Exchange Internship

# Design, Implementation and Preliminary Analysis of General Multidimensional Trees

| | |
|---|---|
| **Author:** | Nikolett Bereczky |
| **Supervisors:** | Amalia Duch Brown, Krisztián Németh |
| **Ponent:** | Lluís Solano |
| **Created at:** | Barcelona, June 2012 |

Escola Tècnica Superior

d'Enginyeria Industrial de Barcelona

# Abstract

In this thesis, a new multidimensional data structure, the q-kd tree, for storing points lying in a multidimensional space is defined, implemented and experimentally analyzed. This new data structure has k-d trees and quad-trees as particular cases.

The main difference between q-kd trees and either kd-trees or quad-trees is the way in which discriminants are assigned to each node of the tree. While this is fixed for kd-trees and quad-trees, it is variable for q-kd trees.

We propose two different ways for assigning discriminants to nodes, the heuristics: Split Tendency and Prob-of-1. These heuristics allow us to build what we call quasi-optimal q-kd trees and randomly-split q-kd trees respectively.

Experimentally we show that our variants of q-kd trees are in between quad-trees and k-d trees concerning the memory space and internal path length, and that by proper parameter settings it is possible to construct q-kd trees taylored to the space and time restrictions we can have.

# Table of Contents

# 1 Introduction

Multidimensional data [10] can be represented as a set of points from a higher dimensional space. Managing multidimensional data plays a very important role in various fields of current applications of computer science, like computer graphics, computer vision, image processing, pattern recognition, database management systems, geographic information systems, etc. These applications can embody different kinds of records: for example, consider a vehicle that has associated a record with attributes corresponding to the vehicle's registration number, type, color, capacity and age. Such records can appear in database management systems and can be considered as points, for example, in this case in a five-dimensional space, although they consist of different data types. If a multidimensional data is related to a location data, there is further information –because for all of the attributes the type is the same– such as, for instance, distance in space. It can be very useful to pose queries that involve proximity: for instance, if in a given application we would like to search for the closest town to Immenstadt in a limited two-dimensional region, where the positions of the cities are drawn.

In a continuous physical space, the multidimensional objects can include for example lines, regions, rectangles and surfaces, and they also can appear disjoint or overlapping. A possibility to handle these kinds of data is to store them explicitly by parameterizing them and in that way they can be converted to a point in a higher dimensional space. For instance, if a line segment in a two dimensional space is considered, it can be represented by the coordinate values of its endpoints. Consequently, a transformation from a two-dimensional to a four-dimensional space can be made. This possibility works for example for queries over the objects, but it does not take into account the geometry inherent to data, neither its connection with the space in which the object lies. For instance, if we want to know whether two line segments are close to each other, the result will not be clear, since the proximity that is calculated in the two-dimensional space can be different from the four-dimensional one. This problem could be solved by projecting the lines back to the original space, but other data representations exist that allow us to make such operations directly on the stored data. These data structures are based on spatial occupancy and they split the space in which the spatial data lie into regions.

The starting points of this thesis are the random k-d trees (Chapter 2.2) and the random quad-trees (Chapter 2.3) and two of their specific measures: their internal path length (IPL) (Chapter 2.2.4) and the number of empty subtrees (Chapter 2.2.5) they contain. These two measures are important because they are related to the running time efficiency of their operation (IPL) and to space efficiency (number of empty subtrees).

In particular, random quad-trees are optimal regarding the running times of operations like searching and inserting, because they have a small IPL. However, they are suboptimal considering the space of memory they occupy, which means that they have a lot of empty subtrees. On the contrary, random k-d trees are space optimal, but not time optimal.

Therefore, our goal in this thesis is to determine if there is a trade-off solution, or in other words, if there is a multidimensional data structure that has better time efficiency than kd-trees, and better space efficiency than random quad-trees.

We are able to answer positively this question by introducing the q-kd trees: a new multidimensional data structure that has k-d trees and quad-trees as particular cases. Moreover, we propose two different heuristics for the construction of q-kd trees: the so-called Split Tendency and the so-called Prob-of-1. We then compare experimentally the performance of q-kd trees obtained with our two heuristics against the performance of random quad-trees and random k-d trees.

For the experimental comparison, we have implemented q-kd trees C++ in such a way that it is possible to tune all their parameters. In the same software the quad-trees and k-d trees are implemented as well, so that their performance can be measured against each other.

Although not exhaustively, our experimental results allow us to conclude that it is possible to tune q-kd trees in order to obtain a trade-off between random kd-trees and quad-trees, regarding the time and space requirements.

The structure of this thesis is the following: after this brief introduction we present in Chapter 2 relevant information in connection with the q-kd trees, such as associative retrieval, k-d trees and quad-trees. Afterwards, we introduce the q-kd trees in Chapter 3 together with the heuristics to build them in a nearly optimal way. Then, we describe the software development methodology in Chapter 4, and the results of the experimentations follow this in Chapter 5. Finally, we summarize the project and give ideas of future works in Chapter 6.

# 2 Preliminaries

In this chapter we give a brief introduction to the notions that are required for the development of this work.

## 2.1 The associative retrieval problem

Given a collection F of k-dimensional records, in which each record is a k-tuple of values (or one-dimensional keys) $(K_0, K_1, \ldots, K_{k-1})$ taken from a domain $D = D_0 \times \ldots \times D_{k-1}$, in which every domain $D_i$, $0 \leq i < k$, is totally ordered, and a query Q (which is a question regarding the records in F), we can define the problem of associative retrieval as the problem of answering to query Q, if Q involves a multiplicity of one-dimensional keys [1]. Otherwise, the problem is equivalent to an exact search in a one-dimensional setting.

Associative queries can be classified depending on the kind of search they require. The most characteristic classes are intersection queries and proximity queries.

There are several techniques for building information retrieval systems capable of working with associative queries. In what follows, we fix our attention only in two of them: k-d trees and quad-trees.

### 2.1.1 Intersection queries

Intersection queries are the most common type of queries. They specify that the records to be retrieved are those that intersect some subsets of the records of F. There are several kinds of intersection queries, some of the most common are described below.

#### Exact match queries

The exact match queries are the simplest associative queries: they consist of searching for a specific record in the data structure.

It is very important to know that if the exact match query is the only type of query that will appear in the application, then it is useless to see the problem as associative retrieval. In such a case the keys can be merged together into one superkey and a data structure for one-dimensional storage and retrieval should be employed.

#### Partial match queries

A partial match query q specifies t one-dimensional keys, leaving k-t not specified. The goal is to find all records in F that match the specified coordinates of q.

#### Region queries

Region queries are delimited by a region of the search space in which the records of F lie. The goal is to find all the records that fall in the given region.

### 2.1.2 Proximity queries

Another important family of associative queries is the group of the proximity queries. Among proximity queries, the most common queries are nearest neighbor queries. As an example for one of these queries, given a point P in the space, in which the records of F lie, and the goal is to find the record F that is the nearest to P.

## 2.2 K-d trees

In this section, a short introduction to k-d trees is given. K-d trees (abbreviation of k-dimensional trees) were introduced by Jon Louis Bentley and this chapter is based on his work [1].

### 2.2.1 Definition

A k-d tree is a multi-dimensional binary search tree, where k is the dimension of the search space from which the records are drawn. It can be seen as a data structure for storage of information to be retrieved by associative searches. This data structure can handle several types of queries in a very efficient way.

K-d trees can be defined as follows.

A k-d tree T of n multidimensional records is:

- either an empty tree, if n=0,
- or, if n>0, a binary search tree whose root stores
    - a k-dimensional record $P=(K_0(P), \ldots , K_{k-1}(P))$,

- two pointers, LOSON(P) and HISON(P), pointing to the two subtrees of the node (that are also k-d trees or they are null pointers, if that subtree is empty)
- a discriminator j (which is an integer between 0 and k-1)
- and the following invariant is true: for any node Q= $(K_0(Q),\ldots,K_{k-1}(Q))$ in LOSON(P), it is true that $K_j(Q) \leq K_j(P)$ and for any node R=$(K_0(R), \ldots, K_{k-1}(R))$ in HISON(P) it is true that $K_j(R) > K_j(P)$.

Please note that this definition tells us to insert a record to the LOSON(P), if $K_j(Q) = K_j(P)$. This is a little simplification to the original definition given by Bentley in [1].

There are several ways to assign discriminants to nodes. In standard k-d trees the discriminator of the root is 0, the discriminator of the next level, which is the second, is 1, and the discriminator of the kth level of the whole tree is k-1. Then the discriminator of the (k+1)th level will be again 0 and it continues cyclically. In general, given a node P, DISC(LOSON(P)) = DISC(HISON(P)) = (DISC(P)+1) mod k.

*Figure 1* shows an example of k-d trees. In this example k=2, and the keys are integers between 0 and 512.

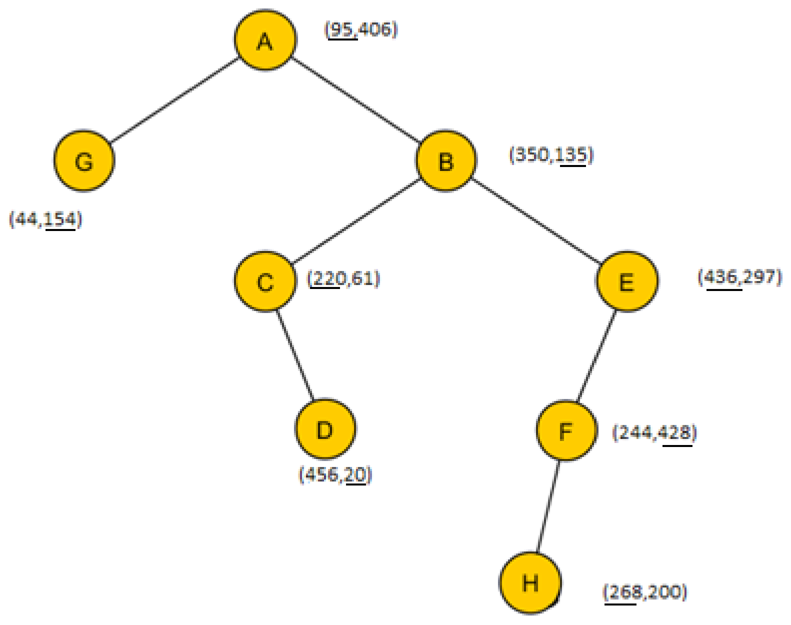

*Figure 1. Example for a k-d tree*

It is worth to observe that k-d tree induces a partition of the space from which the stored records are drawn. *Figure 2* shows the partition induced by the k-d tree of *Figure 1*.

