Divided k-d trees

Marc J. van Kreveld Mark H. Overmars

RUU-CS-88-28
December 1989

University of Utrecht

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Divided k-d trees*

Marc J. van Kreveld Mark H. Overmars

December 5, 1989

Abstract

A variant of k-d trees, the *divided k-d tree,* is described that has some important advantages over ordinary k-d trees. The divided k-d tree is fully dynamic and allows for the insertion and deletion of points in $O(\log n)$ worst-case time. Moreover, divided k-d trees allow for split and concatenate operations. Different types of queries can be performed with equal or almost equal efficiency as on ordinary k-d trees. Both two- and multi-dimensional divided k-d trees are studied.

Key words. k-d tree, Divided k-d tree, Range searching, Dynamization, Splitting, Concatenating.

1 Introduction

k-d trees were defined by Bentley[2, 3] as an alternative for quad-trees ([5]) for storing points (and other objects) in the plane and higher dimensional space. The structure is based on a division of the space into rectangles of decreasing size. Both quad- and k-d trees are often used in Computer Graphics. See Samet[15] for a bibliography of over 250 papers written on these and related structures.

A k-d tree is a binary tree that is constructed as follows: Choose a value v that will be stored at the root of the tree. It splits the set of points to be represented in two subsets, based on the value of the first coordinate of the points. The two subsets will be stored in the two subtrees below the root. In both subsets a value is again taken with which the set is split, but this time with respect to the second coordinate. The four subsets that are obtained are split on the first coordinate again, and the alternation continues until there is only one point in the set. Then a leaf with this point is added. Thus the root node splits the point set on first coordinate, the two nodes on the second level on second coordinate, the four nodes on the third level on first coordinate, and so on. Figure 1 gives an example of a point set and how it can be split for a k-d tree.

*Authors address: Department of Computer Science, University of Utrecht, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

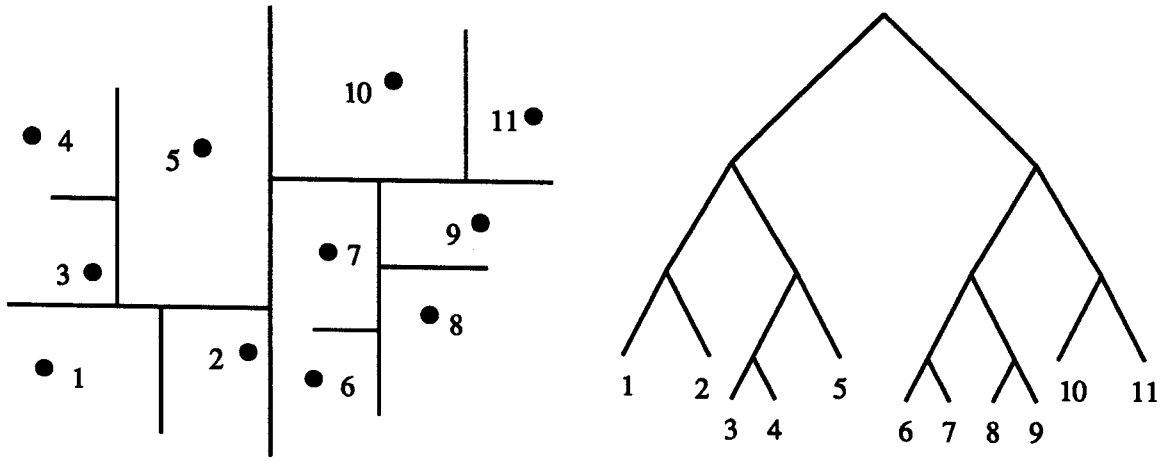

Figure 1: A k-d tree

k-d trees are in particular used for partial match and range searching problems. In this paper we concentrate on the more general range searching problem:

Definition 1 *Given a set S of points in the plane, and an axis-parallel rectangle R $[a_1 : b_1] \times [a_2 : b_2]$ (a range), an (orthogonal) range query with R asks for all points $p = (p_1, p_2)$ in S with $a_1 \leq p_1 \leq b_1$ and $a_2 \leq p_2 \leq b_2$.*

In graphics terms, a range query corresponds to a windowing problem in which we ask which part of a scene is visible inside a rectangle. In windowing problems the scene normally consists of more general objects than points. In this paper however we concentrate on sets of points (although k-d trees can also store other types of objects).

On k-d trees, range queries can be answered in $O(\sqrt{n} + k)$ time, where n is the number of points in the k-d tree and k is the number of answers to the query. The structure uses only linear storage. Other structures have been proposed for solving range queries (see e.g. [4, 9, 13, 16, 17]) but all these structures require more than linear storage. Moreover these structures and their algorithms are much more complicated.

A major problem of quad- and k-d trees is that it seems time consuming to keep them balanced when points are inserted or deleted. Unfortunately, the query time grows quickly when the structure gets out of balance. Some partial solutions to this problem are given in [6, 14]. In [12], Overmars and van Leeuwen introduced pseudo k-d trees, on which for any ε, $0 < \varepsilon < 1$, updates can be performed in $O(\frac{1}{\varepsilon} \log^2 n)$ amortized time, and range queries take $O(n^{1/(2\log(2-\varepsilon))} + k)$ worst-case time. (For decreasing ε, this approximates $O(\sqrt{n} + k)$.)

In this paper we introduce another variant of k-d trees, the *divided k-d tree*, that is fully dynamic and allows for insertions and deletions of points in $O(\log n)$ worst-case time. The time for range queries in the worst case becomes $O(\sqrt{n \log n} + k)$, which is better than the pseudo k-d tree.

The divided k-d tree as we describe it is based on a 2-3 tree rather than a binary tree (although e.g. an AVL-tree could be used as well). We assume the reader is familiar with 2-3 trees (see e.g. [1] or [7] for a description). It again divides the plane in rectangles but the splitting is performed in a different way. First only the second coordinate is split. Later only the first coordinate is split. This gives the structure flexibility that can be used when performing updates.

The paper is organized as follows.

In section 2 the two-dimensional divided k-d tree is introduced, and the algorithms for exact match queries, range queries, insertions and deletions are given. Also, time and storage bounds will be proved.

In section 3 we concentrate on split and concatenate operations. By introducing a concatenable environment we can perform such operations efficiently. Splits and concatenates can be performed on both coordinates. In this way one can, for example, obtain a k-d tree of the points inside a window, a useful operation if one wants to perform further queries on such a subset.

In section 4 we generalize the two-dimensional divided k-d tree to multi-dimensional space. It is shown that again updates can be performed in $O(\log n)$ time.

