Maintaining range trees in secondary memory Part I: Partitions

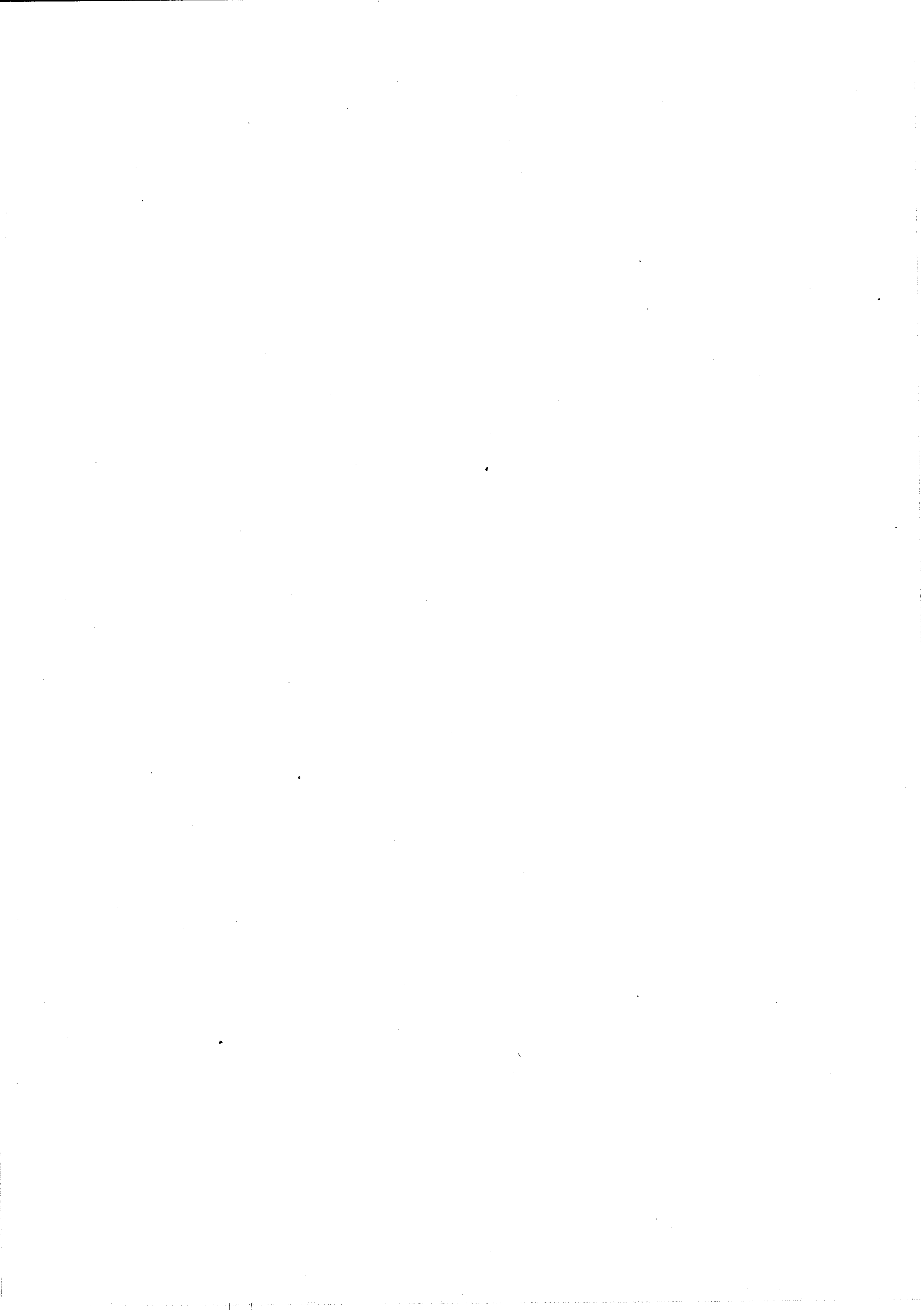
Mark H. Overmars, Michiel H.M. Smid,
Mark T. de Berg and Marc L. van Kreveld

RUU-CS-87-20
November 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands



Maintaining Range Trees in Secondary Memory Part I: Partitions

Mark H. Overmars* Michiel H.M. Smid† Mark T. de Berg*
Marc J. van Kreveld*

November 1987

Abstract

Range trees can be used for solving the orthogonal range searching problem, a problem that has applications in e.g. databases and computer graphics. We study the problem of storing range trees in secondary memory. To this end, we have to partition range trees into parts that can be stored in consecutive blocks in secondary memory. This paper, which is the first part in a series of two, gives a number of partition schemes that limit the part sizes and the number of disk accesses necessary to perform updates and queries. We show e.g., that for each fixed positive integer k, there is a partition of a two-dimensional range tree into parts of size $O(n^{1/k})$, such that each update requires at most $k(2k+1)$ disk accesses, and each query requires at most $4k(2k+1) - 4 + 2t$ disk accesses, where t is the number of answers to the range query. In Part II of this paper, lower bounds are given, which show that many of our partition schemes are optimal.

