DYNAMIC MULTI-DIMENSIONAL DATA STRUCTURES BASED ON QUAD- AND K-D TREES

Mark H. Overmars and Jan van Leeuwen

RUU-CS-80-2

March 1980

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
tel. 030-531454
the Netherlands

DYNAMIC MULTI-DIMENSIONAL DATA STRUCTURES BASED ON QUAD- AND K-D TREES

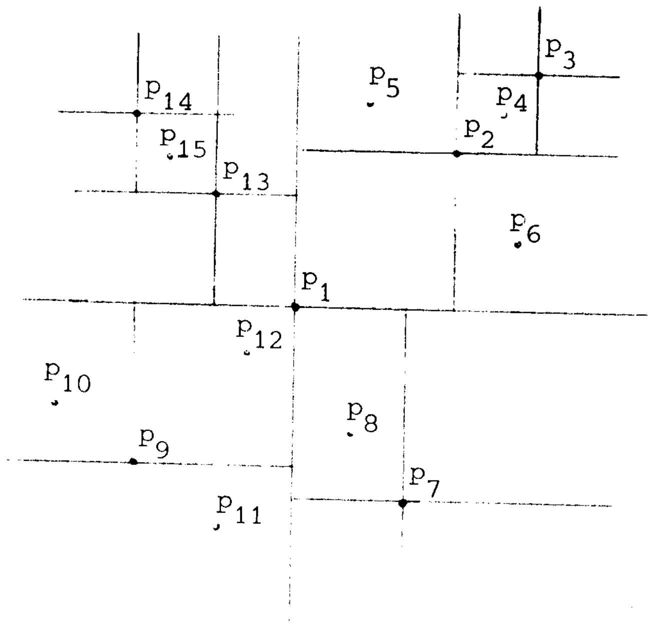
Mark H. Overmars and Jan van Leeuwen

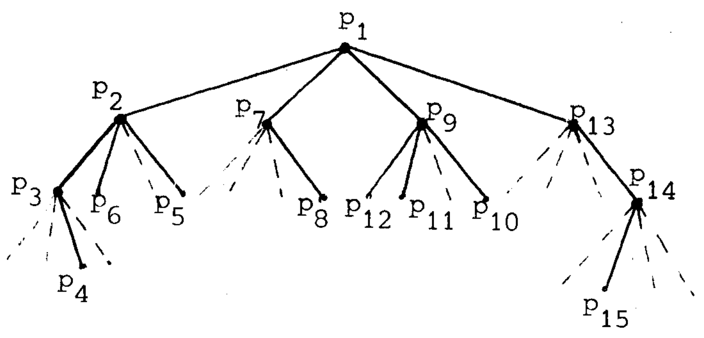
Technical Report RUU-CS-80-2

March 1980

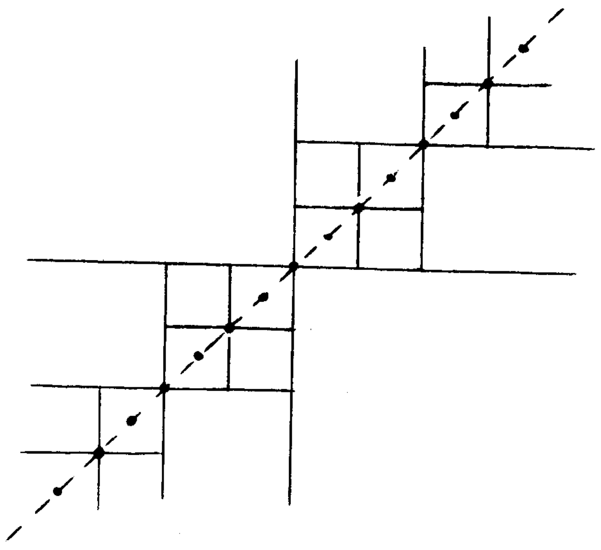
Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

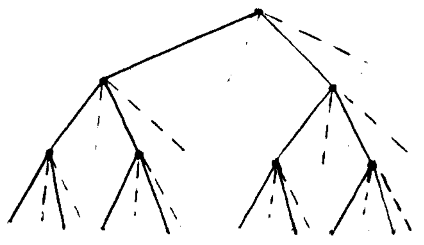
Each quadrant is split again by taking a point of the subset etc. until each quadrant contains at most one point of the set. These final nodes (points) form the leafs of the quad-tree. In the 2-dimensional case we get, for instance





Finkel and Bentley [4] gave several heuristics to build quad-trees as optimal as possible (i.e. with a smallest possible depth), where it is noted that even optimal quad-trees may have a depth of ${}^2\log N$, rather than of ${}^4\log N$, for instance when all points lie on a diagonal line.





Finkel and Bentley [4] showed that it is possible to insert points in quad-trees, but that it is apparently very time consuming to keep the tree optimal. Still they proved an expected time bound of O(log N) per insertion. In section 2 we will show that for every constant $0 < \delta < 1$ it is possible to insert points into a quad-tree in an average time of $O(\frac{d}{\delta} \log^2 N)$ per insertion while keeping

DYNAMIC MULTI-DIMENSIONAL DATA STRUCTURES BASED ON QUAD- AND K-D TREES

Mark H. Overmars and Jan van Leeuwen

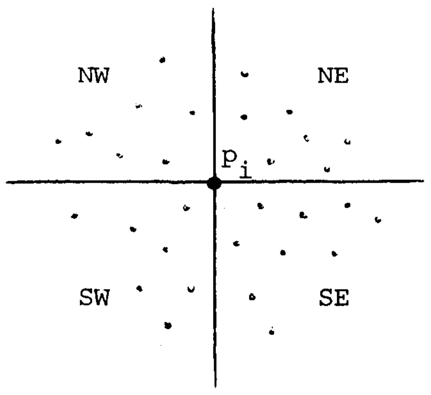
Department of Computer Science, University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht, the Netherlands

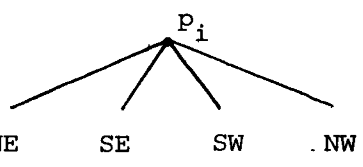
Abstract. We describe a method to insert points in a quad-tree, while keeping the tree balanced, and prove an average time complexity of $O(\log^2 N)$ per insertion for it, where N is the number of points in the quad-tree. We define a structure similar to a quad-tree, called a pseudo quad-tree, and show how this structure can be used to handle both insertions and deletions in $O(\log^2 N)$ average time. We also discuss how quad-trees and pseudo quad-trees can be extended to so-called EPQ-trees for use in configurations of points with more than a constant number of points having a same coordinate, without altering the earlier time bounds for insertions, deletions and queries. We develop similar algorithms for k-d trees and obtain the same average time bounds for insertions and deletions in such structures.

Keywords and phrases: multi-dimensional searching, insertion, deletion, quad-tree, pseudo quad-tree, k-d tree, pseudo k-d tree, EPQ-tree.