In section 5 a concatenable environment is described such that higher-dimensional divided k-d trees can be split and concatenated on all coordinates.

Finally, in section 6, we will give concluding remarks and open problems.

2 Two-dimensional divided k-d trees

In a normal k-d tree, levels that split on the first coordinate and levels that split on the second coordinate alternate. In a divided k-d tree, the upper levels split on the second coordinate only and the lower levels split on the first coordinate only. We will call these upper levels the upper tree, and a subtree in the lower levels is called a lower tree. In the plane, the upper tree divides the point set in a number of horizontal slabs, of which the points are stored in the lower trees, every lower tree being sorted on the first coordinate (see figure 2).

If there are many points with equal valued second coordinates, a balanced division in horizontal slabs may not always be possible. This problem can be solved by using the lexicographical ordering to split in the upper tree (with the second coordinate as the most significant coordinate). In this way the tree becomes balanced again and all operations described below can be performed in exactly the same way. Hence, in the sequel we can assume that no two points have equal valued first or second coordinates.

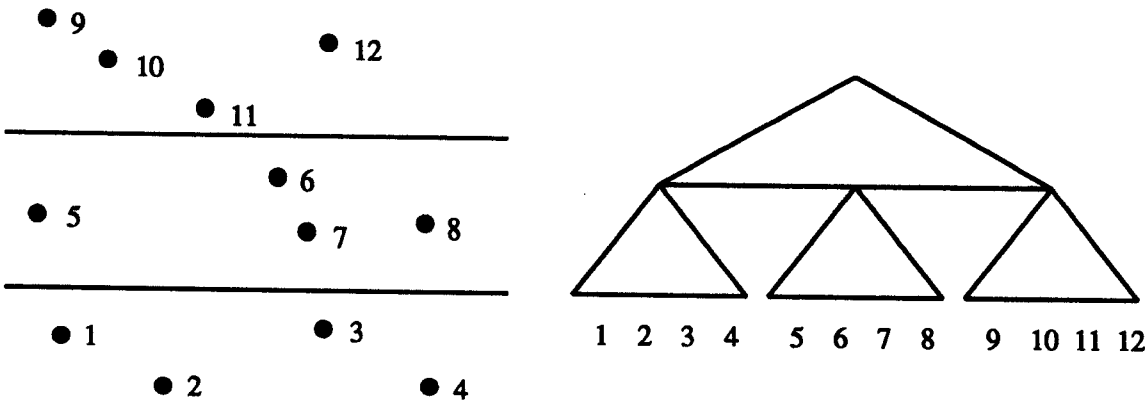

Figure 2: A divided k-d tree

Definition 2 *A two-dimensional tree, representing a set S of n points in the plane, is a divided k-d tree if and only if the tree consists of an upper tree, which divides the point set on the second coordinate, and with each leaf of the upper tree a lower tree, which divides the point set on the first coordinate, and which stores the points of the set S in the leaves. Furthermore, the following balancing conditions must hold:*

- *the upper tree is a balanced search tree and contains at most $2\sqrt{n/\log n}$ leaves;*
- *every lower tree is a balanced search tree and contains at most $2\sqrt{n \log n}$ leaves.*

From this definition it follows that the upper tree has size $\Theta(\sqrt{n/\log n})$, and lower trees have size $O(\sqrt{n \log n})$, but it is possible that a lower tree contains only one point. It may seem strange to use an unequal division in upper and lower trees, but as we will show, it leads to the best results. For the upper and lower trees we will use 2-3 trees.

2.1 Queries

We will first show how to perform exact match and range queries on a divided k-d tree.

Exact match query(T, p)

1. Search with p_2 in the upper tree of T, ending in a leaf δ of the upper tree.
2. Continue the search with p_1 at δ through the corresponding lower tree, ending in a leaf γ.
3. If γ contains p, then p is present, otherwise it is not.

Lemma 1 *On a divided k-d tree, storing n points in the plane, an exact match query can be performed in $O(\log n)$ time.*

Proof: Because both the upper tree and the lower tree we visit are balanced trees, the path we follow is of length $O(\log n)$, which proves the lemma. □

Next the algorithm for a range query with range $[a_1 : b_1] \times [a_2 : b_2]$ is given.

Range query$(T, [a_1 : b_1] \times [a_2 : b_2])$

1. Search with a_2 and b_2 in the upper tree of T, ending in two leaves δ_{a_2} and δ_{b_2} of the upper tree.

2. Traverse the lower trees corresponding to δ_{a_2} and δ_{b_2}, and report every point that lies in the range.

3. For all leaves δ of the upper tree that lie between δ_{a_2} and δ_{b_2} (possibly none), perform a one-dimensional range query with $[a_1 : b_1]$ on the corresponding lower tree.

Lemma 2 *On a divided k-d tree, storing n points in the plane, a range query can be performed in $O(\sqrt{n \log n} + k)$ time, where k is the number of points reported.*

Proof: We first prove the correctness of the algorithm. For all lower trees stored with leaves to the left of δ_{a_2} or to the right of δ_{b_2}, the points they store do not have their second coordinate in the range. For the lower trees stored with δ_{a_2} and δ_{b_2}, we do not know whether the second coordinate of the points they contain lies in the range, thus we perform a complete traversal and report the correct answers. For all lower trees stored with leaves between δ_{a_2} and δ_{b_2}, the second coordinate of the points they contain lie in the range, thus we can perform a one-dimensional range query on the first coordinate on them.

Following the paths through the upper tree takes $O(\log n)$ time, because the upper tree is balanced. The complete traversal of the lower tree(s) takes $O(\sqrt{n \log n})$ time, because this is a bound on the maximal size of a lower tree. Performing a one-dimensional range query on a lower tree takes $O(\log n + k')$ time, because we only follow one path and then start reporting answers. As there are at most $2\sqrt{n/\log n}$ leaves in the upper tree, we perform at most $2\sqrt{n/\log n} - 2$ one-dimensional range queries, giving a total time of $O(\sqrt{n \log n} + k)$. □

Remark: Notice that for any other division in upper trees and lower trees a range query time bound would be obtained that is worse than the one proved above.

Remark: We have seen that the range query time for well-balanced k-d trees is

slightly better than for divided k-d trees. However, for a dynamic k-d tree the range query time is $O(n^{1/(2\log(2-\epsilon))} + k) = O(\sqrt{n}^{1+\delta} + k)$ for a small positive δ, and this is asymptotically worse than $O(\sqrt{n \log n} + k)$ for any positive δ.