*Department of Computer Science, University of Utrecht, P.O.Box 80.012, 3508 TA Utrecht, The Netherlands.
†Department of Computer Science, University of Amsterdam, Nieuwe Achtergracht 166, 1018 WV Amsterdam, The Netherlands. This author was supported by the Netherlands Organisation for the Advancement of Pure Research (ZWO).

1 Introduction

A substantial part of the research in the theory of data structures is concerned with the design of structures and algorithms solving searching problems. In a searching problem, a question (also called a query) is asked about an object x with respect to a given set S of objects. An example is the orthogonal range searching problem.

Definition 1 *Let S be a set of points in d-dimensional space, and let* $([x_1 : y_1], [x_2 : y_2], \ldots, [x_d : y_d])$ *be some hyperrectangle. The* orthogonal range searching problem *asks for all points* $p = (p_1, p_2, \ldots, p_d)$ *in S, such that* $x_1 \leq p_1 \leq y_1, x_2 \leq p_2 \leq y_2, \ldots, x_d \leq p_d \leq y_d$.

The range searching problem has applications in e.g. computer graphics and database design. As an example, consider a salary administration, in which the information for each registered person includes age and salary. We can view each person as a point in 2-dimensional space, with as first coordinate the age, and as second coordinate the salary. Then a question like "give all persons with age between 20 and 25, having a salary between \$ 30,000 and \$ 35,000 a year" is an example of a range query.

A solution to a searching problem consists of a data structure, representing the set S, together with an algorithm that answers queries efficiently. Often, the efficiency of such data structures is caused by the facts that they are dynamic (i.e. they can be maintained efficiently if points are inserted or deleted in the set S), and that they can be stored entirely in main memory.

In this paper, however, we consider the case, where the data structure is too large to fit in core and, hence, has to be stored in secondary memory (a situation that very often occurs in databases). Then, in order to answer queries and to perform updates, parts of the data structure have to be transported from secondary memory to core, and vice versa. Therefore, it is necessary to partition the data structure into parts, such that a query or an update passes through only a small number of parts, each of which has small size (hence only a small amount of data has to be transported).

The partitioning of data structures also has the following important application. Suppose our data structure can be stored entirely in main memory. Then after a system crash, or as a result of errors in software, the contents of main memory can get lost. In that case, the data structure has to be reconstructed from the information stored in the safe secondary memory. A solution to this problem is to store in secondary memory a copy of the data structure. Then after a system crash, we just transport this copy to main memory. Clearly, also in this application it is useful to partition the data structure, such that the copy in secondary memory can be maintained efficiently. For more information about this reconstruction problem, we refer the reader to Smid et al. [11].

In order to be able to analyze the efficiency of partitions, we have to make assumptions how secondary memory is organized. We assume in this paper, that the file in secondary memory is divided into blocks of some fixed size. There is the ability of direct block access: it is possible to access a block directly, provided its physical address is known. Furthermore, it is possible to replace a block by another block, or a number of (physically) consecutive blocks by at most the same number of blocks. Finally, a new block, or a number of consecutive new blocks, can be added at the end of the file.

Now suppose we have partitioned our data structure into parts. Then we store this structure in secondary memory, by putting each part of the partition in a number of (physically) consecutive blocks. A query or an update is performed by successively transporting the blocks, through which the query or update passes, to core and vice versa. We express the complexity of this query/update procedure by:

(i) The number of seeks that has to be done: If we transport a number of consecutive blocks, we have to do one seek. So the number of seeks to be done is equal to the number of parts of the partition which are involved in the query/update process.

(ii) The total amount of memory that has to be transported.

Note that the seek time normally is very high compared to the time required to transport data. Hence it is essential to limit the number of seeks as much as possible. Also, note that if two parts are stored in consecutive blocks, these two parts can be transported requiring only one seek. That is, the number of seeks depends on the way the parts are stored in secondary memory. However, we assume here that the number of seeks necessary to perform a query or an update is equal to the number of parts through which the query or update passes.

Definition 2 *A partition of a dynamic data structure, representing a set of n points, is called an* $(F(n), G(n), H(n))$-partition, *if:*

1. *Each part has size at most $F(n)$.*
2. *There are $O(S(n)/F(n))$ parts, where $S(n)$ is the amount of space required to store the data structure.*
3. *Each query passes through at most $G(n)$ parts.*
4. *Each update passes through at most $H(n)$ parts.*

Note that it follows from *1.* that the number of parts is $\Omega(S(n)/F(n))$. In most cases, we will only be able to prove that an update passes through at most $H(n)$ parts on the average.

The relation of this definition to the above should be clear. It states, that we can store the data structure in secondary memory, such that a query requires at

most $G(n)$ seeks and $F(n)G(n)$ data transport. Also, an update takes at most $H(n)$ seeks and $F(n)H(n)$ data transport.