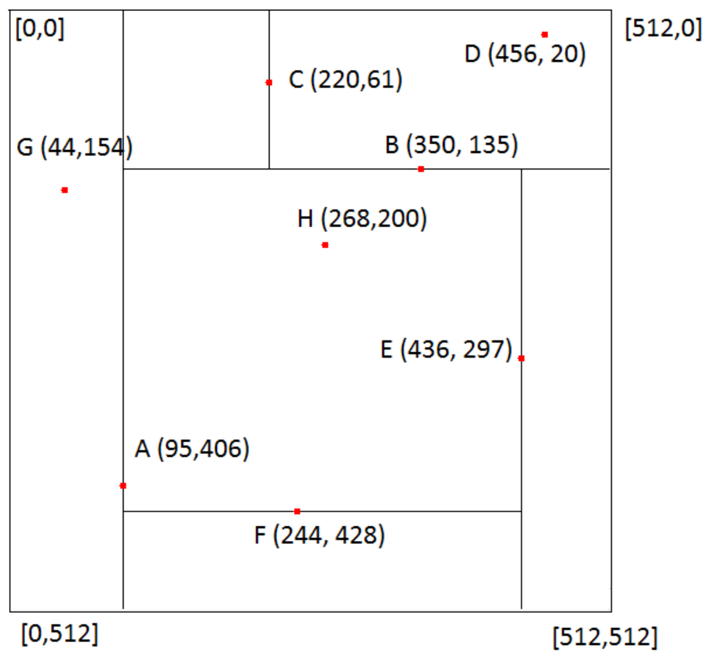

*Figure 2. Partition induced by the k-d tree of Figure 1*

From the induced partition and the sequence of insertions in a multidimensional tree it is possible to define for every node of the tree the notion of bounds array, which can be formally defined as follows:

### 2.2.2 Bounds array

If node P is associated with a bounds array B, then B has 2k entries: B(*0*), … , B(*2k-1*). For all coordinates j ∈ [0, k-1] it is true that B(*2j*) ≤ $K_j$(*Q*) ≤ B(*2j+1*), if Q is a descendant of P and where $K_j$(*Q*) is a key of node Q. Bounds array B should be initialized if for all integers j ∈ [0, k-1], B(*2j*) = -∞ and B(*2j+1*) = ∞.

Informally, the bounds array of a given node corresponds to the region of the space in which the node's key falls when it is inserted in the tree.

An example can be seen in *Figure 1* where the bounds array for node G is (0, 95, 0, 512). This means that all $K_0$ keys in the subtree are bounded by 0 and 95, and all $K_1$ keys are bounded by 0 and 512.

Each hyper-rectangle in the final partition induced by k-d tree corresponds to the bounds array of the leaves of the tree.

### 2.2.3 Exact search, insertion and deletion of records

If we want to insert record P in a k-d tree T then we start from T's root and we compare P with the root as follows:

- If the root is null then we insert a node with key ($K_0$(P),…, $K_{k-1}$(P)), the discriminant will be 0 and HISON(P) and to LOSON(P) will both contain null pointers, and P will become the root of T.
- If the root is not null, let say it has node Q=($K_0$(Q),…, $K_{k-1}$(Q)), and discriminant j, then we compare $K_j$(P) with $K_j$(Q) and we continue the insertion recursively in the appropriate subtree of Q, LOSON(Q) if $K_j(P) \le K_j(Q)$, or HISON(Q) otherwise, until an empty subtree is reached and the record is inserted.

Note that the order in which keys are inserted into k-d trees influence the shape of the tree, and can make the difference on having either well-balanced trees or very unbalanced ones.

The search for an element in a k-d tree is basically the same as the insertion, except that while searching we can stop the traversal of the tree once the sought element is found.

Generally, the deletion of a node in multidimensional trees is difficult. In the case of k-d trees, the deletion of a node is possible but it may be very expensive as well as it may degenerate the shape of the tree. During the deletion it is enough to deal with the root of a given subtree. If P (the node to be deleted) has no descendant then it is very easy, the result will be an empty tree. If P has non-empty subtrees and j is the discriminator, then a node, say Q, from one of its subtrees, has to be inserted in its place, the one which is the j-maximum of his LOSON or the j-minimum of his HISON. After replacing P with Q, an expensive task follows, that is to reorganize the subtrees of the "old" Q.

### 2.2.4 Internal Path Length

In general, all kinds of searches in multidimensional trees consist of following a path (or several paths) of the tree, therefore the length of the longest path, or the sum of the lengths of all paths, have a big influence in the cost or execution time of search and update operations.

In this work we are going to use the IPL (Internal Path Length) [14] of a tree as a potential measure of the cost of the operations regarding the tree. The IPL is defined as the sum of the distance between each internal node and the root. *Figure 3* shows an example of IPL:

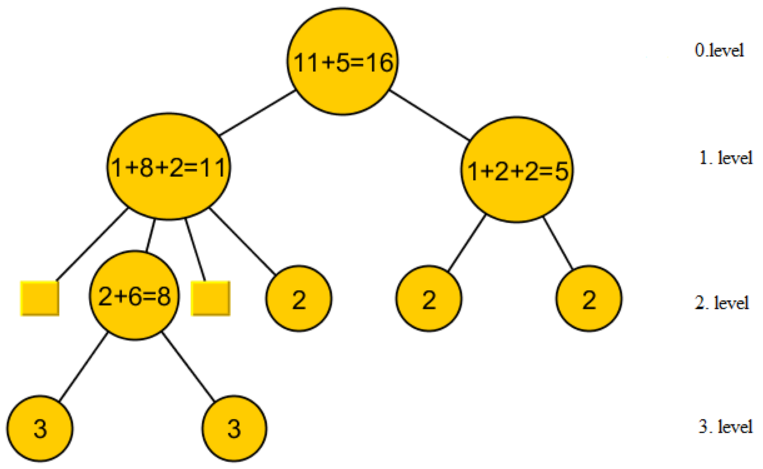

*Figure 3. Example for the IPL of this q-kd tree, which is 16 in this case*

It can be formally proven that the IPL of perfectly balanced k-d trees of n internal nodes is $\log_2(n) + o(1)$ [6], which is also the expected value of IPL for random k-d trees of n internal nodes (random k-d trees are built by a random permutation of the n multidimensional keys).

### 2.2.5 Number of the empty subtrees

We will also be interested in the quantity of the storage space that a multidimensional tree requires. In general, every record needs a node to be stored, but, since every part of the tree ends with an empty node (i.e., a null pointer), a variable quantity of null pointers is also required to be in the corresponding tree.

We are going to use the number of the null pointers present in a given tree as the quantity of space wasted by the tree. So, the less empty subtrees a tree has, the more efficient it is in space terms.

### 2.2.6 Applications

In his article [1], J. L. Bentley considered several areas where k-d trees might have been used. He mentioned possible applications in speech recognition and in geographical information retrieval systems. Here is an example for the latter. Consider a file whose records are cities of Europe in the map. The cities are stored as nodes in a k-d tree and the keys are the latitude and the longitude. There are several possible associative queries:

- An exact match query could be: “Which city is at the latitude of 47° 29′ and at longitude 19° 02′?”
- A partial match query might be: “Which cities are at latitude 41° 22′?”
- A region query can be: “What cities are between the latitudes in the range of 36° 43′ to 36° 51′ and the longitudes in the range of 3° 58′ to 4° 16′?”
- A nearest neighbor query may be: “Which is the closest city to Becske?”

Nowadays, there are more and more areas where k-d trees are used. We can meet them in computer graphics, for example in ray tracing, but also in astronomy or in the field of the approximate fingerprint matching [3].

## 2.3 Quad-trees

Quad-tree is another data structure useful to store k-dimensional records and then to perform associative search on them. It was introduced by J. L. Bentley and Raphael Finkel in 1974 [4]. The literature of quad-trees is very rich, this chapter has been created using these references: [2][7][9][13][8].

If a quad-tree is used to store k-dimensional data, then every internal node has $2^k$ children. So, if we take the case, where the dimension is two, then each node has four children (or no children at all, if it is a leaf) and each child corresponds to a quadrant of the square induced by the coordinates of the record in the search space. This can be seen in *Figure 4* below, where the red points correspond to the nodes in the tree.

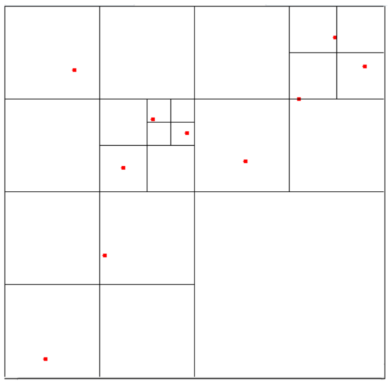

*Figure 4. Example of the partition induces by a quad-tree*

There are several kinds of quad-trees, and they are classified by the type of the data they represent, like points, areas, lines and curves. Thus for example we can have a region quad-tree – in two dimensions it represents a partition of space by decomposing the region into for equal quadrants – or a polygonal map quad-tree – that stores a collection of polygons that can have isolated edges or vertices. An example of these kinds of trees is shown in *Figure 5* and *Figure 6* below.

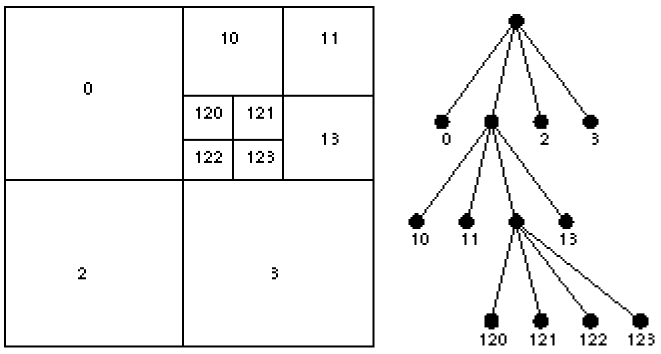

*Figure 5. Region quad-tree [11]*

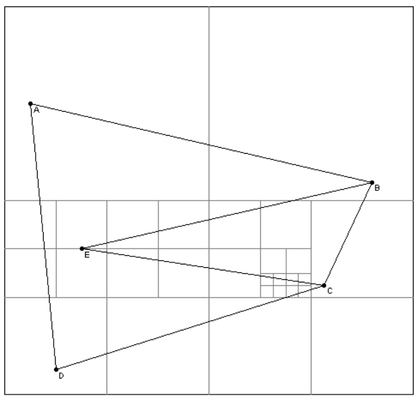

*Figure 6. Polygonal map quad-tree [12]*

If the data that quad-trees represent are points, there are also several possibilities like PR quad-trees and point quad-trees.

### 2.3.1 PR quad-trees

A PR (Point Region) quad-tree stores point sets and it splits squares until each square will contain maximum one point. The tree building algorithm starts with the state where every point to be inserted into the tree is in the square. If the number of the points in a quadrant is more than one then:

- it splits the square into four quadrants
- assigns the points to the squares
- do this recursively for each quadrant

*Figure 7* shows an example of a PR quad-tree, where the circles are the nodes that point to another node, the yellow squares are null pointers (or empty subtrees) and the orange rectangles are the ones that store the points.