1. Introduction.

Quad-trees were introduced by Finkel and Bentley [4] as a suitable data-structure for answering queries about sets of points in multidimensional space (viz. range queries). Let a set of points $S = \{p_1, \ldots, p_N\}$ be given and let the dimension of the space be d. A quad-tree of S is built in the following way. One of the points p_i of the set is taken as the root of the quad-tree. It divides the space into 2^d quadrants, and therefore splits the set into 2^d subsets. These 2^d subsets (quadrants) will be the sons of p_i in the tree. So, when d = 2, we get





Throughout this paper we mean by average insertion time or average deletion time the average time needed to perform N updates in an initially empty structure. (Hence one should not confuse average time with expected time.)

2. Insertions and deletions in (pseudo) quad-trees.

In this section we shall show a technique to perform insertion in quad-tree, while keeping the structure balanced. By slightly changing the structure we will be able to process deletions also. We assume throughout this section that no more than a constant number of points have a same coordinate.

2.1. Insertions in quad-trees.

Inserting a point in a quad-tree itself is no problem. We just locate the subquadrant it is in by a simple search on the tree. If there was no point present we insert it there. Otherwise we use the old point as a splitting point and insert the new point in the appropriate sub-sub quadrant. The problem is how to keep the tree balanced, how to keep it as optimal as possible. The best we may hope to achieve is a depth of about ${}^{2}\log N$ (as pointed out in the introduction) but in this case each insertion would require a complete rebalancing of the tree, which is a lot of work. Hence, to obtain fast insertion times (on the average) we have to make some concessions on the optimality (i.e., the depth) of the quad-tree.

Theorem 2.1.1. For any fixed δ with $0 < \delta < 1$ there is a way to perform N insertions into an initially empty quad-tree such that its depth is always at most ${}^{2-\delta}\log n$ (n is the current number of points in the structure) and the average transaction time is bounded by $\frac{d}{\delta}\log^2 N$.

Proof

To achieve the depth bound of ${}^{2-\delta}\log n$ we put an even stronger condition on the tree, namely that for every internal node, except on the lowest level, with a total of k points in its joint subtrees, every subtree contains at most $\frac{1}{2-\delta}$ k points. Such a tree always has a depth of at most ${}^{2-\delta}\log n$. Our method of insertion will make use of the fact that for every configuration of n points one can build a quad-tree, such that for every internal node with k points in its joint subtrees, every subtree contains at most $\frac{1}{2}k$ points, in only O(d n log n) time. This can be done by repeatedly splitting each subquadrant w.r.t.

the depth to at most ${}^{2-\delta}\log N$. So, at the cost of only a small loss in optimality one can perform insertions in quad-trees efficiently. Deletions in quad-trees are very hard to process, and very time consuming. In fact it has repeatedly been used as a reason for rejecting quad-trees as a sufficiently flexible data structure (see also Samet [9]). In section 2 we modify the quad-tree to a pseudo quad-tree, which can handle queries in the same way but which has better prospectives for updates. Before showing how to perform insertions and deletions in pseudo quad-trees, we develop a method for building a pseudo quad-tree of a set of N points in d-dimensional space with depth at most ${}^{d+1}\log N$, in $O(d.N \log N)$ time. We also prove that there are configurations of points for which this depth bound is optimal. The main objective in developing the pseudo quad-trees, however, is the possibility of performing both insertions and deletions on it. We prove that for every constant $0 < \delta < d$ an average time of $O(\frac{d^3}{\delta} \log^2 N)$ per insertion and deletion can be achieved, while the depth of the (d-dimensional) pseudo quad-tree is kept to at most ${}^{d+1-\delta}\log N$.

When we work with quad-trees or pseudo quad-trees, we normally have to assume that no more than a constant number of points have equal coordinates, i.e., that there is a constant c such that for every x and for every $0 < i \leq d$ there are at most c points with i'th coordinate equal to x. This condition is needed, because it may be impossible otherwise to choose a point that suitably divides the points in some way over the quadrants. In section 3 we show how some extra sons can be added to each node in a quad-tree or pseudo quad-tree to circumvent this condition as a restriction of generality. We show that this extension of the structures does not increase the insertion and deletion time and that, in general, it does not increase the query times.

Bentley [1] presented another data structure for answering queries about more-dimensional pointsets, the k-d tree. He showed that insertion in k-d trees is possible, but that it is again time consuming to keep the tree balanced, and that deletions can be processed also but that they are often even harder to perform. In section 4 we show how insertions in k-d trees can be processed in average time $O(\frac{1}{\delta} \log^2 N)$, while the depth is kept to at most ${}^{2-\delta}\log N$ for any constant $0 < \delta < 1$. To be able to process deletions also, we again define a modification to so called pseudo k-d trees. (The idea behind it was previously introduced by Willard [10].) We prove for pseudo k-d trees an average insertion and deletion time of $O(\frac{1}{\delta} \log^2 N)$, while the depth is kept to at most ${}^{2-\delta}\log N$ for any constant $0 < \delta < 1$. We arque that all results obtained for quad- and psuedo quad-trees carry over in almost the same way to k-d and pseudo k-d trees. In section 5 we offer some concluding remarks.

So a transaction is charged at most O(log N) times, and the charge is always at most $O(\frac{d}{\delta} \log N)$ at a time. Hence the total average transaction time is bounded by $O(\frac{d}{\delta} \log^2 N)$.

□

So it is possible to perform insertions in quad-trees efficiently, when we allow a small increase in depth. In general, the average time needed for insertions is likely to be much smaller than $O(\frac{d}{\delta} \log^2 N)$ because, when subtrees of internal nodes expand equally fast, we have no need for any rebalancing at all. Although we can process insertions in quad-trees efficiently, deletions remain a problem. In the next two subsections we will show how, by changing the structure slightly, we can also obtain an efficient deletion method.

2.2. Pseudo quad-trees.

As stated before, it is very hard to perform deletions in ordinary quad-trees. This is so primarily because almost every point also has a dividing function. Points are used for splitting quadrants into subquadrants and thus for splitting parts of the set. When we delete a point p, we would like to "replace" it by a new dividing point, but it is quite possible that any other point would split the set in a very different way. Finkel and Bentley [4] therefore suggest to reinsert all points in the subtrees of p that are affected. This may take a lot of time, e.g. when we delete the root of the quadtree, but in the expected case it may be quite efficient. Samet [9] shows how, in the two-dimensional case, this number of reinsertions can be decreased by choosing the new dividing point in a very special way. But his technique may still take much time in the worst case.