In this paper, which is the first part in a series of two, we study partition schemes for range trees (see e.g. [2,5,12]), a data structure that answers range queries efficiently. A number of trade-offs between the number of disk accesses (seeks) versus amount of memory that has to be transported are presented. In Part II (cf. Smid and Overmars [10]), several lower bounds for partitions are given, from which it follows that many partition schemes in this paper are optimal (in order of magnitude).

We have to remark that we take a known data structure as our starting point, namely a range tree, and we investigate how it can be partitioned as efficiently as possible. This in contrast to e.g. B-trees (see Bayer and McCreight [1], Comer [4]) or Grid Files (see Nievergelt et al. [6]), which are data structures that are designed especially to be stored in secondary memory. In some cases, however, as in Section 3.2, we also follow this latter approach, in order to get a variation of a range tree that can be partitioned very efficiently.

The paper is organized as follows. In Section 2 we define the basic concepts needed in the rest of the paper, namely BB[α]-trees and range trees. In Section 3 several efficient partitions of two-dimensional range trees are given. To this end, we modify the definition of range trees somewhat, by requiring extra balance conditions. Also, we change range trees, to get a new data structure for the orthogonal range searching problem, having the same performances as ordinary range trees, for which very efficient partition schemes exist. In Section 4 we generalize the results of Section 3 to the multi-dimensional case. In Section 5 we consider storage management in secondary memory. Finally, in Section 6 we give some concluding remarks.

To finish this section, we introduce some notations. First, logarithms, and powers of logarithms, are written in the usual way, i.e., we write $\log n$, $(\log n)^2$, etc. (in this paper all logarithms are to the base two). Furthermore, the k-th iterated logarithm is written as follows. If $k = 1$, then $(\log)^1 n = \log n$. If $k > 1$, then $(\log)^k n = \log\left((\log)^{k-1} n\right)$. The function $\log^* n$ is defined by $\log^* n = \min\{k \geq 1 | (\log)^k n \leq 1\}$.

2 Range trees

In this section we recall the definition of range trees, and we give query and update algorithms for them. First, we define BB[α]-trees, as introduced by Nievergelt and Reingold:

Definition 3 *([7]) Let α be a real number, $2/11 < \alpha \leq 1 - 1/2\sqrt{2}$. A binary tree is called a* BB[α]-tree, *if for each internal node v, the number of leaves in the left*

subtree of v divided by the total number of leaves below v lies in between α and $1-\alpha$.

Obviously, in a BB[α]-tree a similar balance condition holds for the right subtree of each internal node. In this paper, BB[α]-trees are used as leaf search trees. That is, if we want to use a BB[α]-tree T to represent a set S of real numbers, we store the elements of S in sorted order in the leaves of T. Internal nodes contain information to guide searches in the tree. The following theorem gives the complexity of a BB[α]-tree (the proof can be found in Blum and Mehlhorn [3]).

Theorem 1 *Suppose the set S contains n elements. Then a BB[α]-tree for S requires $O(n)$ space, and can be built in $O(n \log n)$ time. Insertions and deletions can be performed in $O(\log n)$ time, by means of single and double rotations. Using this tree, one-dimensional range queries can be performed in time $O(\log n + t)$, where t is the number of reported answers.*

BB[α]-trees are the building blocks of range trees, which we define now (cf. Bentley [2], Lueker [5], Willard and Lueker [12]).

Definition 4 *Let S be a subset of the d-dimensional real vector space. A d-*dimensional range tree *T, representing the set S, is defined as follows.*

1. *If $d = 1$, then T is a BB[α]-tree, containing the elements of S in sorted order in its leaves.*
2. *If $d > 1$, then T consists of a BB[α]-tree, called the* main tree, *which contains in its leaves the elements of S, ordered according to their first coordinates. Furthermore, each internal node v of this main tree contains an* associated structure, *which is a $(d-1)$-dimensional range tree for those elements of S which are in the subtree rooted at v, taking only the second to d-th coordinate into account.*

Let T be a range tree, representing the set S, and let v be a node of T (v is a node of the main tree, or of an associated structure, or of an associated structure of an associated structure, etc.). Let S_v be the set of those points of S which are in the subtree of v. Then node v is said to *represent* the set S_v.

E.g., a 2-dimensional range tree for set S consists of a BB[α]-tree, containing in its leaves the points of S ordered according to their x-coordinates. Let v be an internal node of this tree, and let S_v be the subset of S represented by v. Then node v contains a BB[α]-tree, representing the set S_v, ordered according to their y-coordinates.