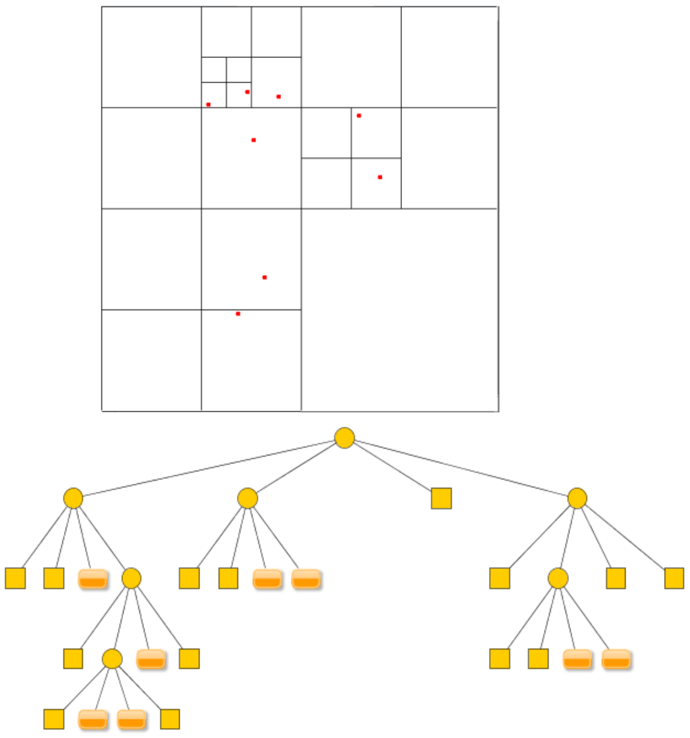

*Figure 7. PR quad-tree and the partition it induces*

The search and the insertion are quite similar to each other. If we want to search for a point in a PR quad-tree, we have to go recursively through the tree and at each node choose the proper way to follow –according to the relation of the keys to the quadrants- and we will find the point as a leaf, if it is in the tree.

If we want to insert a point, we have to go through the tree just as when searching, and in the end we have to check whether it is in a free quadrant. If yes, we have to just put the point in, if not, we have to make a new subdivision and then insert it.

The deletion is quite complicated [1], because in several cases, when a node is deleted, a big part of the tree should be reconstructed.

There are cases, however, where the PR quad-trees are not the best choice, and other data structures are more useful, for example if several points are very close to each other. In this case using PR quad-trees we got an unbalanced tree (see *Figure 8 and Figure 9).*

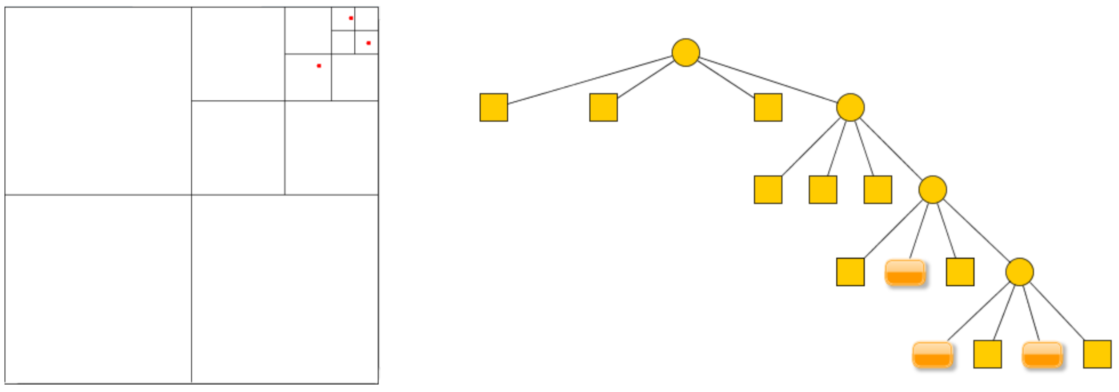

*Figure 8. Unbalanced PR quad-tree with three nodes, depth: 4*

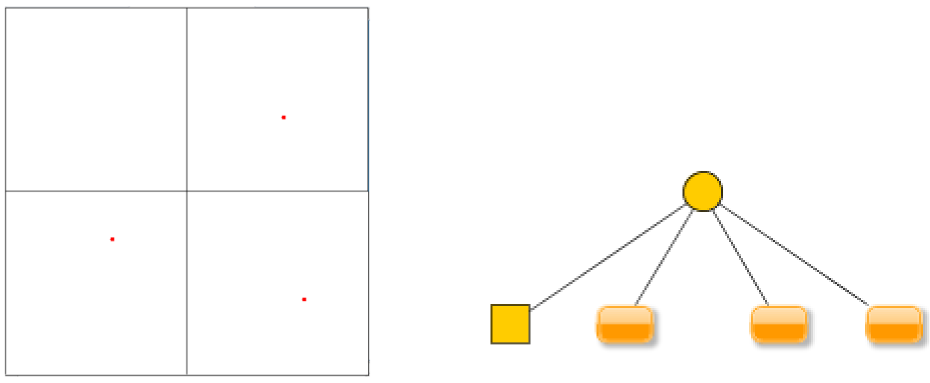

*Figure 9. Balanced PR quad-tree with three nodes, depth:1*

As it is shown in the previous examples, the location of the points highly influences the depth of the trees. Another negative example is when the picture looks like a chess-table, where in each black square there is one point, but no point is in the white areas. In this case, using quad-tree is not recommended because every second leaf will be empty, and thus another data structure may need less memory.

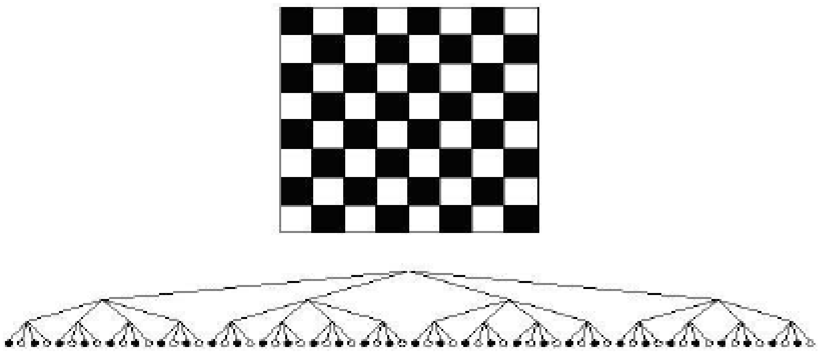

*Figure 10. Example when quad-trees are not recommended (Source: [7])*

To sum up, the memory space that a PR quad-tree requires depends on:

- the number of levels of the tree
- the size of the tree (i.e., the number of nodes in it)
- the location of the points

### 2.3.2 Point quad-trees

In this work we are interested in point quad-trees. When a point quad-tree is constructed, the algorithm also starts with the state where every point is in the search space (a square in a two-dimensional space). If the number of the points in a quadrant is more than one, then it takes the first point to be inserted and splits the square into four quadrants at that point (so the center of each subdivision is a point in the data structure) and it will do it recursively. See *Figure 11* below as an example. When the dimension is greater than two, then the space is spitted into $2^k$ hyper quadrants, exactly in the same way just described.

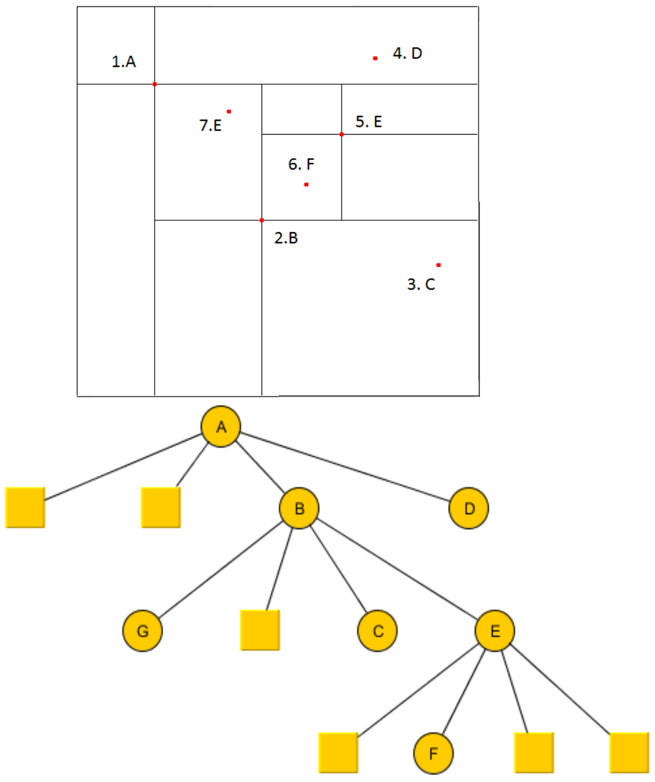

*Figure 11. Point quad-tree and the induced partition of the space*

The search and the insertion of records are very similar to each other and also to these operations in the k-d trees. If we want to search for a record, we have to take the root and compare each coordinate of the record to be inserted with each coordinate of the root then go further in the proper direction and do this recursively until we find the point or there is an empty node. In the latter case the searched record is not in the tree.

If we want to insert a record, we have to go through the tree just as when searching, until we find a leaf record or an empty node. In the latter case we can insert the record to this place. However, if there is another record, we have to make a new subdivision and then insert our record in.

The deletion is very complicated in this case, too, because in several cases if the node would be deleted then a big part of the tree should be reconstructed.

Just as with PR quad-trees, in the case of point quad-trees, the order in which the points are inserted into the tree is very important. An improper order can easily result in an unbalanced quad-tree, as shown in *Figure 12*.

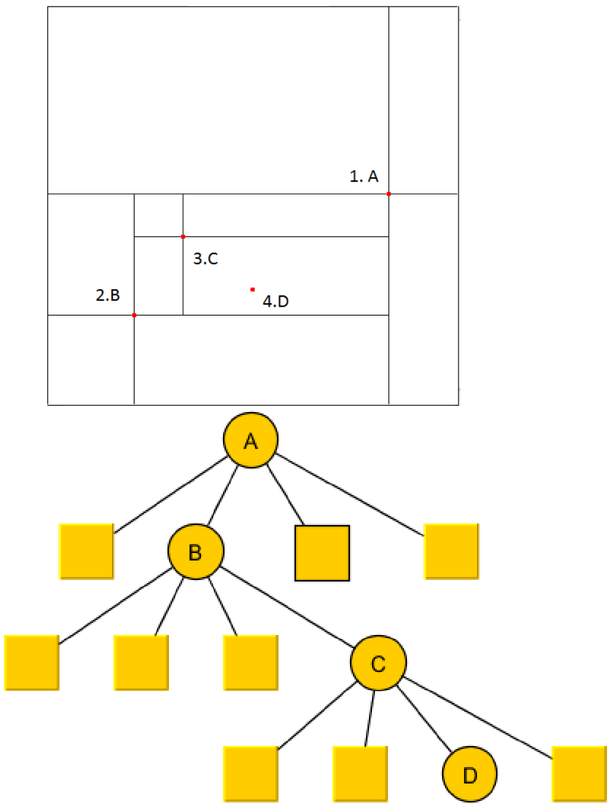

*Figure 12. Unbalanced quad-tree of four nodes, depth: 3*

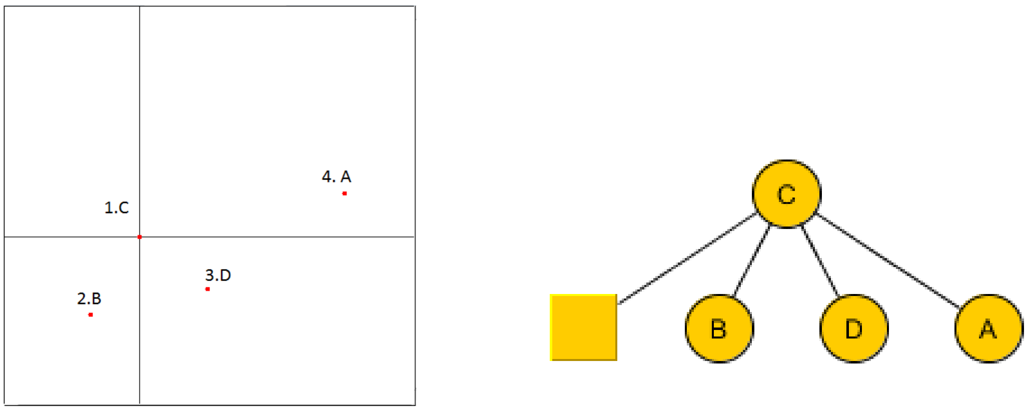

*Figure 13. Balanced quad-tree of four nodes, depth: 1*

Regarding IPL and bounds array the definitions are exactly the same as the ones of k-d trees.