As the problems with deletion arise from the dividing function of the points, it would seem obvious to try and eliminate the dividing function of points. This can be done by allowing arbitrary points to be used for this function instead of points of the set only. Our new structure, which we shall call a pseudo quad-tree, is based on this idea. It is very similar to an ordinary quad-tree, except that the internal nodes that split the space (and therefore the set), no longer are points of the set itself. We use arbitrary points to split the space into quadrants, the quadrants into subquadrants and so on, until every subquadrant contains at most one point of the set. The points of the set thus occur as the leaves of the pseudo quad-tree. E.g. in the two-dimensional case we get a structure as shown in the following diagram. (p_1, ..., p_{12} are the points from the set h_1, ..., h_5 are the arbitrarily chosen points.)

a point whose coordinate is the median along a chosen axis. When we want to insert a point p, we first determine the subquadrant p is in. When there was no point present in that subquadrant, we just insert p there, otherwise we use the one old point present as a splitting point and insert p in the appropriate sub-subquadrant just created. It is very well possible that now somewhere in the tree the balance is disturbed. This can only happen on the path from the root to the newly inserted point. If this occurs, then we determine the highest internal node h for which this is the case, and rebuild the complete subtree below it as a perfectly balanced tree, in the way described above. To obtain a bound on the resulting average transaction time, let us look at some internal node h with k points in its joint subtrees at the moment it is built. Because of the way we build trees, every subtree attached to h contains at most ½k points. We need to rebalance at h only by the time one of its subtrees contains $\frac{1}{2-\delta}$ times the total number of points below h at that moment. If this happens after i insertions into the subtree at h (since the latest rebalancing, or creation, of h) then necessarily:

$$\begin{aligned} &\tfrac{1}{2}k + i \geq \frac{1}{2-\delta}(k+i) && \Rightarrow \\ &i - \frac{1}{2-\delta}\, i \geq \frac{1}{2-\delta}\, k - \tfrac{1}{2}k && \Rightarrow \\ &\frac{1-\delta}{2-\delta}\, i \geq \frac{\delta}{4-2\delta}\, k && \Rightarrow \\ &i \geq \frac{\delta}{2-2\delta}\, k \end{aligned}$$

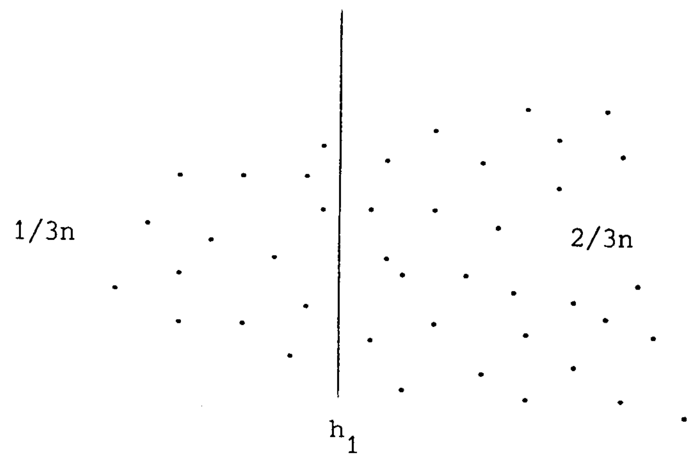
The total cost for the rebuilding of the subtree at h, we need to perform for rebalancing, is d(i+k) log(i+k). When we divide these cost evenly over the i insertions in the subtree at h which took place since the last rebuilding, this makes for

$$\frac{d.(i+k)\ \log(i+k)}{i} =$$

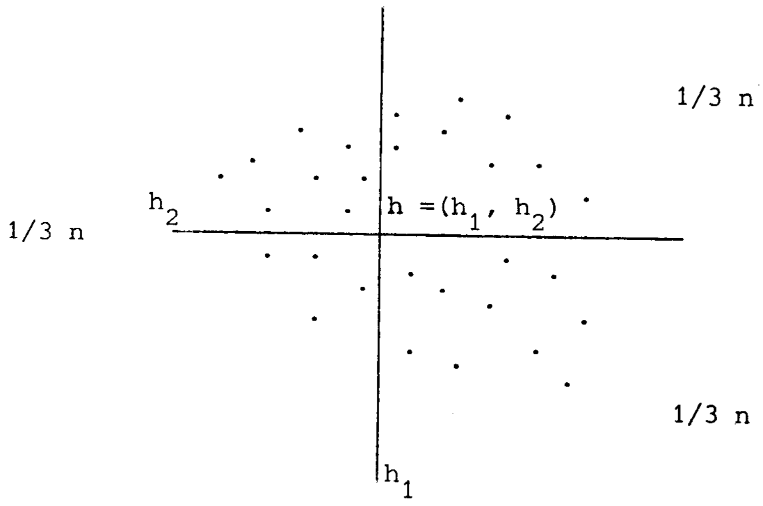
$$d.\log(i+k) + d.\frac{k}{i}\log(i+k) \leq$$

$$d.\log(i+k) + d.\frac{2-2\delta}{\delta}\log(i+k)$$

per transaction. Because δ is a positive constant this is $O(\frac{d}{\delta}\log(i+k))$ and this is at most $O(\frac{d}{\delta}\log N)$. Note that the costs are charged only to insertions performed in the subtree at h since its latest rebuilding. Hence, no costs can be charged anymore to these transactions for rebalancing at the level of h, or at a lower level. (Every internal node below h is also rebalanced by the rebuilding of the subtree at h.) Only internal nodes at higher levels (and on the path from the root to h) can charge more costs to these insertions. It follows that each insertion can be charged at most once from every level. The depth of the quad-tree is at most ${}^{2-\delta}\log n = {}^{2-\delta}\log N = \frac{\log N}{\log(2-\delta)}$. Because δ is a constant < 1 this is O(log N).



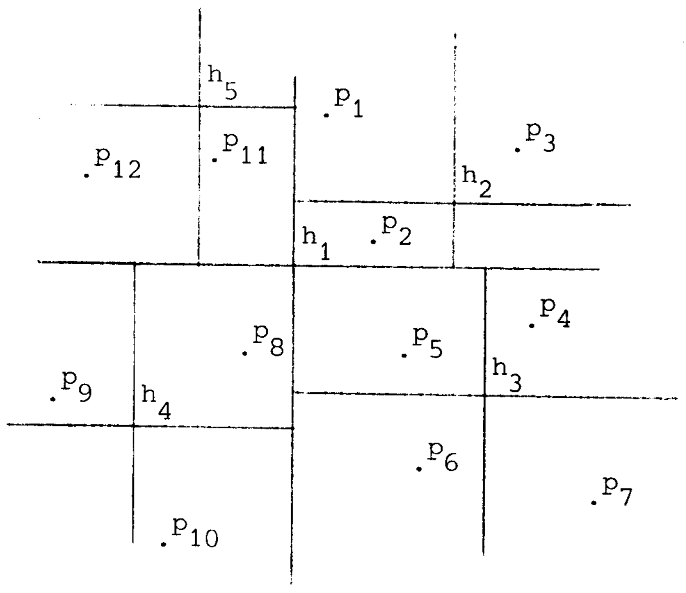
There still are d-1 coordinates of h left to determine. To find the proper h_2-value look only at the points with x_1-coordinate bigger than h_1. (The other part of the set already is small enough and it is not important how this part is split up by any further hyperplanes.) So we have a set of $\frac{d}{d+1}$ n points to split. Choose h_2 such that $\frac{1}{d+1}$ n of these points have x_2-coordinate smaller than h_2 and the other $\frac{d-1}{d+1}$ n points have x_2-coordinate bigger than h_2. We continue with the portion of $\frac{d-1}{d+1}$ n points and choose the appropriate h_3 value in the same way, etc. By the time we have chosen the appropriate h_{d-1} value we are left with a subset of $\frac{2}{d+1}$ n points. We can easily choose a h_d value that splits this remaining subset in two parts of $\frac{1}{d+1}$ n points each, to end the process. So we have obtained a point $h = (h_1, \ldots, h_d)$ that splits the space in quadrants, each containing at most $\frac{1}{d+1}$ n points. E.g. in the two-dimensional case we would get the following diagram.

