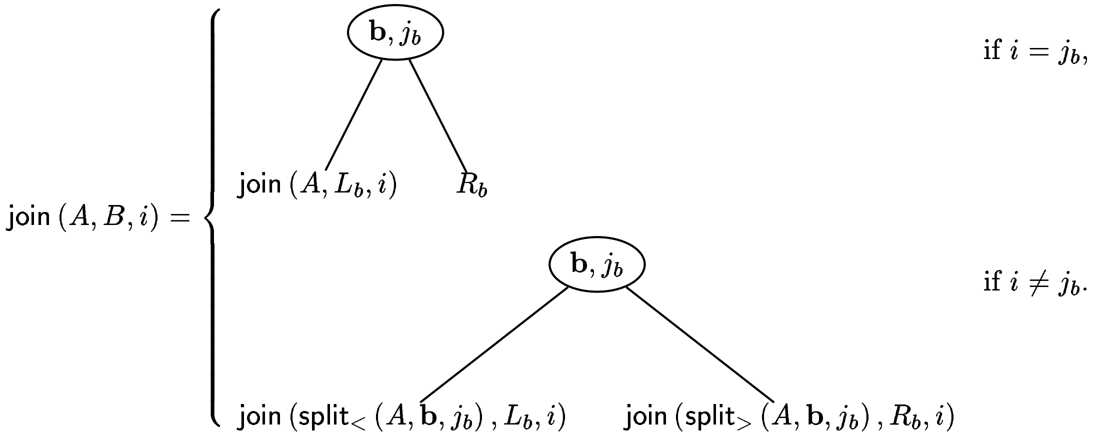
   Two analogous (and symmetrical) subcases arise if the root $(\mathbf{b}, j_b)$ of B is selected to become the root of T.

Symbolically, if A and B are both non empty we have

- With probability $\frac{n}{n+m}$,


$\mathsf{join}(A, B, i) =$
$\mathbf{a}, j_a$
L_a
$\mathsf{join}(R_a, B, i)$
if $i = j_a$,
$\mathbf{a}, j_a$
$\mathsf{join}(L_a, \mathsf{split}_<(B, \mathbf{a}, j_a), i)$
$\mathsf{join}(R_a, \mathsf{split}_>(B, \mathbf{a}, j_a), i)$
if $i \neq j_a$.

- With probability $\frac{m}{n+m}$,


$\mathsf{join}(A, B, i) =$
$\mathbf{b}, j_b$
$\mathsf{join}(A, L_b, i)$
R_b
if $i = j_b$,
$\mathbf{b}, j_b$
$\mathsf{join}(\mathsf{split}_<(A, \mathbf{b}, j_b), L_b, i)$
$\mathsf{join}(\mathsf{split}_>(A, \mathbf{b}, j_b), R_b, i)$
if $i \neq j_b$.

Observe that, the join algorithm traverses the tree in a similar way than the partial match algorithm with one specified attribute, with the additional cost of the split algorithm.

The randomized split and join algorithms preserve the randomness of their input (see Lemma 5.1 below). In other words, when applied to random r-Kd-trees, both the split and the join algorithms produce random r-Kd-trees. Moreover, since this happens, the insert and delete algorithms when applied to random r-Kd-trees produce random r-Kd-trees. All these claims are made explicit in Lemma 5.1, Theorem 5.1 and Theorem 5.2.

Lemma 5.1 *Let T be a r-Kd-tree and let $T_{<_i}$ and $T_{>_i}$ be the r-Kd-trees produced by split$_<$ $(T, \mathbf{x}, i)$ and split$_>$ $(T, \mathbf{x}, i)$, respectively, where $\mathbf{x}$ is any key compatible with T. Then, if T is a random r-Kd-tree then $T_{<_i}$ and $T_{>_i}$ are independent random r-Kd-trees.*

Let T' be the r-Kd-tree produced by join(A, B, i), where A and B are r-Kd-trees such that, for all keys $\mathbf{x}$ of A and all keys $\mathbf{y}$ of B, $x^{(i)} < y^{(i)}$. Then, if A and B are independent random r-Kd-trees then T' is a random r-Kd-tree.