It can be formally proven that the IPL of perfectly balanced quad-trees of n internal nodes is $log_{2^k}(n) + o(1)$ [6] , which is also the expected value of IPL for random quad-trees of n internal nodes (the random quad-trees built by a random permutation of the n multidimensional keys).

### 2.3.3 Applications

Quad-trees are often used in different areas of computer graphics. Although this area is out of the scope of this work, for the sake of completeness, we list here the areas that [5] mentions as application areas of quad-trees: determination of visible objects, collision detection, view frustum culling of terrain data, fractal image analysis and also storing non-uniform meshes. It is important that quad-trees can contain a lot of empty subtrees therefore they are used less frequently than k-d trees.

# 3 Q-kd trees

In this chapter, a new multidimensional data structure useful for associative retrieval is introduced, the q-kd trees. Q-kd trees are a generalization of both quad-trees and k-d trees that is why we named them q-kd trees.

The idea behind q-kd trees arises from the following observation. K-d trees consist of nodes that discriminate by exactly one of the coordinates of the keys that they store, while in quad-trees every node discriminates by all the coordinates of its stored keys. In same sense, k-d trees and quad-trees can be seen as "extreme" cases on the number of discriminating coordinates that node can have (minimal for k-d trees and maximal for quad-trees.) Therefore, it is natural to think in multidimensional trees, where for each node the number of discriminating coordinates lay in between these bounds: the q-kd trees.

Informally, q-kd trees are multidimensional trees in which each node discriminates by a number i (specific for each node), $1 \leq i \leq k$, of coordinates (and thus have $2^i$ subtrees).

## 3.1 Definition

The formal definition is the following:

In a multidimensional setting, if a tree T has $n \geq 0$ nodes then it stores n k-dimensional records and each record K holds k one-dimensional keys: $K = (K_0,\ldots, K_{k-1}) \in D$, where $D = D_0 \times \ldots \times D_{k-1}$, and each $D_j$, $0 \leq j \leq k-1$, is a totally ordered domain. A q-kd tree T is either

- empty when n=0,
- or its root
    - contains a record Kx with k keys, and
    - contains a coordinate split boolean vector $s=(s_0, \ldots,s_{k-1})$ with exactly i ones, where $0\leq i < k$, and i is the order of the vector s, and
    - has $2^i$ subtrees that store the n-1 remaining points as follows. Every subtree $T_w$ is associated to a bitstring $w=w_0w_1\ldots w_{k-1}$, where

$w \in \{0, 1, \#\}^k$, such that $T_w$ is a q-kd tree and for any records $Ky \in T_w$ and for any j, $0 \leq j \leq k-1$, is true that:

- if $s_{j=}0$ then $w_j=\#$
- if $s_j=1$ and $w_i=0$, then $Ky_j \leq Kx_j$
- and if $s_j=1$ and $w_j=1$, then $Ky_j > Kx_j$.

Note that a q-kd tree is a quad-tree if all the values in the split vector are ones and a q-kd tree is a k-d tree when there is always exactly one one in the split vector.

For example, if the dimension is k=5, then a node has maximum $2^5$ subtrees, and if point P1 is in the tree and it has $K_{P1}=(K_0, K_1, K_2, K_3, K_4) = (5, 8, 9, 10, 4)$ keys, and the split vector is $w_{P1}=(w_0w_1w_2w_3w_4) = (10100)$, then if we want to insert the node P2 to the subtree where P1 is the root, then only the $K_0$ and the $K_2$ keys are considered, the others are not important. In this case we have four possibilities, for instance if P2's keys are

- $K_{P2a}=(\underline{3},5,\underline{6},9,6)$ then we insert into the subtree where both coordinates are smaller (see *Figure 14*)
- $K_{P2b}=(\underline{2},8,\underline{11},4,2)$ then we insert into the subtree where one is smaller and the other is bigger
- $K_{P2c}=(\underline{6},7,\underline{8},6,42)$ then we insert into the subtree where one is bigger and the other is smaller
- $K_{P2d}=(\underline{7},9,\underline{13},54,1)$ then we insert into the subtree where both coordinates are bigger

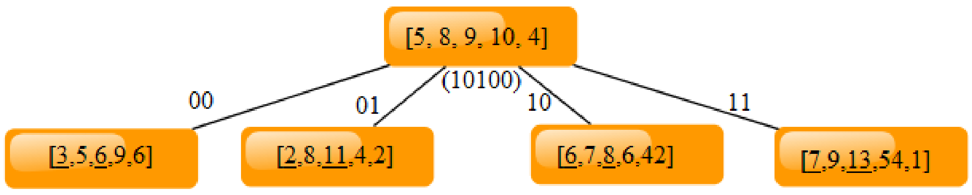

*Figure 14. Example for inserting a record into a q-kd tree*

There are several examples of q-kd trees below. In *Figure 15* (see also *Figure 16*) and in *Figure 17* we denote the keys between [square brackets] and the split vector between (parentheses). The bitstrings are adjacent to the edges and written in smaller letters.

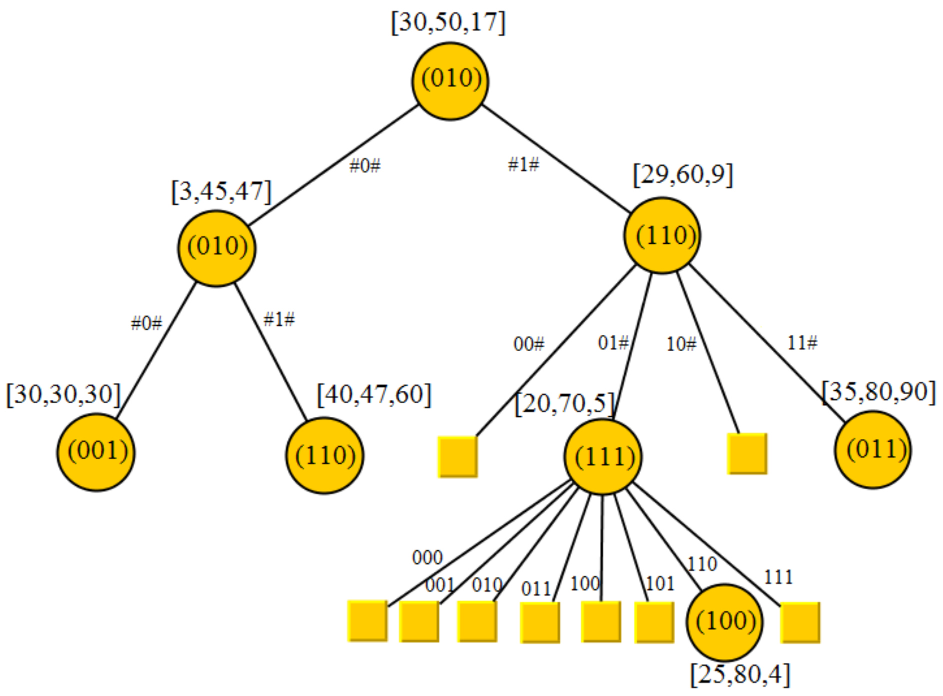

*Figure 15. Example of a 3-dimensional q-kd tree*

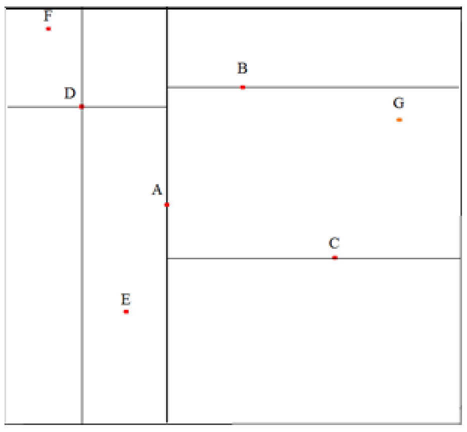

*Figure 16. Example of the partition induced by a 3-dimensional q-kd tree*

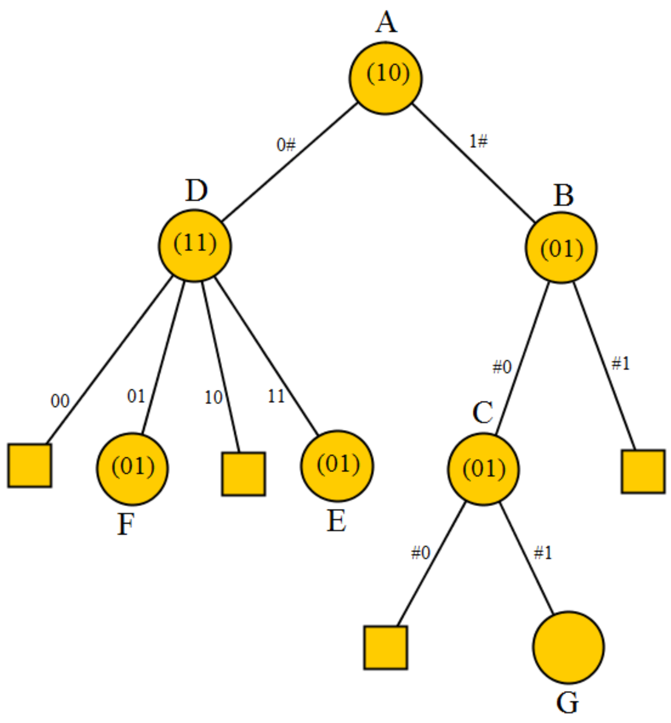

*Figure 17. Example of a 2 dimensional q-kd tree*

## 3.2 Search, insertion and deletion

The search and the insertion of records are very similar to each other and also to these operations in k-d trees and quad-trees.