Range queries are solved as follows. Let $([x_1 : y_1], [x_2 : y_2], \ldots, [x_d : y_d])$ be a query rectangle. Then we begin by searching with both x_1 and y_1 in the main tree. Assume w.l.o.g. that $x_1 < y_1$. Let u be that node in the main tree for which x_1 lies in the left subtree of u, and y_1 lies in the right subtree of u. Then we have to

perform a range query with the remaining $d-1$ coordinates on all points that lie between x_1 and y_1 in T. It is not too difficult to see that it is sufficient to perform recursively $(d-1)$-dimensional range queries in the associated structures of the right sons of those nodes v on the path from u to x_1 for which the search proceeds to the left son of v, and in the associated structures of the left sons of those nodes w on the path from u to y_1 for which the search proceeds to the right son of w. Clearly, there are $O(\log n)$ such nodes v and w. The answer to the entire query is the union of the answers of these queries. It follows, by induction on d, that the time to answer a query is $O((\log n)^d + t)$, where t is the number of reported answers. For details, see e.g. [8].

After an update of the set S, the range tree can be maintained using the partial rebuilding technique (cf. Lueker [5], Overmars [8]): Suppose we want to insert or delete point p in the range tree. Then we search with p in the main tree to locate its position among the leaves, and we insert or delete p in all the associated structures we encounter on our search path (if these associated structures are one-dimensional range trees, we apply the usual insertion/deletion algorithm for BB[α]-trees; otherwise we use the same procedure recursively). Next, we insert or delete p among the leaves of the main tree, and we walk back to the root. During this walk, we locate the highest node v which does not satisfy the balance condition of Definition 3 anymore. Then we rebalance at node v by rebuilding the entire structure rooted at v as a perfectly balanced range tree (*perfectly balanced* means that for each internal node, the number of leaves in the left resp. right subtree differ by at most one). Note that if node v is the root of the main tree, we have to rebuild the entire range tree, which takes $O(n(\log n)^{d-1})$ time. However, in this case $\Omega(n)$ updates must occur before we have again to rebuild the entire structure. In fact, Lueker [5] has shown the following: Let v be a node in a range tree which is in perfect balance. Let n_v be the number of points represented by v, at the moment v gets out of balance. Then there must have been $\Omega(n_v)$ updates in the subtree of v. Using this result, it can be shown that the above sketched update algorithm takes amortized time $O((\log n)^d)$. The proof of the following theorem can be found in Lueker [5] and Overmars [8].

Theorem 2 *Let S be a set of n points in d-dimensional space. Then a d-dimensional range tree for set S can be built in $O(n(\log n)^{d-1})$ time, and requires $O(n(\log n)^{d-1})$ space. Using this tree, orthogonal range queries can be solved in time $O((\log n)^d + t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^d)$.*

In fact, Willard and Lueker [12] have shown that insertions and deletions can even be performed in time $O((\log n)^d)$ in the worst case.

3 Partitions of two-dimensional range trees

In this section, we study partitions of two-dimensional range trees. In order to be able to give efficient partition schemes, we have to modify the definition of range trees somewhat. First, we give some trivial partitions.

Theorem 3 *For a two-dimensional range tree, there exists an*

1. $(O(n \log n), 1, 1)$*-partition.*
2. $(O(1), O((\log n)^2+t), O((\log n)^2))$*-partition, where t is the number of answers to the query.*
3. $(O(n), O(\log n), O(\log n))$*-partition.*

Proof. *1* : Just use the entire tree as one part. *2* : Each node (either of the main tree, or of an associated structure) forms a part in its own. Since a query takes time $O((\log n)^2 + t)$, it passes through $O((\log n)^2 + t)$ parts. Similarly, an update passes through $O((\log n)^2)$ parts on the average. *3* : Each level of the main tree, together with its associated structures, forms a part. □

Clearly, the last partition of Theorem 3 is worse than the first one: We still have to transport an amount of $O(n \log n)$ data, and this requires $O(\log n)$ seeks rather than one.

3.1 Restricted partitions

The first type of partitions we study, are the so-called *restricted partitions*: In a restricted partition, only the main tree is partitioned into parts, whereas associated structures are never subdivided. In such a partition, a node of the main tree and its associated structure are contained in the same part. Remark that this makes the implementation of such partitions a lot easier. Also, in a restricted partition, parts will have size $\Omega(n)$, since the associated structure of the root of the main tree has size $\Theta(n)$.

First we give a restricted $(O(n), O(\log\log n), O(\log\log n))$-partition. The idea is as follows. Suppose we have a perfectly balanced range tree, i.e., for each internal node, the number of leaves in its left resp. right subtree differ by at most one. Now cut the main tree at level $\log\log n$. Each level (together with its associated structures) above level $\log\log n$ forms a part. Each such part has size $O(n)$: the associated structures on a fixed level are binary trees for a subset of the n points represented by the entire data structure, and each of these n points is in exactly one such binary tree. This gives us $O(\log\log n)$ parts, each of size $O(n)$. Each subtree having its root at level $\log\log n$, is a two-dimensional range tree, representing $O(n/\log n)$ points. Hence such a subtree has size $O((n/\log n)\log(n/\log n)) = O(n)$ and, hence, it can form a part. This gives us $O(\log n)$ parts, each of size

$O(n)$. So in total we have $O(\log n)$ parts of size $O(n)$, provided the tree is perfectly balanced. However, as soon as we insert or delete points, the tree is not perfectly balanced anymore. In fact, the number of points represented by a subtree having its root at level $\log\log n$ can become $\Omega((1-\alpha)^{\log\log n} n) = \Omega((1/2\sqrt{2})^{\log\log n} n) = \Omega(n/\sqrt{\log n})$. Hence such a subtree may have size $\Omega(n\sqrt{\log n})$, which is too large to form a part. Of course, we can cut the main tree at a lower level, i.e., a level $\geq \log\log n$. However, then the number of subtrees having their root at this level, and hence the number of parts, becomes too large.