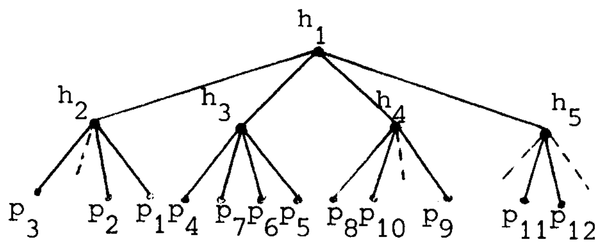


$\square$

Using this theorem we can obtain

<u>Corollary</u> 2.2.1. Given a set of n points in d-dimensional space, there exists a pseudo quad-tree for it with depth at most ${}^{d+1}\log n$.





h_1
h_2
h_3
h_4
h_5
p_3 p_2 p_1 p_4 p_7 p_6 p_5 p_8 p_{10} p_9 p_{11} p_{12}

Because pseudo quad-trees are very much like ordinary quad-trees, all kinds of queries that could be answered from quad-trees, can be answered using pseudo quad-trees in a very similar way. In addition to the fact that in pseudo quad-trees both insertions and deletions can be processed efficiently, as we will show in section 2.3., it is also interesting to note that for a same set of points we can also build pseudo quad-trees that have a smaller depth than the corresponding quad-trees of the unmodified sort. This results from the fact that we may choose every splitting point we like and that we can take the point that splits the set in the smallest parts at any stage of the construction.

Theorem 2.2.1. Given a set of n points $p_1, \ldots, p_n$ in d-dimensional space, there exists a splitting point $h = (h_1, \ldots, h_d)$ such that every quadrant induced by h contains at most $\frac{1}{d+1}$ n points.

Proof.

Choose h_1 such that (up to a constant) $\frac{1}{d+1}$ n points of the set have x_1-coordinate smaller than h_1 and the other $\frac{d}{d+1}$ n points have x_1-coordinate bigger than h_1. Such a h_1 exists because of our assumption that no more than a constant number of points have a same coordinate. We have in fact constructed a hyperplane (of the points with x_1 coordinate equal to h_1) that separates the set in one part with $\frac{1}{d+1}$ n points and another part with $\frac{d}{d+1}$ n points. E.g. in the two-dimensional case it would be a line that divides the set in two subsets of 1/3n and 2/3n points, respectively.

Merely knowing that such an "optimal" pseudo quad-tree exists is not enough, we also have to be able to build it efficiently.

Theorem 2.2.3. Given a set of n points in d-dimensional space, a pseudo quad-tree of depth at most ${}^{d+1}\log n$ for these points can be built in O(d.n log n) time.

Proof

We will use the method of theorem 2.2.1. and corollary 2.2.1. Determining the appropriate splitting point h in the proof of theorem 2.2.1. consists of d times finding a ranked element of the set, with respect to some ordening, which takes O(n) each time. Also the splitting of the set into the different quadrants takes O(d.n). So building the first level of the pseudo quad-tree takes O(d.n). The splitting of a quadrant that contains k points takes O(d.k) in the same way. The quadrants together contain n points, hence the total cost for building the second level of the tree is O(d.n) again. The same argument holds for every level. Since the depth of the tree is at most ${}^{d+1}\log n$, the bound follows.

□

It follows that we can build pseudo quad-trees of lesser depth than quad-trees (which could have depth ${}^{2}\log n$) efficiently. In general the time needed for a query on a quad-tree depends on the depth of the tree. Because these queries can be performed on pseudo quad-trees in a similar way and pseudo quad-trees are of lesser depth, in general, the query times will decrease.

2.3. Insertions and deletions in pseudo quad-trees.

The main objective for constructing pseudo quad-trees is the possibility to handle both insertions and deletions efficiently. (See Overmars and van Leeuwen [8] for another way of dynamizing pseudo quad-trees, based on a general approach to dynamization.) As said earlier, efficient insertion and deletion routines may be obtained only by weakening the strict optimality of the structure. In the case of pseudo quad-trees, we cannot keep the depth of a pseudo quad-tree of n points to ${}^{d+1}\log n$ (as achieved by theorem 2.2.3.), but we can stay close to it.

Theorem 2.3.1. For any fixed δ with $0 < \delta < d$ there is a way to perform N insertions and deletions in an initially empty pseudo quad-tree such that its depth is always at most ${}^{d+1-\delta}\log n$ (where n is the current number of points in the pseudo quad-tree)

Proof

According to theorem 2.2.1. there exists a point h that divides the space into quadrants containing at most $\frac{1}{d+1}$ n points each. Use h as the root of the pseudo quad-tree and proceed. Again by theorem 2.2.1., in each quadrant there exists a point that splits the k points in that quadrant into parts with at most $\frac{1}{d+1}$ k points each. These points form the sons of h. Note that every subquadrant contains at most $\frac{1}{(d+1)^2}$ n points. Repeatedly applying theorem 2.2.1. to the subquadrants, and using the splitting points as internal nodes, gives us a pseudo quad-tree with depth at most ${}^{d+1}\log n$.

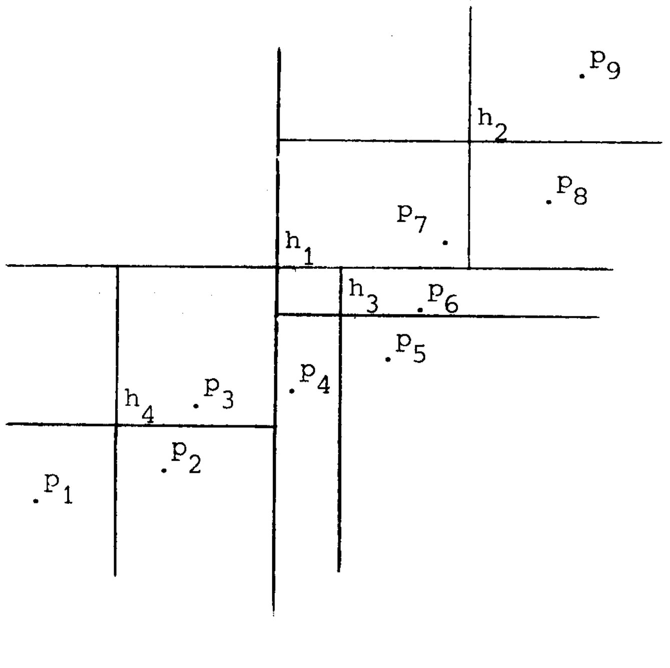
□

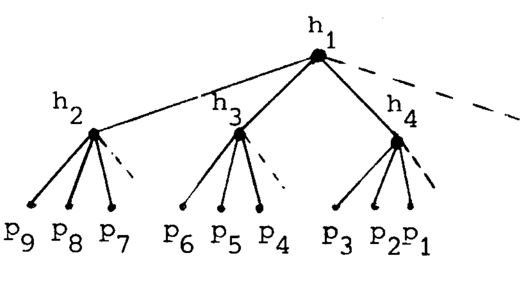
This result is optimal, because we have the following

Theorem 2.2.2. There exists sets of n points $p_1, \ldots, p_n$ in d-dimensional space such that no pseudo quad-tree for it can have depth less than ${}^{d+1}\log n$.