Proof. We prove the two parts of this lemma by induction: on the size n of T to show that split$_<$ and split$_>$ preserve randomness, and on the joint size $n = |A| + |B|$ of T' to show that join also preserves randomness. Observe that to prove the two parts of the lemma for the size n, we will need to inductively and simultaneously assume that both statements are true if T (T') is of size smaller than n. The reason for that is the mutual recursion between the split operations and join.

If T is empty $(n = 0)$ then both split$_<$ $(T, \mathbf{x}, i)$ and split$_>$ $(T, \mathbf{x}, i)$ are empty trees, and hence the first part of the lemma trivially holds for the basis of the induction. Also, if A and B are empty $(n = |A| + |B| = 0)$ then T' is empty and hence random.

Let us consider the case where $n > 0$, assuming now that both parts of the lemma are true for all sizes smaller than n.

We start with the split process, and let n denote the size of T, the r-Kd-tree to be partitioned. Let the root of T be $(\mathbf{y}, j)$ and let L and R denote its left and right subtrees, respectively.

If $i = j$ and $y^{(i)} < x^{(i)}$, then the tree $T_{>_i}$ and the right subtree R' of the tree $T_{<_i}$ are computed by applying recursively split$_<$ and split$_>$ to the subtree R. Since $|R| < n$, by the inductive hypothesis, both $T_{>_i}$ and R' are random and independent. The left subtree of $T_{<_i}$ is L, the left subtree of T. Since T is assumed to be random, L must be random and independent of R. Therefore, L is also independent of R'. In summary, both subtrees of $T_{<_i}$ are random and independent r-Kd-trees, and $T_{<_i}$ and $T_{>_i}$ are independent. To complete the proof for this case, we need only to show that for every key $\mathbf{z}$ of $T_{<_i}$ and every discriminant $j' \in \{1, \ldots, K\}$, the probability that the pair $(\mathbf{z}, j')$ is at the root of $T_{<_i}$ is $\frac{1}{mK}$, where m is the size of $T_{<_i}$. Indeed,

$$\begin{aligned} &\mathbb{P}\left\{(\mathbf{z}, j') \text{ is the root of } T_{<_i} \,|\, y^{(i)} < x^{(i)} \text{ and } i = j\right\} \\ &= \frac{\mathbb{P}\left\{(\mathbf{z}, j') \text{ is the root of } T \text{ and } y^{(i)} < x^{(i)} \text{ and } i = j\right\}}{\mathbb{P}\left\{y^{(i)} < x^{(i)} \text{ and } i = j\right\}} \\ &= \frac{\frac{1}{nK}\frac{1}{K}}{\frac{m}{n}\frac{1}{K}} = \frac{1}{mK}. \end{aligned}$$

The case in which $i = j$ and $y^{(i)} > x^{(i)}$ can be proved in a similar way. Consider now, the case in which $i \neq j$ and $y^{(i)} < x^{(i)}$. In this case the split process is recursively applied to both L and R yielding, by the inductive hypothesis, four independent random r-Kd-trees, namely, $L_{<_i}$, $L_{>_i}$, $R_{<_i}$and $R_{>_i}$. The tree $T_{<_i}$ is built by attaching the trees $L_{<_i}$ and $R_{<_i}$ to the root $(\mathbf{y}, j)$ and $T_{>_i}$ is obtained by joining w.r.t. j the trees $L_{>_i}$ and $R_{>_i}$. As $|L_{>_i}| + |R_{>_i}| < n$, we have that $T_{>_i} = \mathsf{join}\,(L_{>_i}, R_{>_i}, j)$ is, by the inductive hypothesis, a random r-Kd-tree, clearly independent of $T_{<_i}$. Furthermore, the probability that, for any key $\mathbf{z}$ in $T_{<_i}$ and any discriminant $j' \in \{1, \ldots, K\}$, the pair $(\mathbf{z}, j')$ is the root of $T_{<_i}$, given that $i \neq d$, is $\frac{1}{mK}$, where m is the size of $T_{<_i}$:

$$\begin{aligned}
&\mathbb{P}\left\{(\mathbf{z}, j') \text{ is root of } T_{<_i} \,|\, y^{(i)} < x^{(i)} \text{ and } i \neq j\right\} \\
&\quad = \frac{\mathbb{P}\left\{(\mathbf{z}, j') \text{ is root of } T \text{ and } y^{(i)} < x^{(i)} \text{ and } i \neq j\right\}}{\mathbb{P}\left\{y^{(i)} < x^{(i)} \text{ and } i \neq j\right\}} \\
&\quad = \frac{\frac{1}{nK}\left(1 - \frac{1}{K}\right)}{\frac{m}{n}\left(1 - \frac{1}{K}\right)} = \frac{1}{mK}.
\end{aligned}$$