If we want to search for a record in a q-kd tree, we have to traverse it, starting at the root, and examine each values in the split vector of the root: when there is a one we have to compare the coordinate of the root –which has the same index as that we took in the split vector– with the same coordinate of the record we are looking for. In this way we can choose the appropriate subtree to follow. Then we do this recursively until we find the point or a null leaf, in which case the searched node is not in the tree.

If we want to insert a record, we have to search for it as described above, until we reach to an empty node, where we can insert it.

The deletion of a node, if it is not a leaf, could be very complicated, and it is not yet defined, but it could imply, in general, almost a complete reconstruction of the subtree rooted at the node we want to delete.

Regarding IPL and bounds array the definitions are exactly the same as for the k-d trees.

## 3.3 Applications

Q-kd trees have not been proven yet in any application, but in general, they can be used in the same areas, as in which k-d trees and quad-trees have been proven useful.

## 3.4 Heuristics

As we have already explained, our goal is to compare q-kd trees with random kd-trees and random quad-trees.

To this, we propose two different heuristics in order to build and tune q-kd trees, and then (in Chapter 5) we compare the q-kd trees obtained in this way with their opponents (random k-d trees and random quad-trees).

### 3.4.1 Building a randomly-split q-kd tree

In the randomly-split q-kd tree, the split vector will be generated randomly. By default, the probability of having a one or a zero for a coordinate in the split vector will be the same for all the coordinates. This variable, which will be called Prob-of-1, will be the probability of 1.

Note that it can happen that every coordinate in the split vector is zero, which has no meaning. In this case, the first coordinate will be set to one.

### 3.4.2 Building a quasi-optimal q-kd tree

The question arises: how to build a “good” q-kd tree?

First of all, we have to examine the distribution of the points. In the case of non-uniform distribution, when generating points to be stored in the trees, different probability distributions for the dimensions of the keys could be used. Generally, the coordinates are not independent of each other, but in a simple case the coordinates could be generated independently. In this work, however, uniform distribution will be used because of its simplicity, independently for each coordinate. Nevertheless, in further researches q-kd trees can be built by using non-uniform distribution of the keys.

For making a quasi-optimal q-kd tree, some factors have to be considered. If there are too many empty or small subtrees, it causes waste of memory space, but on the other hand, if the height is big, it increases time of the operations in general. The basic idea to make a quasi-optimal q-kd tree is to stop the discrimination of the coordinates that are not worth to be discriminated, which means that if the splitting was made, it would produce some empty or very small subtrees.

Therefore, if we would take the nodes, that are not inserted yet, into account, then we will have better q-kd trees.

However, this consumes a lot of execution time and its implementation is also quite complicated, because not all the nodes that are not yet in the tree have to be considered, just a part of them, since not all will be in the subtree of the node that is to be inserted.

Nonetheless, there is another possibility, which is using information about the general distribution of the points. This means that if the distribution of the points is known, interferences can be drawn about the points in the subtrees.

Concerning these thoughts, a possible solution is to generate random split vectors when a node is inserted. It is very easy but maybe not the best solution.

What seems to be a better solution is to keep track of the bounds array for each node and compare each coordinate of the node to the its part of the bounds array. If the value is more or less in the middle of the bounds array, then a splitting will be made, if not, this coordinate will not be used because it is not worth. For saying what means valuable, that is, how close a coordinate should be to the center of the bounds array, we will make a variable and make several experiments with it to know approximately the best values. This variable will determine the limits, as shown in the following example.

If the bounds array is <0,100,0,100> and the key is <35,52> and this variable –which will be called Split Tendency– is 30, which means that a splitting will be made if the keys are between the 30% and 70% of the bounds arrays, then the split vector will be <1,1>. But if this variable is 40, which means that a splitting will be made if the keys are between the 40% and 60%, then the split vector will be <0,1>. It is very important to know, that this works only if the points are uniformly distributed.

# 4 Software development methodology for the experiments

This chapter is about the development of the software that we have used later for the experiments. First we set up the design requirements for the software, then we have designed and implemented the code. Finally we checked the correctness of the code, so that it will be ready for experimenting. These steps are detailed in this chapter.

## 4.1 Design of the implementation

In this chapter, the design of the C++ implementation of q-kd trees, which includes k-d trees and quad-trees as special cases, are described.

Before the design and the implementation we have defined the requirements about my software. The software should be developed in standard C++, and should be portable, so it will be able to run on any hardware running any kind of operating system that has a standard C++ compiler.

The software must be able to build k-d trees, quad-trees and both randomly-split and quasi-optimal q-kd trees, and make it possible to insert and search for records and delete nodes. The number of nodes must only be limited by the available memory. For these trees the software must be able to calculate the measured metrics: IPL and the number of empty subtrees.

At the beginning of the design, we have decided that two classes will be defined: Hybrid_node and Hybrid_tree. These will be used to represent the q-kd trees, and its special cases: quad-trees and k-d trees as well, with proper setting of the split vector.

A Hybrid_node will contain the following member variables: the multidimensional key, that will be generated randomly, the pointers to the subtrees, the boolean split vector, which says which coordinates are used, and the bounds array. Each of them will be stored in the Vector class, taken from the C++ Standard template Library (STL).

The Hybrid_tree class will be implemented in order to simplify several operations over the trees, for example building or destroying them. It will have just one, private member variable, which is a pointer to the root of the tree.

The constructor of the Hybrid_node will set the multidimensional key, the split vector and the bounds array to the values that it gets as arguments, and in addition it will set the pointers to the subtrees to null. For this setting, we have to know, how many subtrees a node has. Because it will be important in other cases also, a function will be implemented to determine the number of the subtrees, based on the split vector, and the number of ones in it.

The Hybrid_tree's general insert function will get a multidimensional key and a split vector as arguments. The content of the split vector will depend on the calling function. If we are dealing with a quad-tree all the bits will be one. If it is a q-kd tree, the split vectors can be generated in different ways, that we explain in Chapter 3.4. As for a k-d tree, it will have a particular insert function because in this case the split vector depends on the level where the node will be inserted.

In order to know in which subtree we have to insert a new node, a function will be written that will calculate the index of the proper subtree depending on the keys.

The k-d insert function of the Hybrid_tree will only take a multidimensional key as the argument, and will generate and initialize a split vector in a way that the first bit is one and the others are zeros, and pass it to the root node. Then, the k-d insert function of the Hybrid_node will take a split vector as an argument and at the recursive calls will cyclically rotate it, depending on the actual level.

In order to check the correctness of the resulting trees, a recursive function will be implemented that will write out the tree in preorder sequence. Similarly a recursive search function will be created that will be able to search for an exact key.

The destroy function will be also recursive: each node has to destroy its children and then the node itself will be destroyed by its parent. Finally the root will be deleted by the Hybrid_tree object.

For calculating the metrics, a function will be written which will return the internal path length (more details in Chapter 2.2.4) and another function that will return with the number of the empty subtrees in the tree (see Chapter 2.2.5).

For inserting a node into a quasi-optimal q-kd tree, a function will be written, which will set the bounds to the proper values and calculate for each coordinate whether to split or not.

Finally, the program will be able to get several values in the command line, for instance, the dimension, the number of the nodes and the variable that is used for determining the split vector. It will be also possible to make a batch file, which will produce .csv files with the results. Therefore the process of the experimentation will be simplified.

## 4.2 Testing of the implementation

During and after the coding, we have tested and corrected all the functions step-by-step until they were functioning correctly. To check the validity of the software, we have run some tests and examined the result. We have found that they were conforming to the reference, for instance it was true that the k-d tree is more efficient in space than the quad-tree, but in contrary, the quad-tree is more efficient in time than the k-d tree. To sum it all up, we can state that the software performs as we expected.

# 5 Settings and results of the experiments

This chapter is devoted to explain the experimental settings and results that were obtained.

As it was mentioned above, two metrics were used for these experimentations: the IPL and the number of the empty subtrees. If we compare random quad-trees with random k-d trees, it is known that random k-d trees are better considering the number of the empty subtrees and random quad-trees are better considering IPL.

In our experiments we approximate random k-d trees and random quad-trees by k-d trees and quad-trees built by multidimensional keys drawn from a discrete uniform distribution. In order to obtain the expected value of the IPL and the number of the empty subtrees, we repeat each measure for several trees of the same size and keep the corresponding average.

In the rest of this chapter, we will refer to random quad-trees and random k-d trees simply as quad-trees and k-d trees.

Five series of experiments were made, which are described below, together with the results. In all the cases the distribution of the nodes is uniform in the k-dimensional discrete space, and the same elements have been inserted in the same order, when building the different kind of trees.

It is important to observe that the results are limited, they are just true for these parameters and maybe not in general, but in the future, further experiments may be performed for (possibly) state generalities.

## 5.1 The effect of the Split Tendency

First of all, we have made an experiment where the quad-tree, the k-d tree and the quasi-optimal q-kd tree were compared to each other concerning the IPL and the number of the empty subtrees. Depending on the Split Tendency (see Chapter 3.4.2) there are several q-kd trees. We want to know if it is true that the q-kd trees are between the quad-trees and k-d trees concerning these measured metrics and we also search for the best value for the Split Tendency depending on the application requirements.

For this experiment we used 3 dimensional records, 20000 nodes and 100 runs. The results are shown in *Figure 18* and *Figure 19* below.