In order to avoid that subtrees, having their root at level $\log\log n$, become too big, we modify the definition of range trees. Let S be a subset of the two-dimensional real vector space. We suppose that the points of $S = \{p_1 \leq p_2 \leq p_3 \leq \ldots \leq p_n\}$ are ordered according to their x-coordinates. Partition S into subsets $S_1 = \{p_1, p_2, \ldots, p_{h(n)}\}, S_2 = \{p_{h(n)+1}, \ldots, p_{2h(n)}\}, \ldots$, where $h(n) = \lceil n/\log n \rceil$.

Definition 5 *A* modified range tree, *representing the set S, is defined as follows.*

1. *Each set S_i is stored in an ordinary two-dimensional range tree T_i. Let r_i be the root of T_i. These roots are ordered according to $r_1 < r_2 < r_3 < \ldots$.*
2. *The roots r_i are stored in the leaves of a perfectly balanced binary tree T. Let v be a node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$ (v may be a leaf of T). Then v contains an associated structure, which is an ordinary one-dimensional range tree, representing the set $S_i \cup S_{i+1} \cup \ldots \cup S_j$, ordered according to their y-coordinates.*

Note that in this definition, the structure of a range tree is not changed, only the balance conditions are different. Hence in a modified range tree, range queries are solved in the same way as in ordinary range trees. An insertion or deletion of a point p is performed as follows. First we walk down tree T, to find the appropriate root r_i. During this walk we insert or delete p in all associated structures we encounter on our search path. Then we insert or delete p in T_i, using the update algorithm for ordinary range trees. Clearly, this procedure takes time $O((\log n)^2)$ on the average, provided each set S_i contains $\Theta(n/\log n)$ points. Suppose at the moment we build this structure, the set S contains n points. Then each set S_i (except for the "last" one) contains $\lceil n/\log n \rceil$ points. As soon as at least one set S_i contains either $1/2\lceil n/\log n \rceil$ or $2\lceil n/\log n \rceil$ points, we rebuild the entire data structure. That is, we partition the set S into subsets of size $\lceil m/\log m \rceil$, where m is the cardinality of S at that moment. So every $\Omega(n/\log n)$ updates, we have to rebuild the data structure at most once, and this takes $O(n\log n)$ time. It follows that the amortized update time of the modified range tree is $O((\log n)^2)$.

Theorem 4 *A modified range tree, representing n points, can be built in time $O(n\log n)$, and takes $O(n\log n)$ space to store. Range queries can be solved, using this tree, in time $O((\log n)^2 + t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^2)$.*

Proof. The bounds for the storage requirement, the building time and the query time can be proved in the same way as in Theorem 2. The bound for the update time follows from the above discussion. □

Hence the modified range tree has (asymptotically) the same complexity as an ordinary range tree. Observe that if we do not have to rebuild the structure, in T only associated structures are changed after an update (T itself is not changed).

Theorem 5 *For a modified range tree, there exists an* $(O(n), \log\log n + O(1), \log\log n + O(1))$*-partition.*

Proof. Each tree T_i represents $\Theta(n/\log n)$ points. So it has size $O(n)$ and, hence, it can form a part. This gives us $O(\log n)$ parts. Each level of the tree T, together with its associated structures, forms a part, again of size $O(n)$. Since tree T is perfectly balanced, it has depth $\log\log n + O(1)$. So this gives us $\log\log n + O(1)$ parts. A query passes through all levels of T, and through at most 2 trees T_i (since we also store associated structures in the leaves of T). Hence it passes through $\log\log n + O(1)$ parts. Also, an update passes through $\log\log n + O(1)$ parts, if we do not have to rebuild the data structure. If we have to rebuild the structure, $O(\log n)$ parts are involved. Since this has to be done at most once every $\Omega(n/\log n)$ updates, the average number of parts through which an update passes is $\log\log n + O(1) + O(\frac{(\log n)^2}{n}) = \log\log n + O(1)$. □

Theorem 6 *For a modified range tree, there exists an* $(O(n\log\log n), 3, 2 + o(1))$*-partition.*