Proof

Take $p_1 = (x_1^1, \ldots, x_d^1)$, $p_2 = (x_1^2, \ldots, x_d^2)$, ..., $p_n = (x_1^n, \ldots, x_d^n)$ such that for every $1 \le i < n$ and $1 \le j \le d$: $x_j^i < x_j^{i+1}$. Note that in such a configuration no two points have a same coordinate. When we walk from p_1 to p_n, we can change quadrant at most d times. Therefore the points can be divided over at most d+1 quadrants. So there must be one quadrant that contains at least $\frac{1}{d+1}$ n points. The points in this quadrant have the same property again as the set of points originally chosen and therefore, by the same argument, there must be a subquadrant that contains at least $\frac{1}{d+1}$ of these remaining points. Repeating this argument shows that the pseudo quad-tree must have a depth of at least ${}^{d+1}\log n$. In the two-dimensional case we have, for instance,





□

$$\frac{d.(k+i-m)\ \log(k+i-m)}{i+m} =$$

$$d.\left(\frac{i-m}{i+m}\log(k+i-m) + \frac{k}{i+m}\log(k+i-m)\right) \leq$$

$$d.\left(\log(k+i-m) + \frac{(d+1)(d+1-\delta)}{\delta}\log(k+i-m)\right)$$

per transaction. Because $k + i - m \leq N$ this is bounded by $O(\frac{d^3}{\delta}\log N)$. By the same arguments as in the proof of the theorem 2.1.1. one can show that a transaction is charged at at most once at each level. The depth of the pseudo quad-tree is at most ${}^{d+1-\delta}\log n \leq {}^{d+1-\delta}\log N = \frac{\log N}{\log(d+1-\delta)}$. Because $\delta < d$ this is $O(\log N)$. Hence the average time per transaction is bounded by $O(\frac{d^3}{\delta}\log^2 N)$.

□

In general the average time needed for transactions is likely to be much smaller than $O(\frac{d^3}{\delta}\log^2 N)$ because a) the transaction time depends only on the maximum number of points in the set at any moment and b) when subtrees of internal nodes expand or shrink equally fast, we have no need for any rebalancing at all.

From theorem 2.3.1. it follows that a pseudo quad-tree is a more powerful structure then an ordinary quad-tree. Still there is one problem left, namely the assumption, made in this section, that no more than a constant number of points have a same coordinate. In the next section we shall remedy this deficiency by defining an extended pseudo quad-tree (or EPQ-tree), which is able to handle configurations without any such restriction.

3. EPQ-trees.

In section 2 we have developed a fully dynamic multi-dimensional data structure based on a quad-tree, but we made the assumption that no more than a constant number of points had a same coordinate. This was necessary because otherwise it would not always be possible to choose a splitting point that splits the set in the desired fractions. All points could, for instance, lie on a same vertical line. In this case any splitting point would divide the points over at most 2 quadrants. (Theorem 2.2.1. makes essential use of the assumption that this doesn't happen.) In this section we will show how pseudo quad-trees can be modified to so-called EPQ-trees (Extended Pseudo Quad-trees), in order that we can drop the restriction on the points of the set. We will show that queries in an EPQ-tree are, in general, of the same efficiency as in ordinary (pseudo) quad-trees. Also insertions and deletions can be processed fast in EPQ-trees.

and the average transaction time is bounded by $\frac{d^3}{\delta} \cdot \log^2 N$.

Proof

The proof of this theorem is very similar to the proof of theorem 2.1.1. To achieve the depth bound of ${}^{d+1-\delta}\log n$, we again put the stronger condition on the tree, with k points of the set in its joint subtrees, every subtree attached to it contains at most $\frac{1}{d+1-\delta}k$ points. Note that the building method of theorem 2.2.3. delivers a tree such that every internal node h with k points in its joint subtrees has at most $\frac{1}{d+1}$ k points in each subtree attached to it. Whe we want to insert a point p we first determine the subquadrant it is in. If there was no point present we just insert it there. Otherwise, if there was a point p' already, then we take the midpoint of $\overline{pp'}$ as a splitting point and insert p and p' in the appropriate (different) sub-subquadrants. If we want to delete a point p we locate it in the tree and throw it away. (This is possible because it is a leaf.) It is very well possible that the insertion or deletion disturbed the balance somewhere in the tree. This can only occur at a node on the path from the inserted or deleted point towards the root of the tree. To rebalance, take the highest internal node h that is out of balance and rebuild the complete subtree at h using the method of theorem 2.2.3. To obtain an average transaction time,let us again look at some internal node h with k points below it. When the subtree at h is built, each subtree attached to h contains at most $\frac{1}{d+1}$ k points of the set. By the time we need to rebalance at h, one of its subtrees contains $\frac{1}{d+1-\delta}$ of the points below h. Let this happen after there have been i insertions and m deletions in the subtree at h (after h was built). Then necessarily

$$\frac{1}{d+1} k + i \geq \frac{1}{d+1-\delta}(k + i - m) \qquad \Rightarrow$$

$$i - \frac{1}{d+1-\delta}i + \frac{1}{d+1-\delta} m \geq \left(\frac{1}{d+1-\delta} - \frac{1}{d+1}\right)k \quad \Rightarrow$$

$$\frac{d-\delta}{d-\delta+1}\, i + \frac{1}{d+1-\delta}\, m \geq \frac{\delta}{(d+1)(d+1-\delta)}k$$

hence because $\frac{d-\delta}{d-\delta+1} < 1$ and $\frac{1}{d+1-\delta} < 1$,

$$\Rightarrow i + m \geq \frac{\delta}{(d+1)(d+1-\delta)}\ k$$

The total cost for the rebuilding at h is d.(k+i-m) log(k+i-m). When we divide these costs evenly over the i + m transactions in the subtree at h since its last rebuilding this makes for have x_1 coordinate equal to h_1. These points, together with the points that had some other coordinate equal to h, have to be divided over the appropriate associated structures at h. Let $p = (p_1, \ldots, p_i, \ldots, p_d)$ be such a point. The associated structure p belongs to is fully determined by which coordinates p_i of p are equal to the corresponding coordinate h_i of h. This can be tested in O(d). After the splitting point h is located, which takes O(n), we can determine for every point p to which quadrant or associated structure p belongs in O(d) steps. Hence the total cost for splitting the set at h is O(d.n). Splitting the quadrants and building the associated structures continues in the same way. Thus the building of every "level" in the total structure takes O(d.n). One easily sees that, after the splitting each quadrant contains at most one half of the points. It is possible that more than half of the points went into an associated structure, but this can happen at most d times (everytime the dimension decreases with at least one). Hence the depth of the total structure is at most log n + d. Hence, building the structure takes at most O(d.n.(log n + d)). Because d is fixed we have d < log n for n sufficiently large and the time bound is O(d.n log n).