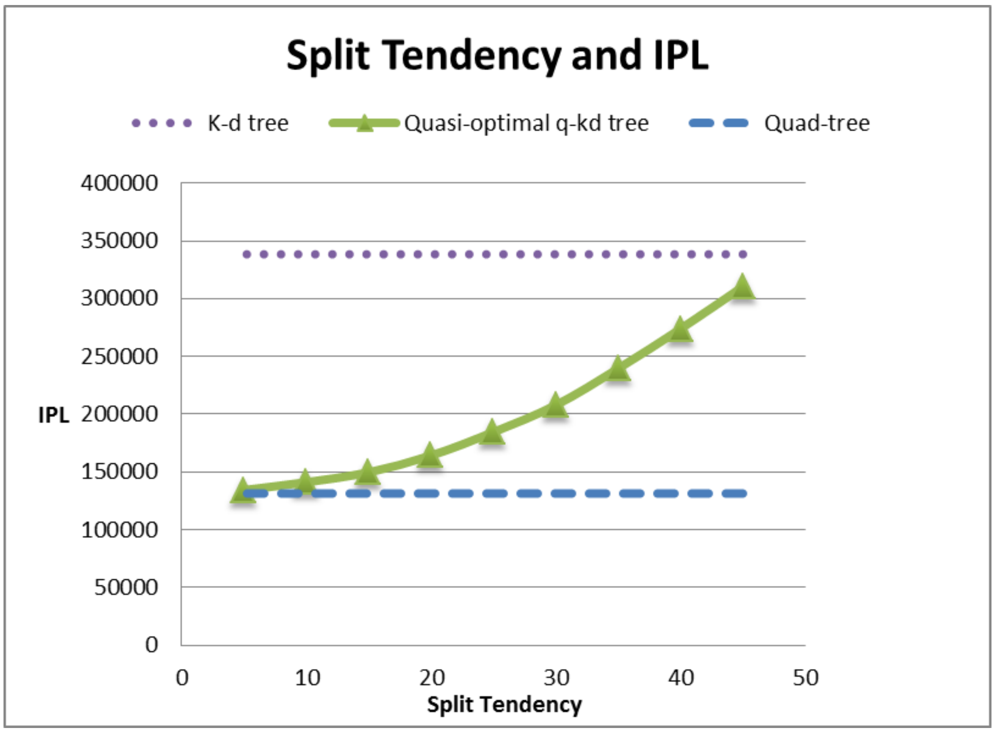

*Figure 18 Split Tendency and IPL*

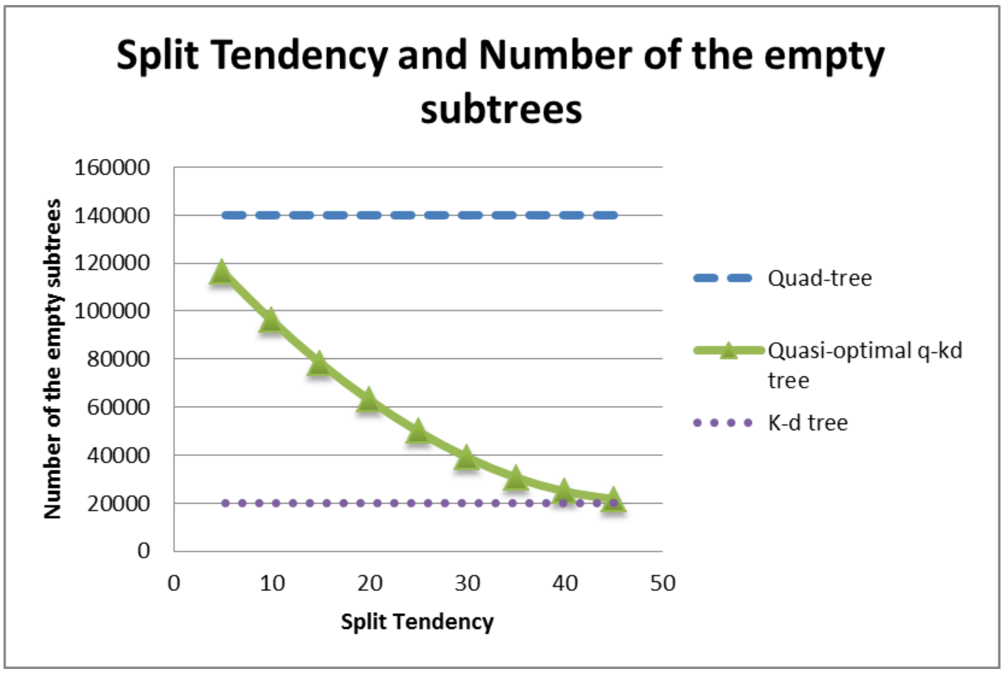

*Figure 19 Split Tendency and Number of the empty subtrees*

As it can be seen on the diagrams, the q-kd trees are between quad-trees and k-d trees considering both the IPL and the number of the empty subtrees. The curves also show that the greater the Split Tendency is, the greater the IPL of the quasi-optimal q-kd tree is and less the space is it needs. Setting this Split Tendency variable a quasi-optimal q-kd tree can be constructed depending on the space and time restrictions we have.

As we mentioned above, the results are limited, they are just true for these numbers and maybe not in general. In the future, further experiments may be made where the dimension or the number of the nodes vary.

## 5.2 The effect of the Prob-of-1

In this experiment, we want to know in the case of randomly-split q-kd tree how the IPL and the number of the empty subtrees depend on the variable Prob-of-1 (see Chapter 3.4.1), which is the probability (in percentage) of the ones in the split vector.

For this experiment we used 3 dimensional records, 20000 nodes and 100 runs. The results are shown in *Figure 20* and *Figure 21*:

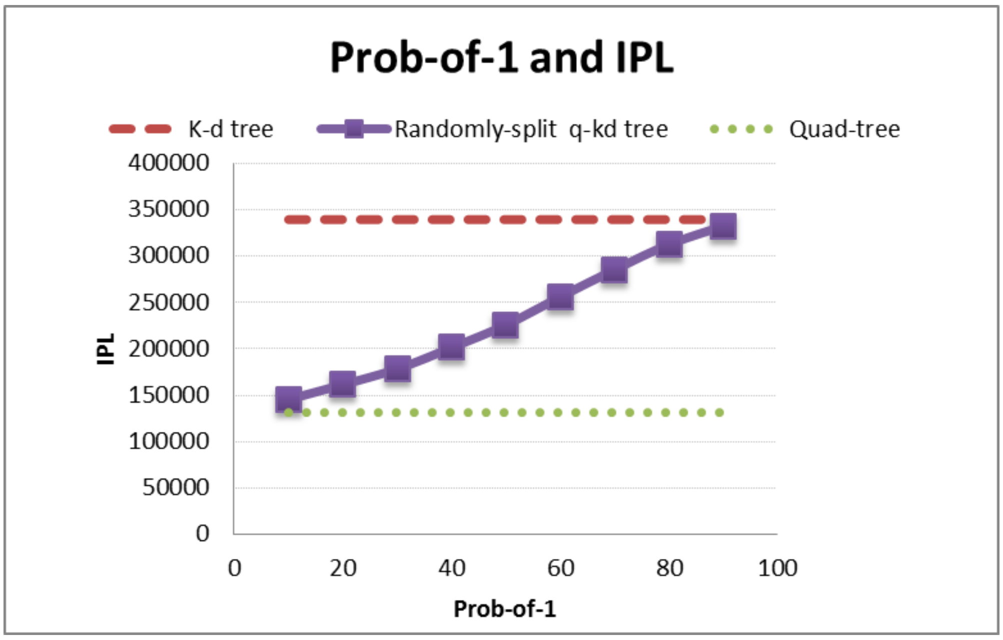

*Figure 20 Prob-of-1 and IPL*

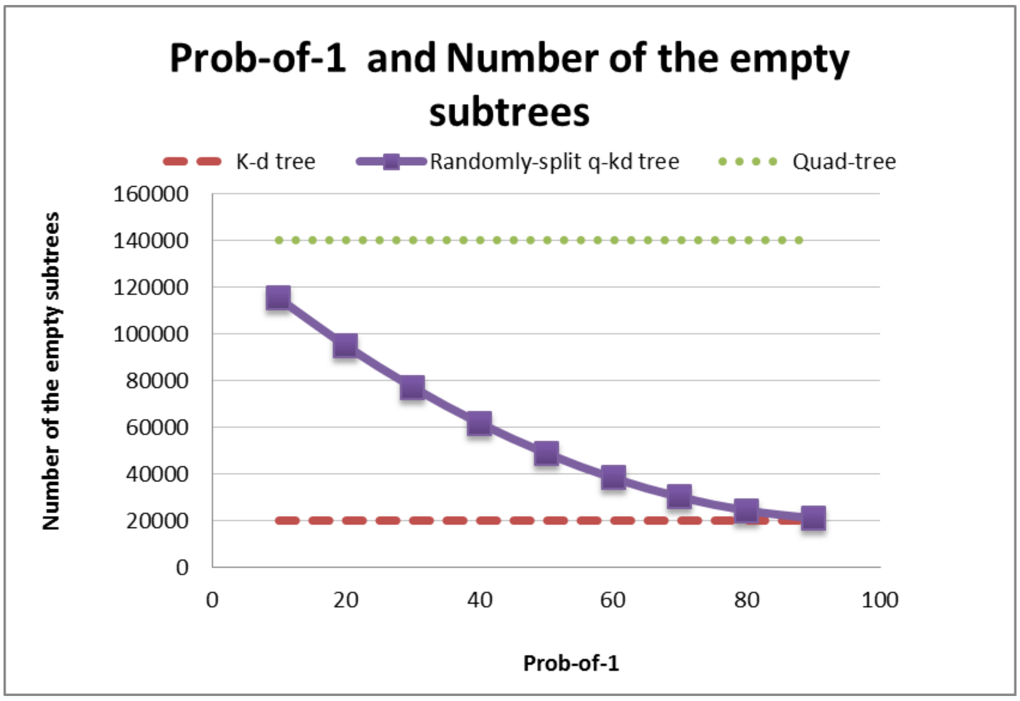

*Figure 21 Prob-of-1 and Number of the empty subtrees*

We can see that the greater the Prob-of-1 is, greater the IPL of the randomly-split q-kd tree is and less the space it needs. With this Prob-of-1 variable a randomly-split q-kd tree can be constructed depending on the space and time restrictions imposed. The results were not surprising, because the more ones we have in the split vector, the more we approximate the quad-trees, and the less ones we have, the more we approximate the k-d trees.

Just like in the previous section, in the future, experiments may be made where the dimension or the number of the nodes vary.

## 5.3 Quasi-optimal versus randomly-split q-kd trees

Now, we are interested in determining when it is worth to use randomly-split q-kd trees and when the quasi-optimal q-kd tree.

For this experiment we used 3 dimensional records, 20000 nodes and 100 runs. The results are shown in *Figure 22* and in *Figure 23*, where the X axes shows the Prob-of-1 is between 0 and 100, and the double value of the values of the Split Tendency (the values are also between 0 and 100).