Remark: A disadvantage of the method as described above is that the query time will generally be $\Omega(\sqrt{n \log n})$ because we always traverse L_{a_2} and L_{b_2} completely. An easy improvement is to search in these lower tree with the range $[a_1 : b_1]$ and for each answer found check whether it lies in the whole range. Especially for small ranges this improves the query time considerably (although the worst case remains the same).

2.2 Insertions and deletions

Next we will show that divided k-d trees allow for insertions and deletions in $O(\log n)$ time amortized, which is superior to the update time bounds of range trees (see e.g. Willard and Lueker[17]), quad-trees and normal k-d trees. The amortized time bounds can be changed into worst-case time bounds, as we will show.

Below the insert and delete algorithms are given.

Insert(T, p)

1. Search with p_2 in the upper tree to determine the lower tree in which p must lie.
2. Insert p in this lower tree using the standard algorithm for inserting in 2-3 trees.
3. If necessary, rebalance.

Delete(T, p)

1. Search with p_2 for the lower tree of T in which p must lie.
2. Delete p from this lower tree using the standard deletion algorithm for 2-3 trees.
3. If the lower tree has become empty, then delete the corresponding leaf from the upper tree using the standard deletion algorithm.
4. If necessary, rebalance.

Inserting a point and deleting a point without rebalancing can be done in $O(\log n)$ time, because the standard insertion and deletion algorithms on 2-3 trees take $O(\log n)$ time.

To keep the tree balanced two different methods of rebalancing are used simultaneously. When a lower tree becomes large, the point set it represents is split in two subsets and two new lower trees are made.

Divide lower tree(L)

1. Read all points from L maintaining the order on the first coordinate, and dispose of L.

2. Find the median of the second coordinates of the points, and divide the set accordingly in two ordered sets of equal size.

3. For each ordered set, build a lower tree and add it to the upper tree.

Clearly the operation will take time $O(\sqrt{n \log n})$ (because the number of points involved is $O(\sqrt{n \log n})$ and the algorithm takes linear time in the number of points).

The second way in which we rebalance is the rebuilding of the whole tree. In this case we reconstruct the tree such that the upper tree has $\sqrt{n/\log n}$ leaves and each lower tree contains $\sqrt{n \log n}$ points.

Rebuild(T)

1. Read all points from T and dispose of T.

2. Sort all points on the second coordinate.

3. Let n be the total number of points. Repeatedly take the first $\lceil \sqrt{n \log n} \rceil$ points of the sorted set, and build a lower tree of the subset (ordered on first coordinate). Do the same with the remaining points.

4. Build an upper tree on the lower trees, choosing the correct splitting values for the internal nodes.

The rebuilding takes $O(n \log n)$ time.

Next we show when rebalancing is performed, and that this choice leads to divided k-d trees which are always balanced. Assume at some moment in time we reconstruct the whole tree and let n_0 be the number of points in the set at that moment. We rebuild the whole tree after $\frac{1}{3}n_0$ insertions or $\frac{1}{3}n_0$ deletions have taken place. Moreover, we split a lower tree when, as a result of an insertion in it, its size becomes larger than $\frac{8}{5}\sqrt{n_0 \log n_0}$.

Lemma 3 *Each lower tree will have size at most* $2\sqrt{n \log n}$.

Proof: Each lower tree will have size $\leq \frac{8}{5}\sqrt{n_0 \log n_0}$. Moreover $n \geq \frac{2}{3}n_0$, i.e., $n_0 \leq \frac{3}{2}n$. Hence, the size will be $\leq \frac{8}{5}\sqrt{\frac{3}{2}n \log(\frac{3}{2}n)} < 2\sqrt{n \log n}$. $\square$

Lemma 4 *The upper tree will have at most $2\sqrt{n/\log n}$ leaves.*

Proof: The upper tree had $\sqrt{n_0/\log n_0}$ leaves after the rebuilding. There have been at most $\frac{1}{3}n_0$ insertions since. Each lower tree had size $\sqrt{n_0 \log n_0}$. To add a leaf to the upper tree there must have been $\frac{8}{5}\sqrt{n_0 \log n_0} - \sqrt{n_0 \log n_0} = \frac{3}{5}\sqrt{n_0 \log n_0}$ insertions in some lower tree. As a result there can be at most $(\frac{1}{3}n_0)/(\frac{3}{5}\sqrt{n_0 \log n_0}) = \frac{5}{9}\sqrt{n_0/\log n_0}$ new leaves in the upper tree. Hence, the upper tree will have less than $(1+\frac{5}{9})\sqrt{n_0/\log n_0} \leq (1+\frac{5}{9})\sqrt{\frac{3}{2}n/\log \frac{3}{2}n} < (1+\frac{5}{9})\sqrt{\frac{3}{2}}\sqrt{n/\log n} < 2\sqrt{n/\log n}$ leaves. □

These two lemmas show that the tree indeed remains balanced. It remains to show that the amortized rebuilding time is small.

Lemma 5 *The amortized rebuilding time is $O(\log n)$ per insertion and deletion.*

Proof: There are two types of rebuilding, rebuilding of the whole tree and rebuilding of a lower tree. When we rebuild the whole tree $\Omega(n_0) = \Omega(n)$ updates have taken place since the last rebuilding. We charge the rebuilding cost to these updates. As the rebuilding takes time $O(n \log n)$ this is $O(\log n)$ per update. When we rebuild a lower tree, at least $\frac{3}{5}\sqrt{n_0 \log n_0}$ insertions have taken place on it. As the rebuilding takes time $O(\sqrt{n_0 \log n_0})$ this is $O(1)$ per insertion. □

This leads to the main result of this section:

Theorem 1 *There exists a two-dimensional tree, the divided k-d tree, for storing a set of n points in the plane, such that exact match queries take $O(\log n)$ time, range queries take $O(\sqrt{n \log n} + k)$ time (where k is the number of points reported) and insertions and deletions can be performed in $O(\log n)$ amortized time. The tree uses $O(n)$ space to store and it takes $O(n \log n)$ time to build.*

Proof: Follows from lemmas 1, 2 and 5. □

2.3 Worst case bounds

The amortized insertion and deletion time bounds of theorem 1 can be changed into worst-case time bounds with the general technique of global rebuilding, introduced by Overmars and van Leeuwen in [11], also described in [10]. We will not describe the technique here, but we will give the theorem that we use to obtain worst-case time bounds.