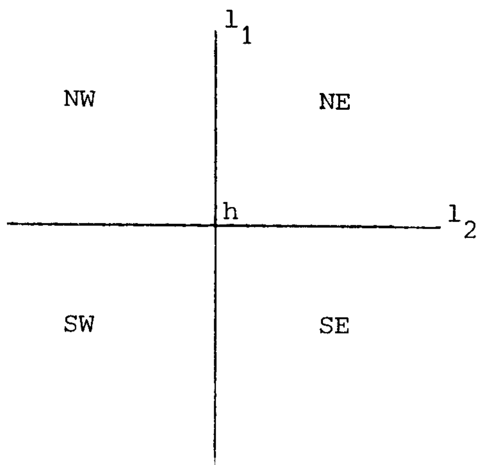
□

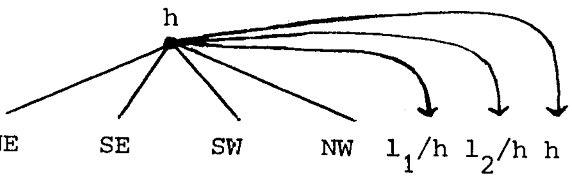
Clearly, in general we may be able to build far more optimal structures, but when, for instance, all points lie on a vertical line, we can achieve nothing better than a depth of log n + d.

Performing queries on an EPQ-tree is a little more complex than it is for pseudo quad-trees but often is of the same efficiency. We will demonstrate this by the example of range queries. A range query asks for all points $x = (x_1, \ldots, x_d)$ of the set such that $a_1 \le x_1 \le b_1$ and ... and $a_d \le x_d \le b_d$ for specified values of a_i and b_i $(1 \le i \le d)$. When we want to perform a range query on an EPQ-tree we start at the root h of the tree. By comparing h with the range given we determine the quadrants the range lies in. But instead of continuing in these quadrants we also have to determine the associated subspaces the range cuts "through", and we must perform a range query on each of these associated structures, using as a range the restriction of the original range to the subspaces. When the configurations satisfy the restriction for ordinary (pseudo) quad-trees then the subspaces contain only constant number of points and therefore the time needed for a query will be the same as for ordinary (pseudo) quad-trees. When the configurations do not satisfy the restriction then also the search of the subspaces takes time, but, because the number of subspaces to be searched is smaller then the largest number of quadrants that might have to be searched, the total time needed will be of the same order as it was for (pseudo) quad-trees (see e.g. Bentley and Stanat [3]).

3.1. The EPQ-structure.

A pseudo quad-tree is based on a way of splitting the d-dimensional space to which the pointset belongs. This is done by choosing an arbitrary point h and dividing the point set over the 2^d quadrants defined by h. These quadrants have hyperplanes in common. Using the original restriction that only a constant number of points have a same coordinate, this is no problem, but when this restriction is dropped, this must be changed. Therefore we cut the hyperplanes off from the quadrants and treat them separately together with the "open" quadrants. The d hyperplanes have (d-2)- dimensional subspaces in common. We again cut these off from the hyperplanes. The resulting subspaces have (d-3)-dimensional subspaces in common, we cut these off, and so on. In this way we get 2^d quadrants, d (d-1)-dimensional subspaces, $\binom{d}{2}$(d-2)- dimensional subspaces, ..., $\binom{d}{d-1}$ lines and 1 point. All these subspaces are associated to the internal node h and separately organized as lower dimensional EPQ-trees. We define a 1-dimensional EPQ-tree to be a (balanced) binary search tree and a 0-dimensional EPQ-tree (a point h) to be an integer that gives the number of points coinciding with h. The total number of these associated structures excluding the ordinary quadrants is $\binom{d}{1} + \binom{d}{2} + \ldots + \binom{d}{d-1} + \binom{d}{d} = 2^d-1$. E.g., in the 2-dimensional case to all internal nodes are added two lines and one point, i.e., two 1-dimensional EPQ-trees and one 0-dimensional EPQ-tree.



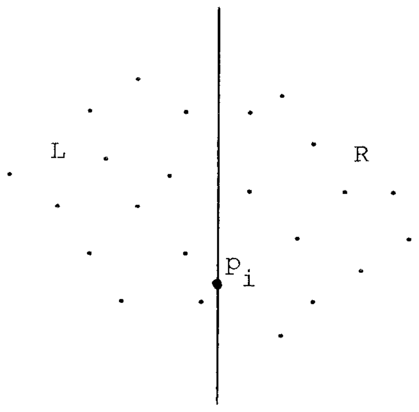


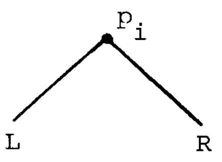
Theorem 3.1.1. Given a set of n points in d-dimensional space, one can build an EPQ-tree of the points with depth $\le \log n + d$ in $O(d\, n \log n)$.

Proof

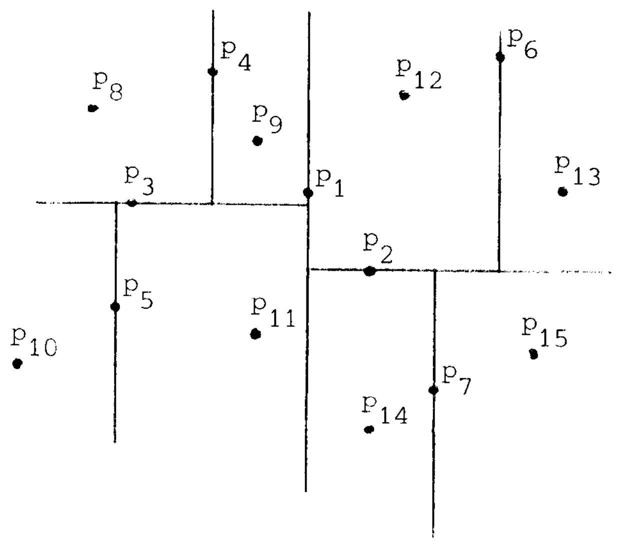
As the splitting point we choose a point $h = (h_1, \ldots h_d)$ (not necessarily of the set) such that at most half of the points have x_1 coordinate strictly less than h_1 and at most half of the points have x_1 coordinate strictly greater than h_1. Such a point h exists but it may be that many points are left which

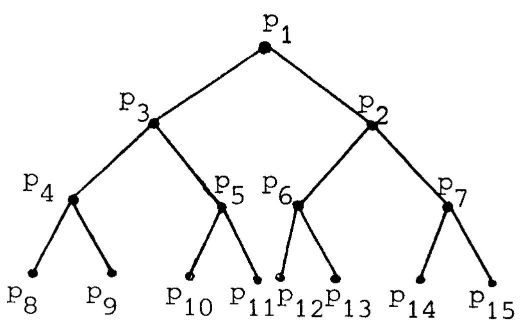
$\{p_1, \ldots, p_n\}$ take a point p_i. It splits the set in two parts, according to its first coordinate. The point p_i is made the root of the k-d tree, the two subsets will be its sons. E.g. in the two-dimensional case we get





In both subsets we take a point again and split the subsets w.r.t. the second coordinate. Likewise we split w.r.t. the third coordinate etc. After splitting w.r.t. the d 'th coordinate, we continue with the first coordinate again. In the two-dimensional case we get, for instance,





One can build optimal k-d trees (i.e. k-d trees of depth ${}^2\log n + 1$) in $O(n \log n)$ by taking for the splitting point the median with respect to the splitting coordinate (cf. Bentley [1]). We assume again that no more than a constant number of points have a same coordinate.