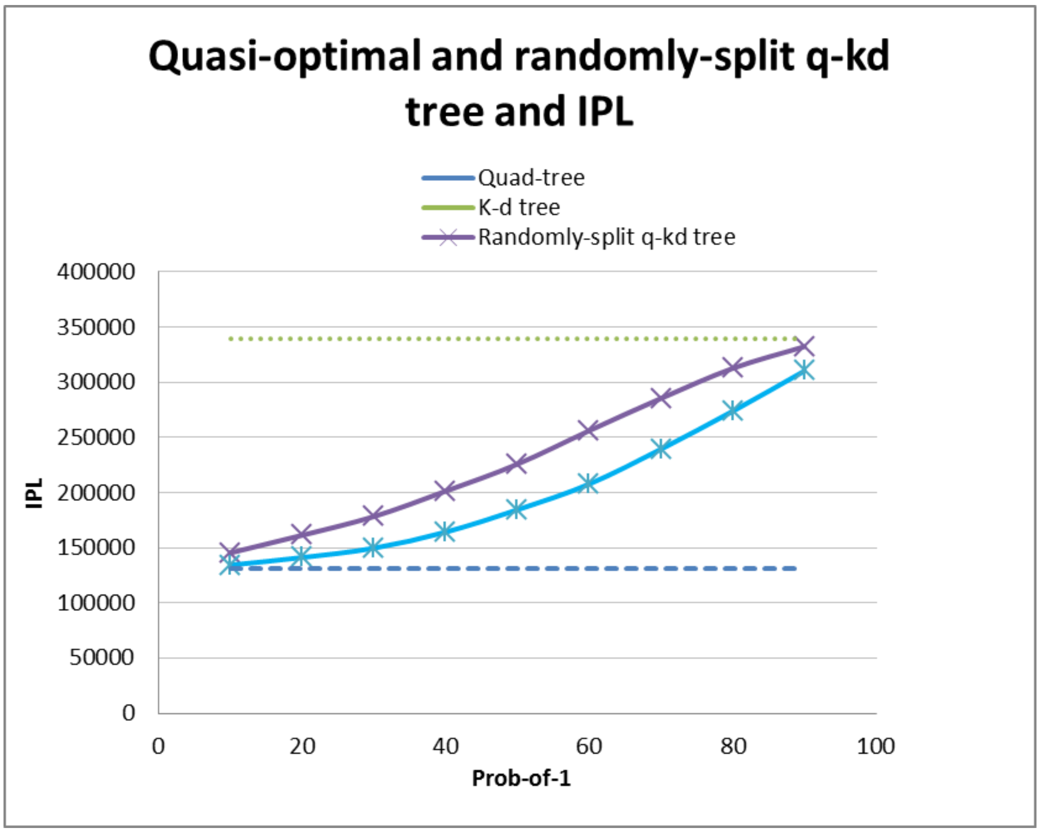

*Figure 22 Quasi-optimal and randomly-split q-kd trees and the IPL*

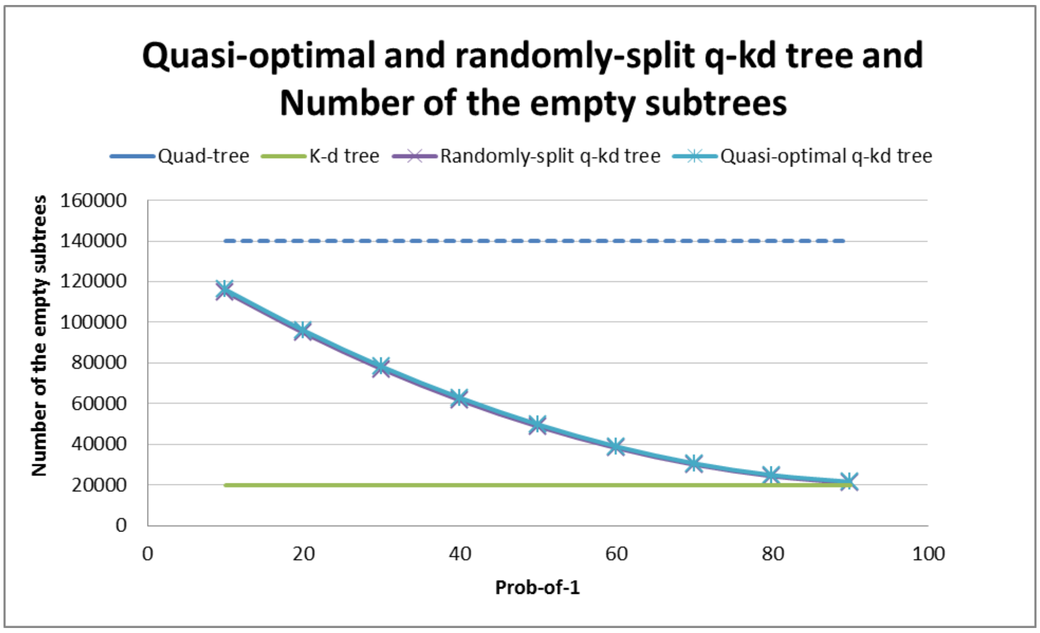

*Figure 23 Quasi-optimal and randomly-split q-kd trees and Number of the empty subtrees*

On one hand, as it can be seen in the diagrams, regarding the IPL, the quasi-optimal q-kd tree is better. On the other hand, concerning the number of the empty subtrees, they are almost identical, as it also can be seen in *Figure 23*. However, if the extra IPL is not a problem for a certain application, then it might be worth to use the randomly-split q-kd tree because the insertion algorithm is simpler.

In the future, experiments may be made where the dimension or the number of the nodes vary.

## 5.4 The effect of the number of the empty subtrees

We want to know how the IPL and the number of the empty subtrees depend on the overall number of the nodes.

For this experiment we used 3 dimensional records, 30 Split Tendency and 100 runs. The results are shown in *Figure 24* and in *Figure 25*:

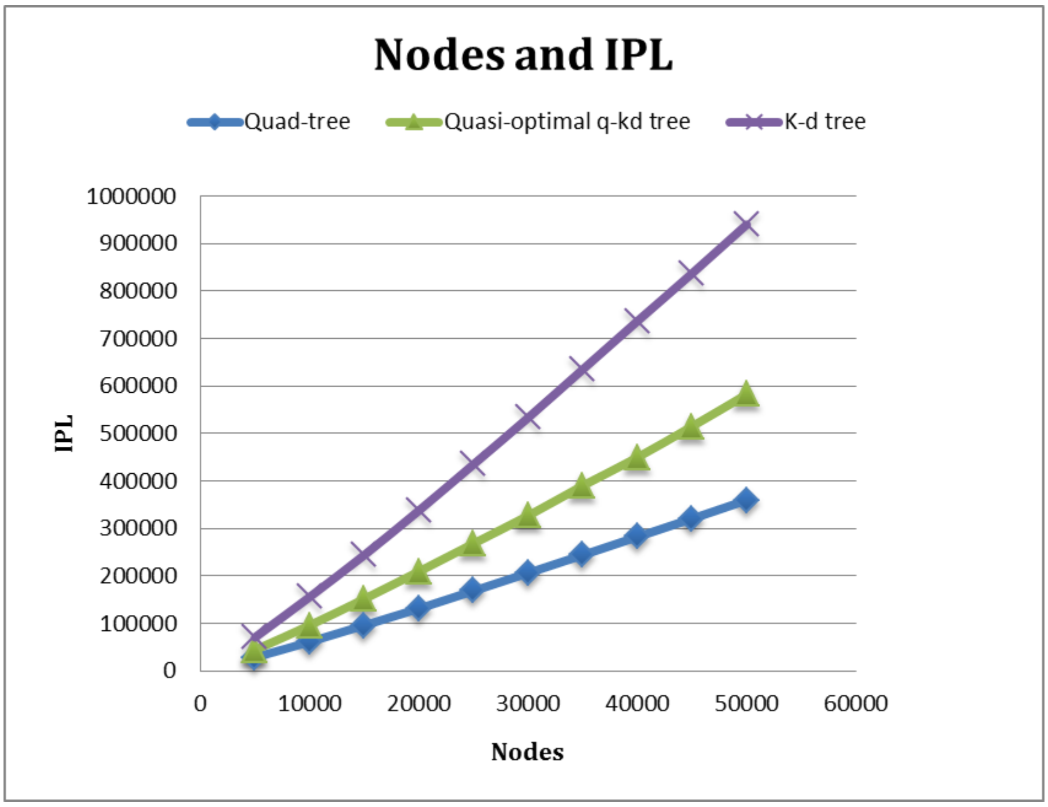

*Figure 24 Nodes and IPL*

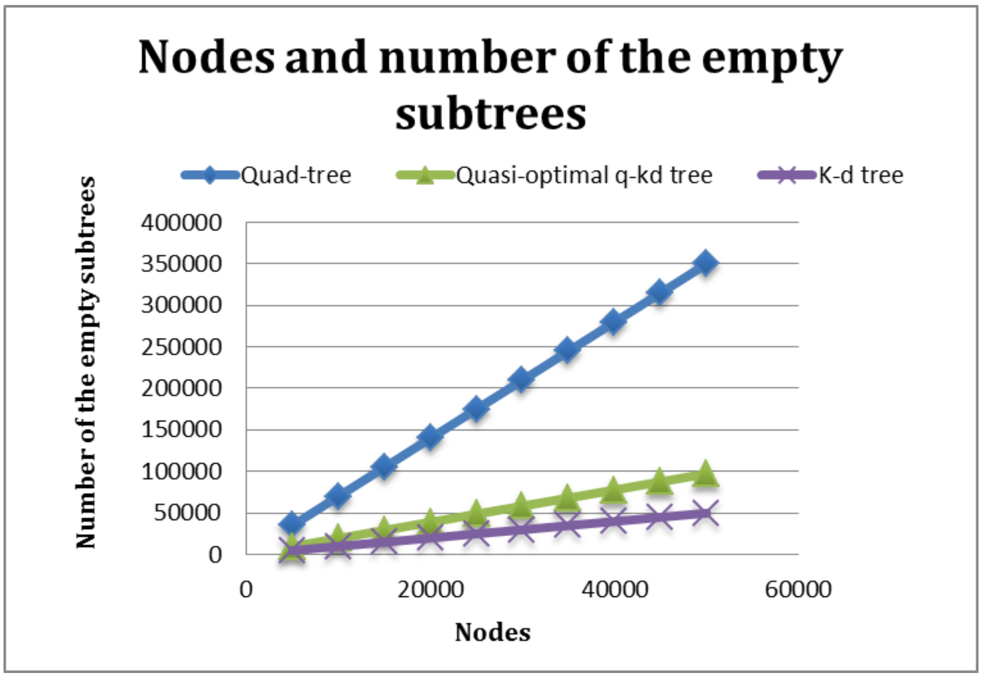

*Figure 25 Nodes and Number of the empty subtrees*

This shows that the IPL and the number of the empty subtrees of all the trees are directly proportional to the number of the nodes and that quasi-optimal q-kd trees seem to be always in the middle.

In the future, experiments may be made where the dimension or the Split Tendency vary.

## 5.5 The effect of the dimension

The next question is how the IPL and the number of the empty subtrees depend on the dimension?

For this experiment we used 30 Split Tendency, 5000 nodes and 100 runs. The results are the following (*Figure 26*, *Figure 27*, *Figure 28*): Note that *Figure 28* is the enlargement of *Figure 27*, without the result for the quad-trees.