Proof. The tree T forms a part in its own, of size $O(n\log\log n)$. Furthermore, we put sets of $\lceil\log\log n\rceil$ trees T_i together in one part. A query passes through at most 3 parts: the part containing tree T, and at most 2 parts containing trees T_i (again we use the fact that we also store associated structures in the leaves of T). Also, an update passes through exactly 2 parts, if the data structure is not rebuilt. Since rebuilding of the structure has to be done at most once every $\Omega(n/\log n)$ updates, and since then $O(\log n/\log\log n)$ parts are involved, the average number of parts through which an update passes is $2 + O(\frac{(\log n)^2}{n\log\log n}) = 2 + o(1)$. □

Remark. The partition of the above theorem is optimal, in the following sense: In each restricted partition of a two-dimensional range tree, such that each update passes through at most 2 parts, there is a part of size $\Omega(n\log\log n)$. For a proof, the reader is referred to Part II of this paper [10].

Next we shall improve Theorem 5 considerably. First we give a lemma. We remind the reader to our notation $(\log)^k n$ for the k-th iterated logarithm, and to the definition of the function $\log^* n$ (see Section 1).

Lemma 1 *Let the integer sequence (a_k) be given by $a_0 = 0, a_{k+1} = 2^{a_k} + a_k$, for $k \geq 0$. Let n and d be integers, such that $d = \log\log n + O(1)$ (we assume that n is sufficiently large). Let $m = \min\{i \geq 0 | a_i > d\}$. Then $m \leq \log^* n + O(1)$.*

Proof. We show that

$$(\log)^i d \geq a_{m-i-1} \quad \text{for} \quad i = 1, 2, \ldots, m-3. \tag{1}$$

By definition of m, we have $d \geq a_{m-1} = 2^{a_{m-2}} + a_{m-2} \geq 2^{a_{m-2}}$. Hence $(\log)^1 d = \log d \geq a_{m-2}$. Now let $1 \leq i < m-3$, and suppose that $(\log)^i d \geq a_{m-i-1}$. Then

$$(\log)^i d \geq a_{m-i-1} = 2^{a_{m-i-2}} + a_{m-i-2} \geq 2^{a_{m-i-2}}.$$

Since $a_{m-i-2} \geq 0$, we have $(\log)^i d \geq 1$. Hence $(\log)^{i+1} d$ exists, and $(\log)^{i+1} d \geq a_{m-i-2}$, which proves (1).

Now take $i = m-3$ in (1). Then $(\log)^{m-3} d \geq a_2 = 3$, and hence $(\log)^{m-2} d > 1$. By the definition of $\log^* d$, it follows that $m - 2 < \log^* d$. Then, by using the relations $\log^*(N + O(1)) = \log^* N + O(1)$, and $\log^* N = 1 + \log^*(\log N)$, we get

$$m - 2 < \log^* d = \log^*(\log\log n + O(1)) = \log^* n + O(1).$$

□

Theorem 7 *For a modified range tree, there exists an $(O(n), 4\log^* n + O(1), \log^* n + O(1))$-partition.*

Proof. Since we want to partition the data structure into parts of size $O(n)$, each tree T_i can form a part. This gives us $O(\log n)$ parts.

So we are left with the tree T. We first sketch how this tree is partitioned. The root of T, together with its associated structure, forms a part. This removes the top level of T. Now consider the two sons v and w of the root. Look at the subtree consisting of v and its two sons. It takes, together with its associated structures, $O(n)$ storage and, hence, can form a part. Similarly for w. This removes two more levels of T; so we are left with 8 sons. For each of these sons u, we make a part consisting of the tree with root u, of depth 8. This subtree, of course with its associated structures, uses $O(n)$ space. We now have removed 11 levels. So we are left with 2^{11} sons. For each son, we take a subtree of depth 2^{11}, with associated structures, which takes $O(n)$ storage. Next we are left with $2^{2^{11}+11}$ sons, etc. The reader should note that the tree T is (and remains) perfectly balanced. So a node on level i indeed represents $\Theta(n/2^i)$ points (cf. the discussion at the beginning of this section).

We will describe the above more precisely. Let $a_0 = 0$, and $a_{k+1} = 2^{a_k} + a_k$ for $k \geq 0$. Let d be the depth of tree T (d is the number of nodes in the longest path in T from the root to a leaf). Since T is perfectly balanced, we have $d =$

$\log\log n + O(1)$. Let $m = \min\{i \geq 0 | a_i > d\}$. Then it follows from Lemma 1, that $m \leq \log^* n + O(1)$. Now tree T is partitioned as follows. For each $k, 0 \leq k \leq m-1$, there are 2^{a_k} parts. Each such part is a subtree of T, together with its associated structures, having its root at level a_k, of depth 2^{a_k}. Clearly, each part has size $O(n)$. Furthermore, the number of parts in which T is partitioned is

$$\sum_{k=0}^{m-1} 2^{a_k} = O(2^{a_{m-1}}) = O(2^d) = O(2^{\log\log n + O(1)}) = O(\log n).$$