In this section we will argue that a similar theory as presented in section 2 and 3 for quad-trees, can be set up for k-d trees as well. We will show how to

3.2. Insertion and deletion in EPQ-trees.

Although its structure may seem to be rather complex, an EPQ-tree can be built efficiently and is recursive, i.e., every substructure is built in exactly the same way as the total structure. We can therefore make it dynamic again, with only a minor loss in optimality.

Theorem 3.2.1. For any fixed δ with $0 < \delta < 1$ there is a way to perform N insertions and deletions in an initially empty EPQ-tree such that its depth is at most ${}^{2-\delta}\log n + d$ (where n is the current number of points in the EPQ-tree) and the average transaction time is bounded by $\frac{d}{\delta} \log^2 N$.

Proof.

The proof is similar to the proof of theorem 2.3.1. When we want to insert or delete a point p we first determine, by a search on the EPQ-tree, where (in the main tree or an associated structure) p needs to be inserted or deleted, and we perform the action. It is possible that somewhere at a node on the path from p to the root (of the main tree) the balance is disturbed. Determine the highest such node h on the search-path and rebuild the complete sub EPQ-tree of which h is the root. This takes $O(d.k \log k)$ where k is the number of point below h. In the same way as in the proof of theorem 2.3.1. we can show there must have been $O(\delta\ k)$ transactions in the subtree at h since the last rebalancing at h. When we charge the cost of rebuilding the subtree to the transactions after the latest rebuilding, it makes for a cost of $O(\frac{d}{\delta} \log n) \leq O(\frac{d}{\delta} \log N)$ per transaction. By the same arguments as in the proof of theorem 2.3.1. one can show that each transaction is charged only once from each level of the EPQ-tree. It follows that the average transaction time is bounded by $O(\frac{d}{\delta} \log^2 N)$.

□

The extensions made in an EPQ-tree do not increase the storage required by more than a constant fraction. Hence an EPQ-tree is a same powerful structure as a pseudo quad-tree, but with the property that it can handle all sorts of configurations, even the most degenerated ones.

4. k-d trees.

Another well-known data-structure for multi-dimensional queries is the k-d tree, presented by Bentley [1,2]. To build a k-d tree for a set of points

Because the splitting points only split w.r.t. one coordinate one has no need for splitting points, in practice, but only for the splitting coordinate of the points.

One can build an optimal pseudo k-d tree, i.e., a pseudo k-d tree with depth $\log n + 1$, in $O(n \log n)$ in exactly the same way as we did for ordinary k-d trees, except that we do not take the median itself as splitting point, but some point between the median and its nearest neighbour (w.r.t. the splitting coordinate). Hence, pseudo k-d trees have the same properties as ordinary k-d trees, but they have the advantage that the points of the set have no splitting function (as in the case of pseudo quad-trees). We will show that it is possible to delete points from pseudo k-d trees efficiently. Willard [10] already developed a dynamization of pseudo k-d trees (which he called k-d* trees), based on the decomposable nature of the problems they are used for, by building and maintaining a forest of pseudo k-d trees of different sizes (see also Overmars and van Leeuwen [8]). This way of dynamizing pseudo k-d trees has the disadvantage that the query time tends to increase by a multiplicative factor of $O(\log n)$. It is possible to maintain a pseudo k-d tree dynamically itself in the same way as we did for pseudo quad-trees.

Theorem 4.1.2. For any fixed δ with $0 < \delta < 1$ there is a way to perform N insertions and deletions in an initially empty pseudo k-d tree such that its depth is always at most ${}^{2-\delta}\log n$ (where n is the current number of points in the pseudo k-d tree) and the average transaction time is bounded by $\frac{1}{\delta} \log^2 N$.

Proof.

The proof is completely analogous to the proof of theorem 2.3.1. with $d = 1$.

$\square$

Thus a pseudo k-d tree is an efficient, fully dynamic datastructure for queries about multi-dimensional pointsets. But there is again one problem. To eliminate the restriction that no more than a constant number of points had a same coordinate, we have to extend the pseudo k-d tree in a way similar to the extension of the pseudo quad-tree to EPQ-tree.

4.2. Extended pseudo k-d trees.

In section 3 we saw how to extend pseudo quad-trees so they can be used for arbitrary configurations. In this subsection we will show that a similar technique also applies to pseudo k-d trees. Again we add associated structures to each node. For pseudo k-d trees this is much easier than for pseudo quad-trees. When we split perform insertions and deletions in so-called pseudo k-d trees, and extend the structure to be able to handle all kinds of configurations.

4.1. Pseudo k-d trees.

As for quad-trees, one can perform insertions in k-d trees efficiently, with only a little loss in optimality of the structure.

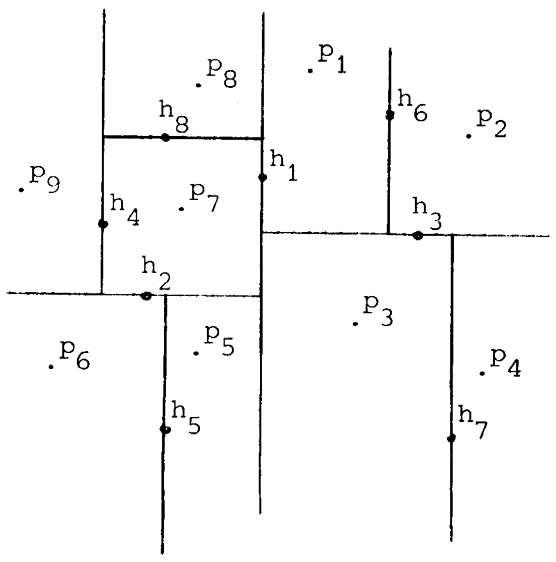
Theorem 4.1.1. For any fixed δ with $0 < \delta < 1$ there is a way to perform N insertions into an initially empty k-d tree such that its depth is always at most ${}^{2-\delta}\log n$ (where n is the current number of points in the structure) and the average transaction time is bounded by $\frac{1}{\delta} \log^2 N$.