Theorem 2 *Given a tree T representing n points with rebuilding time bounded by $O(n \log n)$ that allows for weak insertions and for weak deletions, we can dynamize it into a structure T' such that*

- *the insertion time on T' is $O(WI_T(n) + \log n)$;*
- *the deletion time on T' is $O(WD_T(n) + \log n)$;*
- *the query time and amount of storage space used do not change asymptotically.*

($WI_T(n)$ and $WD_T(n)$ are the weak insertion time and weak deletion time for a tree T representing n points. An update is called weak if there exists an α, $0 < \alpha < 1$, such that after $\alpha \cdot n$ updates, all time and storage bounds of the structure haven't increased asymptotically.)

We cannot apply this theorem directly, because lower trees can become large within $o(n)$ updates. Remember that it took $\Omega(\sqrt{n \log n})$ updates in a lower tree before it could become large. We first apply theorem 2 to each lower tree, so that an update on a lower tree will be a weak update on the whole tree. Now we can apply theorem 2 to the divided k-d tree, and obtain the following result.

Theorem 3 *There exists a two-dimensional tree, the divided k-d tree, for storing a set of n points in the plane, such that exact match queries take $O(\log n)$ time, range queries take $O(\sqrt{n \log n} + k)$ time (where k is the number of points reported) and insertions and deletions can be performed in $O(\log n)$ time worst-case. The tree uses $O(n)$ space to store and it takes $O(n \log n)$ time to build.*

3 A concatenable environment for divided k-d trees

We will now concentrate on splitting and concatenating divided k-d trees on second and first coordinate (or equally, with horizontal and vertical lines, respectively). It is clear from the structure that splitting and concatenating on second coordinate will mainly involve the upper trees, because the nodes in the upper tree divide the point set in the plane on second coordinate. Likewise, splitting and concatenating on first coordinate mainly involves the lower trees.

As the upper and lower trees of divided k-d trees are 2-3 trees we can use the standard algorithms for splitting lower and upper trees in $O(\log n)$ time. Splitting on the second coordinate would require splitting the upper tree and rebuilding one lower tree completely, namely the lower tree that corresponds to the leaf of the upper tree in which the search path to the splitting value ended. Splitting on the first coordinate requires splitting all lower trees. These can be added to two copies of the upper tree. Concatenating on the second coordinate only requires concatenating the top trees. But concatenating on the first coordinate is not that simple. The problem is that the upper trees of the two divided k-d trees to be concatenated are not necessarily equal, and consequently the lower trees of the two divided k-d trees do not correspond one to one such that they can be concatenated. Figure 3

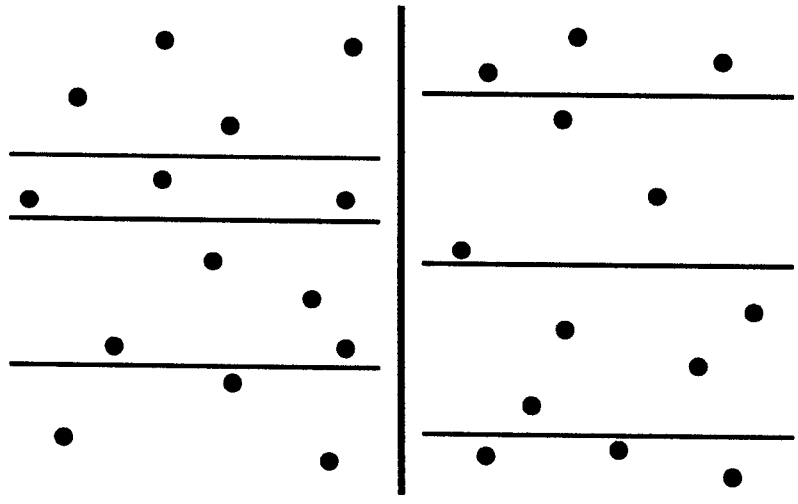

Figure 3: Problem with concatenating on first coordinate

illustrates the problem that occurs when the horizontal slabs that arise from the upper trees, do not coincide.

Besides this problem, another difficulty arises in keeping the upper and lower trees balanced. Splitting on the second coordinate, for example, can divide the upper tree in two upper trees of size $\frac{1}{k}$ and $\frac{k-1}{k}$ times the number of nodes in the original upper tree, for any k. This implies that for every divided k-d tree with n points, there is a way to split on second coordinate such that the resulting tree has $\Omega(n)$ points and is out of balance. Restoring the balance will take too much time to make splitting efficient.

To overcome these difficulties, we require that all upper trees of divided k-d trees that can be involved in splitting and concatenating, will divide the plane in the same horizontal slabs. To impose balance, we relax the balance conditions and let n be the total number of points in all divided k-d trees rather than in an individual k-d tree. We construct one balanced divided k-d tree (called MAIN) that contains all the points. This tree defines the way in which the upper trees of all the other k-d trees split the plane. (The points stored in the lower trees of course differ from tree to tree). As a consequence, time bounds will not be expressed in the number of points in the tree itself, but in n, the number of points in all divided k-d trees. If a tree to be split has $\Theta(n)$ points, it does not matter that the time bounds are expressed in n, because this is also a tight bound on the size of the tree itself. But for small trees time bounds can be rather bad. We will call the collection of divided k-d trees that can be split and concatenated with each other a concatenable environment for divided k-d trees.

Definition 3 *A concatenable environment for two-dimensional divided k-d trees, representing a set S of n points (in the plane) in k trees, consists of the following:*

- *A balanced divided k-d tree MAIN, representing all n points. Let the lower trees of MAIN be $B_1, \ldots, B_m$.*
- *Divided k-d trees $T_1, \ldots, T_k$, for which the lower trees are 2-3 trees, the upper trees are 2-3 trees, but the divided k-d trees themselves are unbalanced. The trees form a partition of the n points. Every tree T_j, $1 \leq j \leq k$, has lower trees $B_{1j}, \ldots, B_{mj}$, such that a point p is in a lower tree B_{ij} if and only if p is in both B_i and T_j $(1 \leq i \leq m)$. Only non-empty lower trees B_{ij} are stored.*
- *With every lower tree B_i of MAIN $(1 \leq i \leq m)$, an extra tree C_i is stored with a reference to every lower tree B_{ij} that is not empty.*

Thus every divided k-d tree is partitioned in the same lower trees as MAIN is. We could also say that every lower tree B_i is partitioned in the lower trees $B_{i1}, \ldots, B_{ik}$, situated in the trees $T_1, \ldots, T_k$, respectively. Because we keep MAIN balanced with the balancing conditions of divided k-d trees, the unbalanced divided k-d trees T_j are also in a way balanced (they can never be worse than MAIN).