Now let $([x_1 : y_1], [x_2 : y_2])$ be a query rectangle, and consider the path in T from the root to x_1. Look at a node v through which this path passes, and let Π be the part of the partition containing this node. If this path proceeds to the left son, we have to search the associated structure of the right son of v. If v is not at the bottom level of Π, these left and right sons are also contained in Π. Otherwise, we have to pass through 2 different parts. So, since the number of parts through which this left path passes is m, the left path of the entire query passes through at most $2m + 1$ parts ($2m$ parts in tree T, and one part containing a tree T_i). Hence the number of parts through which a query passes is at most $4m + 2$, which is bounded above by $4 \log^* n + O(1)$. Finally, an update passes through $m \leq \log^* n + O(1)$ parts of T and through one part containing a tree T_i, if we do not have to rebuild the data structure. If we take the cost of rebuilding into account, we see that on the average $\log^* n + O(1) + O(\frac{(\log n)^2}{n}) = \log^* n + O(1)$ parts are involved in an update. □

This is an interesting result, because it means that we can query and maintain a modified range tree, stored in secondary memory, by transporting $O(\log^* n)$ parts of size $O(n)$. Observe that although $\log^* n$ goes to infinity as n does, for all practical values of n, we have $\log^* n \leq 5$.

Remark. Also this partition turns out to be optimal, now in the following sense. Suppose we have a restricted partition of a two-dimensional range tree into parts of size $O(n)$. Then there is an update which passes through $\Omega(\log^* n)$ parts. For a proof, see Part II [10].

To finish this section on restricted partitions, we generalize Theorem 6. Therefore, we again have to change the definition of range trees.

Definition 6 *Let $S = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. A k-fold* modified range tree *is defined as follows.*

1. *For $k = 1$, a 1-fold modified range tree is an ordinary range tree.*

2. *Let $k > 1, m = \lceil n \frac{(\log)^k n}{(\log)^{k-1} n} \rceil$. Partition S into sets $S_1 = \{p_1, p_2, \ldots, p_m\}, S_2 = \{p_{m+1}, \ldots, p_{2m}\}, \ldots$. Then a k-fold modified range tree consists of the following. Each set S_i is stored in a $(k-1)$-fold modified range tree T_i. Let r_i be the root of T_i. These roots are ordered according to $r_1 < r_2 < r_3 < \ldots$. We store these roots in a perfectly balanced binary leaf search tree T. Let v be a node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$ (v may be a leaf of T). Then v contains an associated structure, which is a one-dimensional range tree for the set $S_i \cup S_{i+1} \cup \ldots \cup S_j$, ordered according to their y-coordinates.*

Note that also in this case, the structure of a range tree is not changed, only the balance conditions are different. Hence the query algorithm in a k-fold modified range tree is similar to that of an ordinary range tree. The update algorithm is similar to that of a modified range tree. In order to keep the structure balanced, we completely rebuild it as soon as at least one set S_i contains either $1/2\,m$ or $2\,m$ points. So every $\Omega(m)$ updates, we have to rebuild the data structure at most once. The following theorem shows that a k-fold modified range tree has the same performances as an ordinary range tree.

Theorem 8 *A k-fold modified range tree, representing n points, can be built in time $O(n \log n)$, and takes $O(n \log n)$ space to store. In this tree, range queries can be solved in time $O((\log n)^2 + t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^2)$.*

Proof. The proof is by induction on k. For $k = 1$, the theorem follows from Theorem 2. So let $k > 1$, and suppose the theorem is proved for $k-1$. Each tree T_i requires $O(m \log m)$ space, where $m = \lceil n \frac{(\log)^k n}{(\log)^{k-1} n} \rceil$. Since there are $O(\frac{(\log)^{k-1} n}{(\log)^k n})$ such trees, they take together an amount of space bounded by

$$O\left(m \log m \;\frac{(\log)^{k-1} n}{(\log)^k n}\right) = O(n \log n).$$

Each level of tree T takes $O(n)$ space. Since T has depth

$$O(\log \frac{n}{m}) = O\left(\log(\frac{(\log)^{k-1} n}{(\log)^k n})\right) = O((\log)^k n),$$

it requires $O(n(\log)^k n)$ space. Hence the entire data structure takes $O(n \log n)$ space. The bounds on the building time and the query time can be proved in an analogous way. An insertion or deletion of a point p is performed as follows. First we walk down tree T, to find the appropriate root r_i. During this walk we insert or delete p in all associated structures we encounter on the search path. Then we insert or delete p in T_i, using the update algorithm for a $(k-1)$-fold modified range tree. This procedure takes amortized time $O((\log)^k n \log n + (\log m)^2) =$

$O((\log n)^2)$, provided the data structure is not rebuilt. Since the structure has to be rebuilt at most once every $\Omega(m)$ updates, and since this rebuilding takes time $O(n \log n)$, it follows that the amortized update time of the k-fold modified range tree is $O((\log n)^2)$. □

The following theorem generalizes Theorem 6.