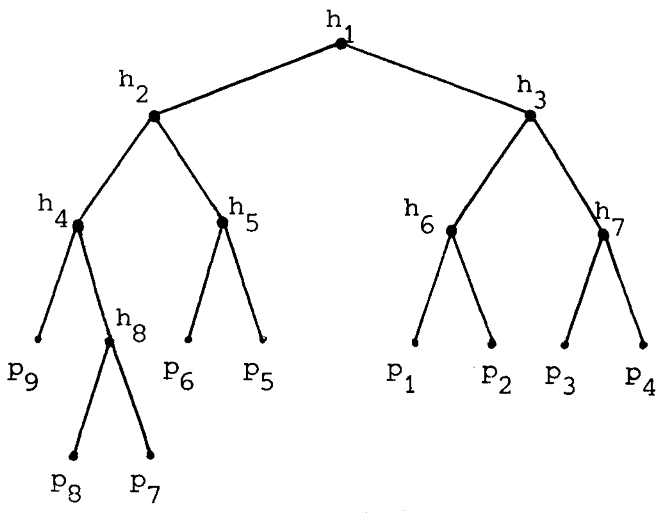
Proof.

The proof is similar to the proof of theorem 2.1.1.

□

Performing deletions in k-d trees is just as hard as it is in quad-trees. Therefore we again have to modify the structure. The so-called pseudo k-d tree is similar to a pseudo quad-tree, being a k-d tree with arbitrary points as internal nodes instead of points of the set. The points of the set occur only at the leafs of a pseudo k-d tree. So, for instance, a 2-dimensional pseudo k-d tree is shown in the following diagram. (p_1, ..., p_9 are the points of the set and h_1, ..., h_8 are the arbitrary points.)





Theorem 4.2.2. For any fixed δ with $0 < \delta < 1$ there is a way to perform N insertions and deletions in an initially empty extended pseudo k-d tree such that its depth is at most ${}^{2-\delta}\log n + d + 1$ (where n is the current number of points in the extended pseudo k-d tree) and the average transaction time is bounded by $\frac{1}{\delta}\log^2 N$.

Proof.

The proof is analogous to the proof of theorem 3.2.1. with d = 1.

$\square$

The extension does not increase the storage required for the structure (which is still O(n)). Hence, an extended pseudo k-d tree is a fully dynamic multi-dimensional data structure, that can handle all kinds of configurations of points, even the most degenerated ones.

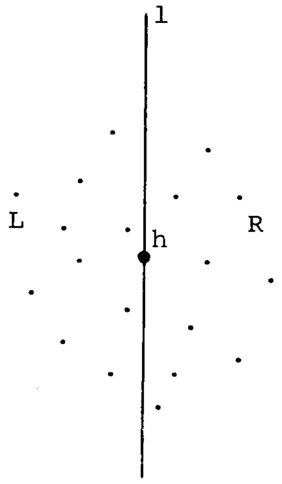
5. Concluding remarks.

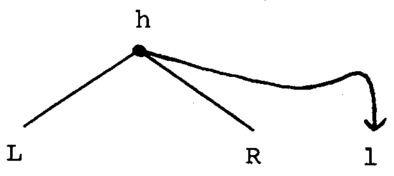
We have shown that quad-trees and k-d trees, both known efficient data structures for multi-dimensional queries on static pointsets, can be transformed into fully dynamic data structures with the same efficiency w.r.t. queries and with very reasonable average insertion and deletion times. The dynamization was obtained by a technique of "local rebuilding". This technique can also be used in various other data structures, e.g. in range trees (see Lueker [6] and Willard [11]) and in structures for convex hulls, intersections of half spaces, and maximal elements (Overmars and van Leeuwen [7]).

The pseudo quad-tree we described has a smaller depth than the ordinary quad-tree for a same set of points, which in general will lead to a decrease in the query time needed. Finkel and Bentley [4] and later also Kersten and van Emde Boas [5] have tried to achieve this for ordinary quad-trees by means of "local optimization techniques". Similar techniques seem to be applicable to pseudo quad-trees, but we have not investigated this further at the present time.

The average insertion and deletion times of $O(\log^2 N)$ for pseudo quad-trees, pseudo k-d trees and the extensions really are upper-bounds only because, when we insert or delete random points, we will not often have to rebuild parts of the tree. An expected time complexity of O(log N) per transaction seems reasonable, but remains to be proved.

the set at an internal node of a pseudo k-d tree we do this by choosing a hyperplane (with respect to one coordinate). So the only structure we have to add to an internal node is the splitting hyperplane. An extended pseudo k-d tree is a pseudo k-d tree in which every internal node h contains an extended pseudo (k-1)-d tree of the points that lie on the splitting hyperplane of h. (We define a 1-dimensional pseudo k-d tree as an ordinary balanced binary search tree.) In the 2-dimensional case we get, for instance,





Theorem 4.2.1. Given a set of n points in d-dimensional space, we can build an extended pseudo k-d tree of depth at most log n + d + 1 in O(n log n).

Proof.

Building an extended pseudo k-d tree works in exactly the same way as building an ordinary pseudo k-d tree, the only difference being that, when there are more points with the same median splitting coordinate, we build them in the associated structure. One can easily see that this does not increase the building time. Because we cannot go more than d times into an associated structure, the depth will never exceed the bound of log n + d + 1.

□

By the same arguments as for EPQ-trees, the extension will, in general, not increase the query time. Also the bounds on the insertion and deletion times are not affected by the extension.

6. References.

[1] Bentley, J.L., Multidimensional binary search trees used for associated searching, Comm. of the ACM 18 (1975), pp. 509-517.

[2] Bentley, J.L., Multidimensional binary search trees in database applications, IEEE Trans. on Software Eng. SE-5 (1979), pp. 333-340.

[3] Bentley, J.L., and D.F. Stanat, Analysis of range searches in quad-trees, Inf. Proc. Let. 3 (1975) pp. 170-173.

[4] Finkel, R.A., and J.L. Bentley, Quad-trees; a data structure for retrieval on composite keys, Acta Informatica 4 (1974) pp. 1-9.

[5] Kersten, M.L., and P. van Emde Boas, Local optimizations of quad-trees, Informatica Rapport IR-51, Vrije Universiteit Amsterdam, 1979.

[6] Lueker, G.S., A data structure for orthogonal range queries, Proc. 19th Symp. Foundations of Computer Science, IEEE, Oct. 1978, pp. 28-34.

[7] Overmars, M.H., and J. van Leeuwen, Maintenance of configurations in the plane, Techn. Rep. RUU-CS-79-9, Dept. of Computer Science, University of Utrecht, 1979.

[8] Overmars, M.H., and J. van Leeuwen, Two general methods for dynamizing decomposable searching problems, Techn. Rep. RUU-CS-79-10, Dept. of Computer Science, University of Utrecht, 1979.

[9] Samet, H., Deletion in two dimensional quad trees, unpublished manuscript, Maryland, 1979.

[10] Willard, D.E., Balanced forests of k-d* trees as a dynamic data structure, TR-23-78, Aiken Computation Lab., Harvard University, 1978.

[11] Willard, D.E., The super-B-tree algorithm, TR-03-79, Aiken Computation Lab., Harvard University, 1979.