The symmetric case where $i \neq j$ and $y^{(i)} > x^{(i)}$ can be proved in a similar way.

Now we tackle the second part of the lemma and show that join preserves randomness when $n > 0$. If either A or B is empty, then join returns the non-empty tree in the pair which, by hypothesis, is random. Thus we shall only consider the case where both $|A| > 0$ and $|B| > 0$. Let A have root $(\mathbf{a}, j_a)$, left subtree L_a and right subtree R_a, and let B have root $(\mathbf{b}, j_b)$, left subtree L_b and right subtree R_b. If we select the pair $(\mathbf{a}, j_a)$ to become the root of T', and $i = j_a$ then we will recursively join R_a and B. By the inductive hypothesis, the result is a random r-Kd-tree. Thus, we have that the left subtree of T' is L_a, which is random r-Kd-tree and that the right subtree is $\mathsf{join}\,(R_a, B, i)$ which is also random. Both subtrees are independent and the probability that $(\mathbf{a}, j_a)$ was the root of L times the probability that it is selected as root of T' is $\frac{1}{|A| \cdot K} \frac{|A|}{|A|+|B|} = \frac{1}{nK}$. The same reasoning shows that the lemma is true when $(\mathbf{b}, j_b)$ is chosen as the root of T' and $i = j_b$.

Finally, if $(\mathbf{a}, j_a)$ is chosen as the root of T' but $i \neq j_a$ then we need first to split B w.r.t. to $\mathbf{a}$ and discriminant j_a. This process yields, by the inductive hypothesis, two independent random r-Kd-trees, namely, $B_{<_{j_a}} = \mathsf{split}_<\,(B, \mathbf{a}, j_a)$, and $B_{>_{j_a}} = \mathsf{split}_>\,(B, \mathbf{a}, j_a)$. Then, join is recursively applied to the pairs $(L_a, B_{<_{j_a}})$ and $(R_a, B_{>_{j_a}})$ yielding, by the inductive hypothesis, two random independent r-Kd-trees that are attached to the root $(\mathbf{a}, j_a)$ as left and right subtrees, respectively. The probability that $(\mathbf{a}, j_a)$ is the root of T' is $\frac{1}{K(|A|+|B|)} = \frac{1}{nK}$ for the same reason as in the previous case (when $(\mathbf{a}, j_a)$ was selected to become the root but $i = j_a$ instead). Also, if $(\mathbf{b}, j_b)$ were chosen as the root of T' and $i \neq j_b$ then lemma holds, by reasoning as before. □

Theorem 5.1 *If T is a random r-Kd-tree that contains the set of keys X and $\mathbf{x}$ is any key compatible with X, then $\mathsf{insert}\,(T, \mathbf{x})$ returns the random r-Kd-tree containing the set of keys $X \cup \{\mathbf{x}\}$.*

Proof. The proof is by induction on the size n of T. If $n = 0$, then T is the empty tree (a random r-Kd-tree), and $\mathsf{insert}\,(T, \mathbf{x})$ returns a random r-Kd-tree with root $(\mathbf{x}, i)$, and two empty subtrees, where i is uniformly generated from $\{1,\ldots,K\}$. We assume now that T is not empty and that the theorem is true for all sizes $< n$. The insertion of $(\mathbf{x},i)$ in T have two possible results, with probability $\frac{1}{(n+1)}$, $(\mathbf{x},i)$ is the root of $T' = \mathsf{insert}\,(T,\mathbf{x})$ and with complementary probability $(\mathbf{x},i)$ is recursively inserted in the corresponding left or right subtree of T.