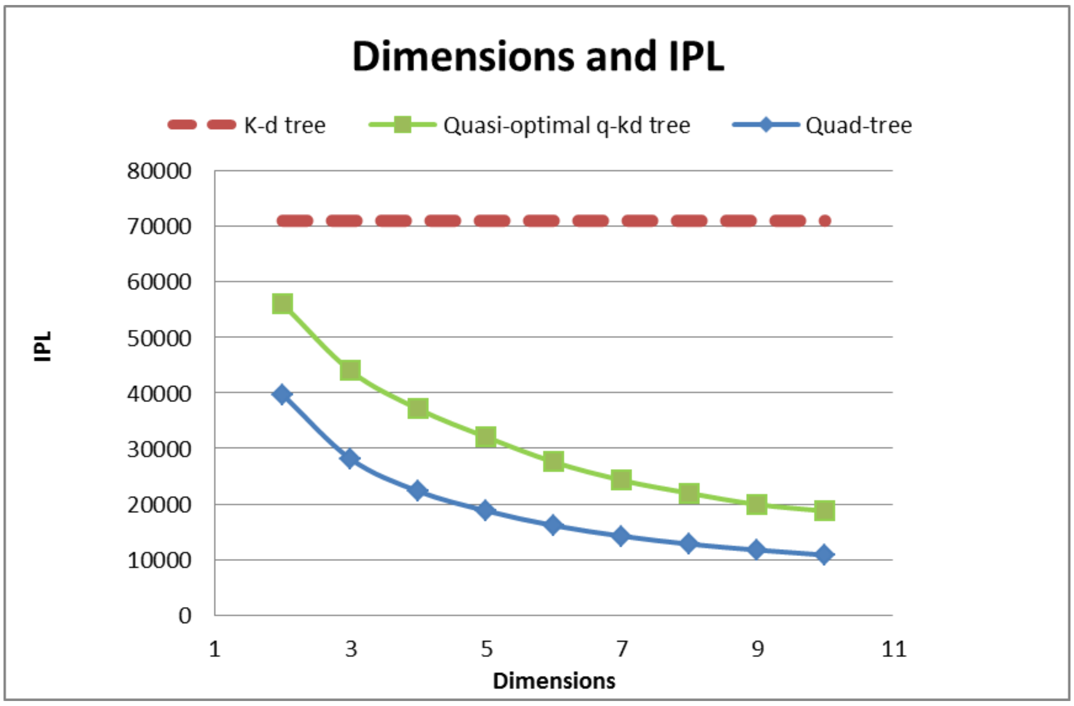

*Figure 26 Dimensions and IPL*

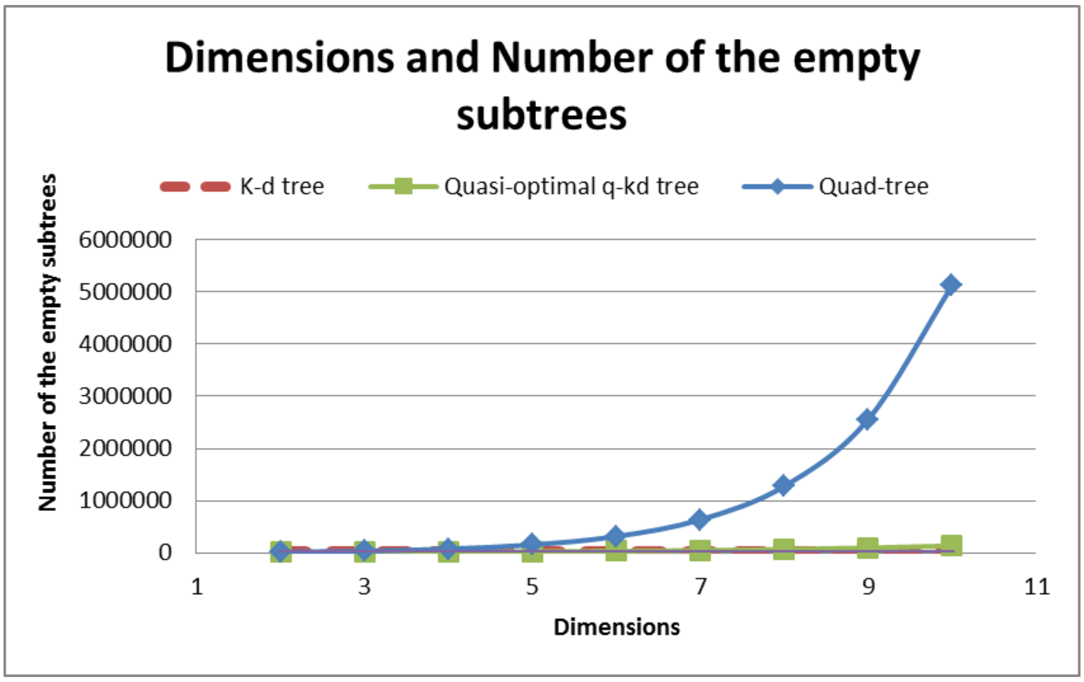

*Figure 27 Dimensions and Number of the empty subtrees*

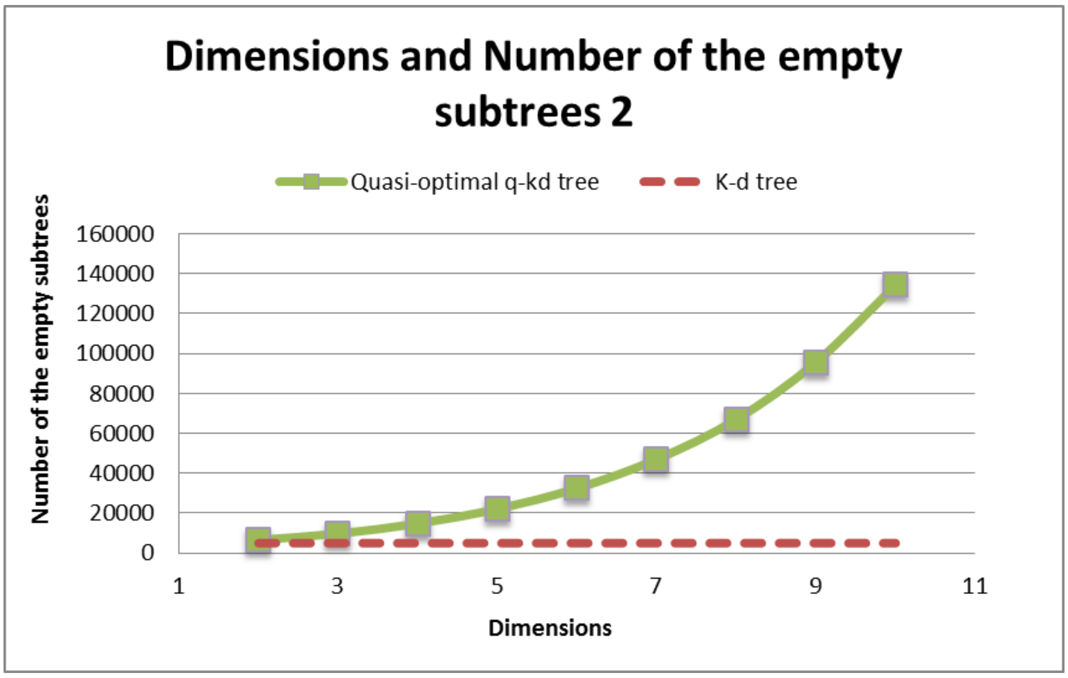

*Figure 28 Dimensions and Number of the empty subtrees 2*

Considering higher dimensions, the IPL of quad-trees and q-kd trees decrease because the splittings create more hyper-rectangulars. But in the case of the quad-trees, as it can be seen in *Figure 27* for higher dimension they need enormous space, so it is not worth using them.

In the future, experiments may be made where the number of the nodes or the Split Tendency vary.

# 6 Summary and plans for the future

This work is concerned with data structures capable of storing and associatively retrieving multidimensional records, such as k-d trees and quad-trees. In particular, a new, flexible data structure is introduced, the q-kd tree, which has k-d trees and quad-trees as particular cases.

After introducing the notion of q-kd trees, two heuristics are proposed: Split Tendency and Prob-of-1 that allow us to build what we call quasi-optimal q-kd trees and randomly-split q-kd trees respectively.

We performed extensive experimentations where we compared these trees with k-d trees and quad-trees using different metrics. It is known that random quad-trees can be, in some sense, considered time optimal, because the expected execution time of update and search operations depend on their expected height, which is the smallest possible. On the other hand they are not optimal considering space, which means that they have a lot of empty subtrees, and these null pointers take a large amount of computer memory. On the contrary, k-d trees are space optimal but not time optimal, since their expected height is greater than the one of the quad-trees and therefore the expected cost of update and search operations may also be greater.

Our experimental results show that the performance of q-kd trees built under the Split Tendency and the Prob-of-1 heuristics is between the one of random quad-trees and the one of random k-d trees considering both Internal Path Length and the number of empty subtrees.

There are several possibilities to go further with this work, among them, we mention the following:

First of all, thinking about a better algorithm for building q-kd trees, for example by considering the nodes that are not inserted yet.

Secondly, it would be useful to try them with points that are non-uniformly distributed: because by their nature q-kd trees are more adaptable than k-d trees and quad-trees and they may have better performance in such settings.

Thirdly, it is also possible to try them with biased associative queries.

Finally, it may be interesting to use q-kd trees as a formal framework to analyze at once all multidimensional trees, since all the other ones are particular cases of them.

# 7 Acknowledgements

Writing this thesis would not have been possible without the guidance of my wonderful supervisors, support from my dear family and my amazing friends.

I would like to thank Amalia Duch Brown for the idea of my thesis, her patience and hard work, for teaching me neatness and for inspiring me towards professional enthusiasm.

I am deeply grateful to Krisztián Németh for ameliorating my programming skills, for his unlimited patience, for keeping me on track of the engineering aspect, his undiminished trust in me and for his sparkling wit.

I would like to show my gratitudes towards my family and friends who supported me in various ways.

I owe my deepest gratitude to our Dearest Lamas, His Holiness the 17th Karmapa Trinley Thaye Dorje and Lama Ole Nydahl, who show us our mind's unlimited potential, and also to the Sangha, our friends on the way all over the world.

# 8 References

[1] Bentley, Jon Louis, *Multidimensional Binary Search Trees Used for Associative Searching*, Communications of the ACM Magazine, Volume 18 Issue 9, Sept. 1975, Pages 509 - 517, ACM New York, NY, USA

[2] Buchin, Kevin, (TU Eindhoven), *Geometric Algorithms*, Lecture 3: Quad-tree, lecture slides, http://www.win.tue.nl/~kbuchin/teaching/2IL55/slides/03meshes-and-quadtrees.pdf

[3] Fan, Wu, *Uses of KD-Trees in Applications*, 2010

[4] Finkel, Raphael and Bentley, Jon Louis (1974), *Quad Trees: A Data Structure for Retrieval on Composite Keys*. *Acta Informatica* 4 (1): 1–9. doi:10.1007/BF00288933

[5] http://en.wikipedia.org/wiki/Quadtree

[6] Mahmoud, Hosam M., *Evolution of Random Search Trees*, Wiley-Interscience series in discrete mathematics and optimization 1991, ISBN: 978-0471532286

[7] Nagy Tímea, *Négyágú-fák*, 2010, Kolozsvár

[8] Samet, Hanan, *Deletion in two-dimensional quad trees*, Communications of the ACM Magazine, Volume 23 Issue 12, Dec. 1980, Pages 703 - 710, ACM New York, NY, USA

[9] Samet, Hanan, *Foundations of Multidimensional and Metric Data Structures*, The Morgan Kaufmann Series in Computer Graphics, 2006, ISBN-13: 978-0123694461

[10] Samet, Hanan, *Multidimensional data structures*, In M. J. Atallah, editor, Handbook of Algorithms and Theory of Computation, Chapter 18, pages 18-1-18-28. CRC Press, Boca Raton, FL, 1999.

[11] Source of the figure: http://www.cs.ru.nl/~ths/rt2/col/h9/quadtree.GIF

[12] Source of the figure: https://wiki.cs.umd.edu/cmsc420/images/2/2e/Pm1_example.png

[13] Tu, Yicheng, CIS6930 Database Systems: Design and Implementation, University of South Florida, lecture notes, http://www.cse.usf.edu/~ytu/teaching/CIS6930_S08/Slides/Quad-tree.ppt

[14] Wolfram Mathworld website: Discrete Mathematics > Graph Theory > Graph Properties > Internal Path Length: http://mathworld.wolfram.com/InternalPathLength.html