The extra trees C will not influence the bound on the amount of memory used by the environment, because the extra tree cannot be larger than the lower tree of MAIN it is stored with.

See figure 4 for an example of a concatenable environment.

Lemma 6 *An exact match query on a divided k-d tree in a concatenable environment with n points takes $O(\log n)$ time, and a range query takes $O(\sqrt{n \log n} + k)$ time, where k is the number of points reported.*

Proof: This follows easily because no divided k-d tree in the concatenable environment can have a lower tree or upper tree bigger than the corresponding lower tree or upper tree of MAIN, and MAIN is balanced. □

When we update a tree T_j, we also perform this update on MAIN. Only the updating of MAIN may cause rebuilding. The algorithms for inserting, deleting and rebalancing are given below.

Insert(T_j, p)

1. Insert p in MAIN as described in the previous section, but without rebalancing. Let the updated lower tree of MAIN be B_i.
2. If there is no lower tree B_{ij} yet in T_j, then add a leaf for this lower tree to the upper tree and add a reference to B_{ij} in C_i.

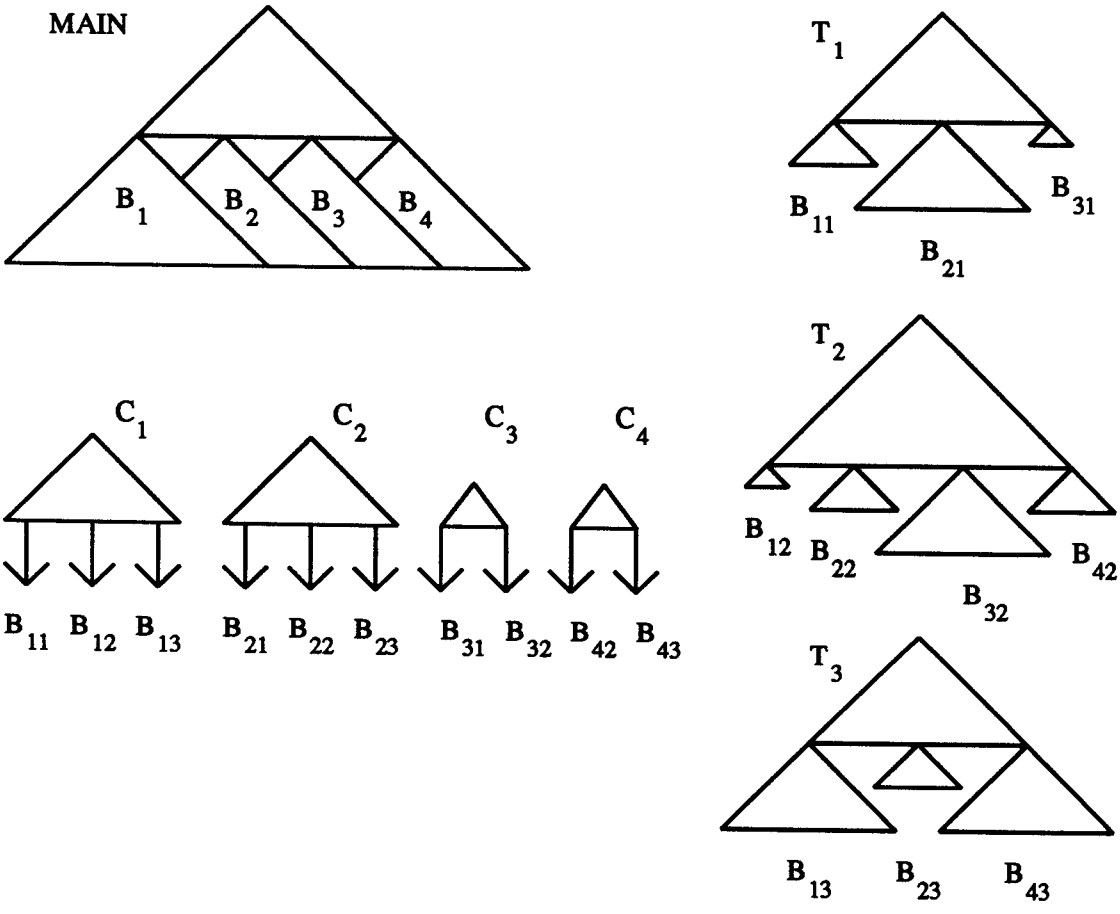

Figure 4: Example of a concatenable environment

3. Insert p in the lower tree B_{ij}.

4. If necessary, rebalance.

Delete(T_j, p)

1. Delete p from MAIN as described in the previous section, but without rebalancing.

2. Delete p from T_j as described in the previous section, but without rebalancing. If the lower tree B_{ij} has become empty, then delete the corresponding leaf from the upper tree and delete the reference to this lower tree from C_i.

3. If necessary, rebalance.

Divide lower tree(B_i)

1. Divide B_i as described in the previous section, obtaining lower trees B_i' and B_i''. Let z be the median found.

2. For every lower tree B_{ij} to which there is a reference in C_i, divide B_{ij} using z as splitting value into B_{ij}' and B_{ij}''.

3. Build new trees C_i' and C_i'' with references to the lower trees B_{ij}' and B_{ij}'' (if they are not empty), respectively.

Rebuild(**MAIN**)

1. Rebuild MAIN as described in the previous section.

2. Make $C_1, \ldots, C_m$ empty.

3. For every tree T_j, $1 \leq j \leq k$, do the following.

   - Let T_j' be an initially empty tree (for the points in T_j).
   - For every point p in T_j, search in the upper tree of MAIN to decide in which lower tree B_{ij}' of T_j' it must be stored. If this lower tree is not present yet, then add a leaf to the upper tree of T_j' for this lower tree, and add to C_i a reference to this lower tree. Insert p in the lower tree B_{ij}'.
   - Dispose of T_j.

Lemma 7 *Insertions and deletions on a divided k-d tree in a concatenable environment with n points take* $O(\log n)$ *time.*

Proof: As in the previous section, a lower tree is divided when the corresponding lower tree of MAIN, as result of an insertion, contains more than $\frac{8}{5}\sqrt{n_0 \log n_0}$ points (again, n_0 is the number of points with which the concatenable environment was rebuilt the last time). Also, rebuilding takes place after $\frac{1}{3}n_0$ insertions or $\frac{1}{3}n_0$ deletions. The amortized rebalancing time is again constant. In the same way as in the previous section, the amortized bounds can be changed into worst-case bounds. □

Next we will show how divided k-d trees in a concatenable environment can be split and concatenated both on first and second coordinate.