Let us consider first the case in which $(\mathbf{x},i)$ is not inserted at the root of T. Consider an item $(\mathbf{y},j) \in T$. The probability that $(\mathbf{y},j)$ is the root of T before the insertion of $\mathbf{x}$ is $\frac{1}{Kn}$, since by hypothesis T is a random r-Kd-tree. The probability that $(\mathbf{y},j)$ is at the root of T' is the probability that $\mathbf{x}$ is not at the root of T' and $(\mathbf{y},j)$ was at the root of T, which is $\frac{1}{Kn} \times \frac{n}{n+1}$, resulting in the desired probability. Moreover, since $(\mathbf{x},i)$ is not inserted at the root of T' at the first step, the insertion proceeds recursively in either the left or the right subtrees of T, which are independent random r-Kd-trees of sizes $< n$. One of them is not modified during the insertion and is a random r-Kd-tree, and by the inductive hypothesis, the other one, after insertion, is also a random r-Kd-tree. Thus, T' is a random r-Kd-tree of size $n+1$.

On the other hand, with probability $\frac{1}{n+1}$, $(\mathbf{x},i)$ becomes the root of T'. Since i is the discriminant of $\mathbf{x}$ with probability $\frac{1}{K}$, the resulting probability is $\frac{1}{K(n+1)}$ as expected. The tree T, which by hypothesis is a random r-Kd-tree, must be split with respect to $(\mathbf{x},i)$, and because of Lemma 5.1 this step produces two independent random r-Kd-trees which are the left and right subtrees of T', thus T' is also a random r-Kd-tree. □

Theorem 5.2 *If T is a random r-Kd-tree that contains the set of keys X, then, $\mathsf{delete}\,(T,\mathbf{x})$ produces a random r-Kd-tree T' that contains the set of keys $X\backslash\{\mathbf{x}\}$.*

Proof. If the key $\mathbf{x}$ is not in T, then the algorithm does not modify the tree, which is random. Let us now suppose that $\mathbf{x}$ is in T, this case is proved by induction on the size of the tree. If $n = 1$, $\mathbf{x}$ is the only key in the tree and after deletion we obtain the empty tree which is a random r-Kd-tree. We now assume that $n > 1$ and that the theorem is true for all sizes $< n$. If $\mathbf{x}$ was not the key at the root of T we proceed recursively either in the left or in the right subtrees, and by inductive hypothesis we obtain a random subtree. If $\mathbf{x}$ was the key at the root of T, the tree after deletion is the result of joining the left and right subtrees of T, which produce a random r-Kd-tree because of Lemma 5.1. Finally, after deletion, each node has probability $\frac{1}{K(n-1)}$ of being the root. Indeed, let $(\mathbf{y},j)$ be any (key,discriminant) pair in T such that $\mathbf{y} \neq \mathbf{x}$,

$$\begin{aligned}
\mathbb{P}\left\{\mathbf{y} \text{ is the root of } T'\right\} &= \mathbb{P}\left\{\mathbf{y} \text{ is the root of } T' \,|\, \mathbf{x} \text{ was not the root of } T\right\} \\
&\quad \times \mathbb{P}\left\{\mathbf{x} \text{ was not the root of } T\right\} \\
&\quad + \mathbb{P}\left\{\mathbf{y} \text{ is the root of } T' \,|\, \mathbf{x} \text{ was the root of } T\right\} \\
&\quad \times \mathbb{P}\left\{\mathbf{x} \text{ was the root of } T\right\} \\
&= \frac{1}{n-1} \times \frac{n-1}{n} + \frac{1}{n-1} \times \frac{1}{n} \\
&= \frac{1}{n-1}
\end{aligned}$$

Since j has probability $1/K$ of being the discriminant of $\mathbf{y}$, we obtain the desired probability. □

Combining the two previous theorems we obtain the following important corollary.