Split2(T_j, T_j', T_j'', s) (split T_j on second coordinate with splitting value s)

1. Split the upper tree of T_j with s using the standard splitting algorithm for 2-3 trees, resulting in the upper trees for T_j' and T_j''.
2. Let B_{ij} be the lower tree corresponding to the leaf in which the splitting path ended. Divide B_{ij} with splitting value s by traversing it completely, and build two new lower trees B_{ij}' and B_{ij}''.
3. Add B_{ij}' as rightmost lower tree of T_j' and add B_{ij}'' as leftmost lower tree of T_{ij}''.
4. Delete the reference to B_{ij} from C_i, and add references to B_{ij}' and B_{ij}'' (if they are not empty).

Concatenate2(T_j, T_l, T_q) (concatenate T_j and T_l on second coordinate, where T_j contains the points with the smaller values for the second coordinate)

1. Concatenate the upper trees of T_j and T_l using the standard concatenating algorithm for 2-3 trees, resulting in the upper tree for T_q.
2. Let B_{ij} be the rightmost lower tree of T_j, and let $B_{i'l}$ be the leftmost lower tree of T_l. If $i = i'$, then make one new lower tree by merging the points in B_{ij} and $B_{i'l}$, delete from C_i the references to B_{ij} and $B_{i'l}$, and add to C_i a reference to the new lower tree.

Lemma 8 *A divided k-d tree in a concatenable environment, containing n points, can be split on second coordinate in time $O(\sqrt{n \log n})$, and two divided k-d trees can be concatenated on second coordinate in time $O(\sqrt{n \log n})$.*

Proof: Splitting the upper tree with the standard algorithm takes $O(\log n)$ time, the deletion and insertions on C_i take $O(\log n)$ time, and dividing the lower tree can be done in $O(\sqrt{n \log n})$ time. The time bound for concatenating can be proved in a similar way. □

Split1(T_j, T_j', T_j'', s) (split T_j on first coordinate with splitting value s)

1. Split all lower trees of T_j with s using the standard splitting algorithm, resulting in a number of lower trees for T_j' and a number of lower trees for T_j''.
2. Build new upper trees for T_j' and T_j''.
3. For every tree C_i, $1 \leq i \leq m$, delete the reference to the lower tree of T_j (if it was not empty), and add references to the resulting lower trees of T_j' and T_j'' (if they are not empty).

Concatenate1(T_j, T_l, T_q) (concatenate T_j and T_l on first coordinate)

1. For every two corresponding lower trees B_{ij} and B_{il} of T_j and T_l, respectively, concatenate them using the standard concatenating algorithm.
2. Build a new upper tree for all lower trees (either resulting from concatenation or not).
3. For every tree C_i, $1 \leq i \leq m$, delete the references to the lower trees of T_j and T_l, and add a reference to the lower tree of T_q (if it is not empty).

Lemma 9 *A divided k-d tree in a concatenable environment, containing n points, can be split on first coordinate in time $O(\sqrt{n \log n})$, and two divided k-d trees can be concatenated on first coordinate in time $O(\sqrt{n \log n})$.*

Proof: There are $O(\sqrt{n/\log n})$ lower trees to be split, which takes in total $O(\sqrt{n/\log n}) \times O(\log n) = O(\sqrt{n \log n})$ time. Building the upper tree can be done in $O(\sqrt{n/\log n})$ time, and the updates on the extra trees take $O(\sqrt{n \log n})$ time. The bound for concatenating can be proved in a similar way. □

We summarize the main results of this section in the following theorem.

Theorem 4 *In a concatenable environment for divided k-d trees, containing in total n points, insertions and deletions on a divided k-d tree can be performed in $O(\log n)$ time, exact match queries take $O(\log n)$ time, range queries take $O(\sqrt{n \log n} + k)$ time (where k in the number of points reported), and divided k-d trees can be split and concatenated on first and second coordinate in $O(\sqrt{n \log n})$ time. The environment uses $O(n)$ space to store.*

Proof: Follows from lemmas 6, 7, 8 and 9. □

4 Higher-dimensional divided k-d trees

The d-dimensional divided k-d tree is a generalization of the two-dimensional divided k-d tree. The first levels split the point set to be represented on last coordinate, the next levels on one but last coordinate, and so on to the last levels, which split the point set on first coordinate.

Definition 4 *A d-dimensional tree ($d > 1$), representing a set S of n points in d-dimensional space, is a (d-dimensional) divided k-d tree if and only if the tree consists of an upper tree, which is a balanced search tree with at most $2n^{1/d}\log^{-1/d} n$ leaves, and in which the internal nodes split the point set on last (d^{th}) coordinate. Each leaf corresponds to the root of a ($d-1$)-dimensional divided k-d tree with at most $2n^{1-1/d}\log^{1/d} n$ points, restricted to their first $d-1$ coordinates.*

A 1-dimensional divided k-d tree is an ordinary balanced search tree.

As in the two-dimensional case, we use an unequal division in layers. The 1-dimensional divided k-d trees of a d-dimensional divided k-d tree with n points have size $O(n^{1/d}\log^{1-1/d} n)$, and the upper trees of all c-dimensional divided k-d trees ($1 < c \leq d$) are of size $O(n^{1/d}\log^{-1/d} n)$. A subtree rooted at the root of a c-dimensional divided k-d tree contains $O(n^{c/d}\log^{1-c/d} n)$ points. For all balanced search trees we will use 2-3 trees.

An exact match query on a d-dimensional divided k-d tree ($d \geq 2$) can easily be performed by searching in the upper tree to find the lower tree that may contain the query point, and continuing the search in that lower tree.

To perform a d-dimensional range query with $[a_1 : b_1] \times \cdots \times [a_d : b_d]$, we follow the paths through the upper tree to a_d and b_d, ending in the leaves L_{a_d} and L_{b_d}. If $L_{a_d} = L_{b_d}$, then we traverse the whole lower tree rooted at this leaf and report the points that lie in the range. Otherwise, we traverse the whole lower trees rooted at the leaves L_{a_d} and L_{b_d}, and report all points that lie in the range. Also, for all leaves L that lie between L_{a_d} and L_{b_d}, we perform a $(d-1)$-dimensional range query with $[a_1 : b_1] \times \cdots \times [a_{d-1} : b_{d-1}]$ on the lower tree corresponding to L.

Lemma 10 *On a d-dimensional divided k-d tree, storing n points in d-dimensional space, an exact match query can be performed in $O(\log n)$ time, and a range query can be performed in $O(n^{1-1/d}\log^{1/d} n + k)$ time, where k is the number of points reported.*

Proof: An exact match query can be done by following only one path through the tree, which obviously has length $O(\log n)$. Here and in the sequel we consider d to be a constant. (If not, the query time would be $O(d \cdot \log n)$.)

Let $RQ(d, n)$ denote the range query time on a d-dimensional divided k-d tree with n points. Because a lower tree of a d-dimensional divided k-d tree with n points has size $O(n^{1-1/d}\log^{1/d} n)$, we obtain the following reccurence: $RQ(d, n) = O(n^{1-1/d}\log^{1/d} n) + O(n^{1/d}\log^{-1/d} n) \cdot RQ(d-1, c \cdot n^{1-1/d}\log^{1/d} n)$ for $d > 1$ and

some constant c, and $RQ(1,n) = O(\log n + k)$. The first term is the time needed for complete traversal of the lower trees corresponding to the leaves in which we end, and the second term is the time needed for all $(d-1)$-dimensional range queries. Solving the recurrence proves the time bound. □

4.1 Insertions and deletions

The insert and delete algorithms on two-dimensional divided k-d trees generalize immediately to higher dimensions. We first give the insert and delete algorithms, then we describe how rebalancing is done.

To insert a point $p = (p_1, \ldots, p_d)$ in a d-dimensional divided k-d tree ($d > 1$), we follow the path through the upper tree using the last coordinate of p to a leaf of the upper tree. Then we insert a point in the lower tree corresponding to this leaf (an insertion in a $(d-1)$-dimensional divided k-d tree). When the balancing conditions are violated, we rebalance.

To insert a point p in a 1-dimensional divided k-d tree we use the standard insertion algorithm.

To delete a point from a d-dimensional divided k-d tree we use the same approach.

There are two ways in which the balance in a d-dimensional divided k-d tree will be restored. When a lower tree becomes large, we divide the point set stored in that lower tree in two halves using the median of the last coordinate of the point set, and two new lower trees are built in $O(n^{1-1/d}\log^{1+1/d} n)$ time. These two lower trees are put in the upper tree, involving an insertion of a leaf in the upper tree. Next, suppose that after an update the upper tree has too many leaves. Then we rebuild the whole (d-dimensional) tree in $O(n \log n)$ time.

Lemma 11 *On a d-dimensional divided k-d tree, storing n points in d-dimensional space, insertions and deletions can be performed in $O(\log n)$ amortized time.*

Proof: Suppose a d-dimensional divided k-d tree has just been rebuilt. As in the two-dimensional case, it is easy to show that it takes $\Omega(n^{1-1/d}\log^{1/d} n)$ insertions in a lower tree before it must be split, provided that there have been no more than $c \cdot n$ deletions in the whole tree, for a suitably chosen constant c. Furthermore, it takes $\Omega(n)$ insertions or $\Omega(n)$ deletions in the whole tree before the number of leaves in the upper tree becomes too large. Finally, $O(\log n)$ time is spent for each update to follow the path through the upper tree. Let the update time of a d-dimensional divided k-d tree with n points be $U(d,n)$. Then $U(d,n) = O(\log n) + O(n \log n)/\Omega(n) + O(n^{1-1/d}\log^{1+1/d} n)/\Omega(n^{1-1/d}\log^{1/d} n) + U(d-1, c \cdot n^{1-1/d}\log^{1/d} n) \leq O(\log n) + U(d-1, c \cdot n)$, and because $U(1,n)$ is $O(\log n)$, $U(d,n)$ is $O(\log n)$ amortized. □

The performance of the d-dimensional divided k-d tree is summarized in the following theorem.

Theorem 5 *There exists a d-dimensional tree, the d-dimensional divided k-d tree, for storing a set of n points in d-dimensional space, such that exact match queries take $O(\log n)$ time, range queries take $O(n^{1-1/d}\log^{1/d} n + k)$ time (where k is the number of points reported) and insertions and deletions can be performed in $O(\log n)$ time amortized. The tree uses $O(n)$ space and it takes $O(n \log n)$ time to build it.*

In this theorem the amortized update time bounds can be changed into worst-case bounds by applying theorem 2 on every lower tree. We obtain the same results as in theorem 5, but with worst-case insertion and deletion time bounds instead of amortized time bounds.

5 A concatenable environment for d-dimensional divided k-d trees

In this section we will make a concatenable environment for d-dimensional divided k-d trees, such that they can be split and concatenated on every coordinate. To be able to do this, the upper trees of all divided k-d trees in the concatenable environment must divide d-dimensional space in the same way.

Definition 5 *A concatenable environment for d-dimensional divided k-d trees ($d \geq 3$), representing a set S of n points (in d-dimensional space) in k non-empty sets $T_1, \ldots, T_k$, consists of the following:*
(Let MAIN be a d-dimensional divided k-d tree representing the set S of all points.)

- *The upper tree of MAIN, partitioning the point set S in subsets $S_1, \ldots, S_m$. The leaves in the upper tree correspond to the subsets $S_1, \ldots, S_m$.*
- *The upper trees for the sets $T_1, \ldots, T_k$. The upper tree for T_j, $1 \leq j \leq k$, is defined as follows. Let T_j be partitioned in subsets $T_{1j}, \ldots, T_{mj}$, such that $T_{ij} = T_j \cap S_i$ for $1 \leq i \leq m$. Every non-empty subset T_{ij} corresponds to a leaf in the upper tree for the set T_j.*
- *A concatenable environment for $(d-1)$-dimensional divided k-d trees, representing a set S_i of points in the non-empty sets of $T_{i1}, \ldots, T_{ik}$, for all i, $1 \leq i \leq m$.*
- *A balanced tree C_i for every leaf in the upper tree of MAIN ($1 \leq i \leq m$), of which the leaves correspond to the leaves of the upper trees for the sets $T_1, \ldots, T_k$ that correspond to the subsets $T_{i1}, \ldots, T_{ik}$, if such a leaf exists (or equally, if T_{ij} is non-empty).*

This is a generalization of the two-dimensional concatenable environment defined in definition 3.

Exact match queries, range queries, insertions and deletions are performed in a way similar to the operations on a tree in a concatenable environment for two-dimensional divided k-d trees. Details are left to the reader.

To split a divided k-d tree T_j (in a concatenable environment for d-dimensional divided k-d trees) on the zth coordinate, we do the following.

If $z = d$, we split the upper tree with the standard algorithm for splitting 2-3 trees. For the leaf in which we end, we divide and reconstruct the corresponding lower tree. Also, we adjust the trees C. Finally, we pass back the roots of the two resulting upper trees.