Corollary 5.1 *The result of any arbitrary sequence of insertions and deletions, starting from an initially empty tree is always a random r-Kd-tree.*

6 Final Remarks

Throughout this work we have presented a randomized generalization of Kd-trees (the randomized Kd-trees) that inherits the simplicity, robustness, flexibility and efficiency of standard Kd-trees, yet guarantees the expected performance of all the operations (updates and all associative queries, including those considered in this paper) because the randomized algorithms are free from any assumption about the order in which updates (insertions and deletions) are performed. In particular, the expected external path length of randomized Kd-trees is $\mathcal{O}(n \log n)$, exact match queries can be answered in $\mathcal{O}(\log n)$ time while partial match queries with s out of K keys specified can be answered in $\mathcal{O}(n^{1-\frac{2}{3}\frac{s}{K}+\mathcal{O}((s/K)^2)})$ time. The expected cost of nearest neighbor queries is $\mathcal{O}(\log n)$. The expected cost of deletions and insertions is also $\mathcal{O}(\log n)$. None of the update operations or queries require preprocessing. Finally, randomized Kd-trees are also efficient in space ($\mathcal{O}(n)$), requiring only slightly more space than standard Kd-trees, because for randomized Kd-trees we need to store, for each node, its discriminant and the size of the subtree beneath it.

It is important to point out that similar results can be obtained by means of the technique of Aragon and Seidel [1]. In this approach, a priority is assigned uniformly at random to each record and a r-Kd-tree is built as usual with the additional restriction that no node can have subtrees whose elements have higher priority than the given node (that is, they must respect the so called treap order). The split and join algorithms can be easily modified to produce random r-Kd-trees using these priorities instead of random choices made upon the sizes of the subtrees.

Acknowledgements

We thank Carme Àlvarez, Salvador Roura, Fatos Xhafa and anonymous referees for their useful comments on earlier versions of this work. We also thank H. Prodinger for technical advice and comments about the analysis of partial match.

References

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

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

[3] J. Culberson. The effect of updates in binary search trees. In *ACM Sym. Theory of Computing (STOC'85)*, pages 205–212, 1985.

[4] W. Cunto, G. Lau, and Ph. Flajolet. Analysis of kdt-trees: kd-trees improved by local reorganisations. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Work. Algorithms and Data Structures (WADS'89)*, volume 382 of *LNCS*, pages 24–38. Springer-Verlag, 1989.

[5] J. L. Eppinger. An empirical study of insertion and deletion in binary search trees. *Communications of the ACM*, 26(9):663–669, 1983.

[6] Ph. Flajolet and A. Odlyzko. Singularity analysis of generating functions. *SIAM J. on Discrete Mathematics*, 3(1):216–240, 1990.

[7] Ph. Flajolet and C. Puech. Partial match retrieval of multidimensional data. *Journal of the ACM*, 33(2):371–407, 1986.

[8] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[9] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 2nd edition, 1994.

[10] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1998.

[11] H. M. Mahmoud. *Evolution of Random Search Trees.* John Wiley and Sons, 1992.

[12] C. Martínez, A. Panholzer, and H. Prodinger. On the number of descendants and ascendants in random search trees. *Electronic Journal on Combinatorics*, 5(1), 1998. http://www.combinatorics.org.

[13] C. Martínez, A. Panholzer, and H. Prodinger. Partial match queries in relaxed multidimensional search trees. In preparation, 1998.

[14] C. Martínez and S. Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.

[15] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, Cambridge, USA, 1995.

[16] K. Mulmuley. Randomized multidimensional search trees: further results in dynamic sampling. In *IEEE Sym. Foundations of Computer Science (FOCS'91)*, pages 216–227, 1991.

[17] K. Mulmuley. Randomized multidimensional search trees: lazy balancing and dynamic shuffling. In *IEEE Sym. Foundations of Computer Science (FOCS'91)*, pages 180–196, 1991.

[18] M. H. Overmars and J. van Leeuwen. Dynamic multi-dimensional data structures based on quad- and k-d-trees. *Acta Informatica*, 17(3):267–285, 1982.

[19] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[20] S. Roura. An improved master theorem for divide an conquer recurrences. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Int. Col. Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*. Springer-Verlag, 1997.

[21] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[22] R. Sproull. Refinements to nearest neighbor searching in k-dimensional trees. *Algorithmica*, 6:579–589, 1991.

[23] M. J. van Kreveld and M. H. Overmars. Divided k-d-trees. *Algorithmica*, 6:840–858, 1991.