If $z < d$, we split every lower tree of T_j recursively on the zth coordinate, obtaining two groups of lower trees. For both of these two groups we build a new upper tree. Furthermore, we adjust the trees C, and pass back the roots of the two upper trees obtained.

Lemma 12 *In a concatenable environment with n points, a d-dimensional divided k-d tree can be split on first and second coordinate in $O(n^{1-1/d}\log^{1/d} n)$ time, and on the other coordinates in $O(n^{1-1/d}\log^{1+1/d} n)$ time.*

Proof: Let the split time of a divided k-d tree in a concatenable environment for d-dimensional divided k-d trees with n points on the zth coordinate be $S(z,d,n)$, and its rebuilding time $R(d,n)$. Then we have the following recurrence. $S(d,d,n) \leq O(\log n) + R(d-1, c \cdot n^{1-1/d}\log^{1/d} n)$ for some constant c. (The first term is the time taken to split with the standard split algorithm and to adjust the tree C, and the second term to rebuild a lower tree.) For $z < d$, $S(z,d,n) \leq O(n^{1/d}\log^{-1/d} n) \cdot S(z, d-1, c \cdot n^{1-1/d}\log^{1/d} n) + O(n^{1/d}\log^{-1/d} n)$ for some constant c. (The first factor is a bound on the number of lower trees, the second factor is the split time of the lower trees, and the second term is the time needed to build new upper trees and to adjust the tree C.) For the rebuilding time we have $R(0,n) = 0$ because in this case there is nothing to rebuild, $R(1,n) = O(n)$ because we have the points in sorted order, and $R(d,n) = O(n \log n)$ for $d \geq 2$. Solving the recurrence leads to the time bounds stated. □

To concatenate two divided k-d trees T_j and T_l on the zth coordinate, we do the following.

If $z = d$, we concatenate the upper trees of T_j and T_l with the standard algorithm for concatenating 2-3 trees. If the rightmost lower tree of T_j must be taken together with the leftmost lower tree of T_l (because the points they contain are in the same lower tree of MAIN), then we rebuild these two lower trees as one. Also, we adjust the trees C. Finally, we pass back the root of the resulting upper tree.

If $z < d$, we concatenate every two corresponding lower trees of T_j and T_l recursively on the zth coordinate, obtaining one group of lower trees. For this group we build a new upper tree. Furthermore, we adjust the trees C, and pass back the root of the upper tree obtained.

Lemma 13 *In a concatenable environment with n points, two d-dimensional divided k-d trees can be concatenated on the first and second coordinate in* $O(n^{1-1/d}\log^{1/d} n)$ *time, and on the other coordinates in* $O(n^{1-1/d}\log^{1+1/d} n)$ *time.*

Proof: This can be proved in a way similar to lemma 12. □

We summarize the main results of this section in the following theorem.

Theorem 6 *In a concatenable environment for d-dimensional divided k-d trees, containing n points, insertions and deletions on a divided k-d tree can be performed in* $O(\log n)$ *time, exact match queries take* $O(\log n)$ *time, range queries take* $O(n^{1-1/d}\log^{1/d} n + k)$ *time (where k in the number of points reported), and divided k-d trees can be split and concatenated on the first and second coordinate in* $O(n^{1-1/d}\log^{1/d} n)$ *time, and on the other coordinates in* $O(n^{1-1/d}\log^{1+1/d} n)$ *time. The concatenable environment uses* $O(n)$ *space.*

6 Concluding remarks and open problems

In this paper a variant of the k-d tree, called a divided k-d tree, has been described for storing two-dimensional objects, in particular points, in the plane. It allows for insertions and deletions in $O(\log n)$ worst-case time, which is superior to the insertion and deletion time bounds of normal k-d trees, quad-trees and other data structures for range queries such as range trees. Also, its update and rebalance algorithms are simpler than for the other trees. The divided k-d tree allows for exact match queries in $O(\log n)$ time and range queries in $O(\sqrt{n\log n} + k)$ time. Because dynamic quad- and k-d trees cannot be kept well-balanced without permitting high update time bounds, the range query time for divided k-d trees is better than for dynamic quad- and k-d trees. Like quad- and k-d trees, the divided k-d tree uses linear space and can be built in $O(n\log n)$ time.

Also, an environment for divided k-d trees has been constructed in which they can be split and concatenated on both coordinates. This can be useful in graphics packages and database systems if one wants to work on a subset (orthogonal range) of the complete set of points (items) in the structure.

The divided k-d tree generalizes to higher dimensions, in which case only the range query time bound increases. Range queries on d-dimensional divided k-d trees take $O(n^{1-1/d}\log^{1/d} n + k)$ time. Also, a concatenable environment can be made such that we can split and concatenate on every coordinate. Again, the multi-dimensional divided k-d tree is far superior to other dynamic versions of quad- or k-d trees.

In particular in graphics applications k-d trees are often used to store other objects than points. For many such applications divided k-d trees can be used as well, improving the (dynamic) behaviour.

A number of open problems do remain. An important question is whether $O(\sqrt{n \log n})$ bounds for split and concatenate can be obtained, where n is the actual number of points in the tree rather than the size of all trees in the environment. Another question one could ask is whether there are two-dimensional trees for which better bounds for split and concatenate can be found. In particular, one could wonder whether range trees could be split and concatenated efficiently. Some general solutions to this problem can be found in [8].

References

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] Bentley, J.L., Multidimensional binary search trees used for associated searching, *C. ACM* 18 (1975) pp. 509-517.

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

[4] Bentley, J.L., Decomposable searching problems, *Inf. Proc. Lett.* 8 (1979) pp. 244-251.

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

[6] Kersten, M.L., and P. van Emde Boas, *Local optimizations of quad-trees*, Techn. Rep. IR-51, Free University Amsterdam, 1979.

[7] Knuth, D.E., *The art of computer programming, vol. 3: sorting and searching*, Addison-Wesley, Reading, Mass., 1973.

[8] van Kreveld, M.J., and M.H. Overmars, *Concatenable structures for decomposable problems*, Techn. Rep. RUU-CS-89-16, University of Utrecht, 1989.

[9] Lueker, G.S., A data structure for orthogonal range queries, *Proc. 19th IEEE Symp. on Foundations of Comp. Science,* 1978, pp. 28-34.

[10] Overmars, M.H., *The design of dynamic data structures*, Lect. Notes in Comp. Science 156, Springer-Verlag, Heidelberg, 1983.

[11] Overmars, M.H., and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching problems, *Inf. Proc. Lett.* 12 (1981) pp. 168-173.

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