Theorem 9 *For a k-fold modified range tree, there exists an $(O(n(\log)^k n), 2k - 1, k + o(1))$-partition.*

Proof. Again, the proof is by induction on k. For $k = 1$, the claim is obvious. So let $k > 1$, and suppose the theorem is proved for $k - 1$. We saw in the proof of Theorem 8, that the tree T takes $O(n(\log)^k n)$ space. Hence it can form a part. Each tree T_i is a $(k-1)$-fold modified range tree, representing $\Theta(m)$ points, where $m = \lceil n \frac{(\log)^k n}{(\log)^{k-1} n} \rceil$. We partition this tree T_i, recursively, into parts of size $O(m(\log)^{k-1} m) = O(n(\log)^k n)$, such that each query passes through at most $2(k-1) - 1$ parts, and each update passes through at most $(k-1) + o(1)$ parts on the average. Then the entire data structure is partitioned into parts of size $O(n(\log)^k n)$. Clearly, an update of the entire data structure passes through $k + o(1)$ parts, if we do not have to rebuild the structure. Since the structure is rebuilt at most once every $\Omega(m)$ updates, and since in that case $O(\frac{\log n}{(\log)^k n})$ parts are involved in the update, it follows that each update passes through at most $k + o(1) + O(\frac{\log n}{(\log)^k n} \frac{1}{m}) = k + o(1)$ parts on the average. So we are left with the bound on the number of parts through which a query passes. Let $h(k)$ be the maximal number of parts through which the "left path" of a query in a k-fold modified range tree passes. Then $h(1) = 1, h(k) \leq 1 + h(k-1)$ for $k > 1$, since we also store associated structures in the leaves of T. Hence $h(k) \leq k$. It follows that a query in the entire data structure passes through at most $2h(k) - 1 \leq 2k - 1$ parts:$h(k)$ parts for the left path, $h(k)$ for the right path, -1 since we counted the top part of the tree twice. This proves the theorem. □

Note that the value of k should be less than or equal to $\log^* n$, since otherwise $(\log)^k n \leq 0$, or is not even defined. Hence in practical situations, we have $k \leq 5$.

Remark. Again, the above partition is optimal: In Part II [10] it is shown, that in each restricted partition of a two-dimensional range tree, such that each update passes through at most k parts, there is a part of size $\Omega(n(\log)^k n)$.

3.2 Changing range trees to make them partitionable

In the preceding section, we defined the modified range tree. It was shown that for such a range tree, there exists an $(O(n), O(\log^* n), O(\log^* n))$-partition. The purpose of this section, is to show that it is possible to change range trees in such

a way that they can be partitioned into a restricted $(O(n), 3, 2 + o(1))$-partition. Also, the new data structure has asymptotically the same complexity as an ordinary range tree.

Let $S = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. We partition the set S into subsets $S_1 = \{p_1, \ldots, p_{h(n)}\}, S_2 = \{p_{h(n)+1}, \ldots, p_{2h(n)}\}, \ldots$, where $h(n) = \lceil n/\log n \rceil$.

Definition 7 *A* reduced range tree *representing the set S is defined as follows.*

1. *Each set S_i is stored in an ordinary two-dimensional range tree T_i. Let r_i be the root of T_i.*
2. *These roots r_i are stored in the leaves of a perfectly balanced binary tree T.*

So in a reduced range tree, nodes that are high in the main tree (i.e. nodes representing many points) do not contain an associated structure. As we will see, this does not increase the query time asymptotically. First we give the query algorithm for a reduced range tree. Let $([x_1 : y_1], [x_2 : y_2])$ be a query rectangle. Then we search with x_1 and y_1 in tree T for the appropriate roots, say r_i resp. r_j. If $i = j$, then we perform a query, with the rectangle $([x_1 : y_1], [x_2 : y_2])$, in the range tree T_i. Now suppose that $i < j$. Then we perform queries, with the strip $([x_1 : \infty], [x_2 : y_2])$ in tree T_i, and with $([-\infty : y_1], [x_2 : y_2])$ in tree T_j. Furthermore, we perform one-dimensional range queries, with query interval $[x_2 : y_2]$ in the associated structures of the roots of the trees $T_{i+1}, \ldots, T_{j-1}$.

Suppose we want to insert or delete point p in a reduced range tree. Then we walk down tree T, to find the appropriate root r_i, and we insert or delete p in the tree T_i, using the update algorithm for ordinary range trees. Just as for modified range trees, we completely rebuild the data structure as soon as one set S_i contains either $1/2\lceil n/\log n\rceil$ or $2\lceil n/\log n\rceil$ points.

Theorem 10 *A reduced range tree, representing n points, can be built in time $O(n \log n)$, and takes $O(n \log n)$ space to store. In this tree, range queries can be solved in time $O((\log n)^2 + t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^2)$.*

Proof. The bounds on the building time, the space requirement and the update time can be proved in the same way as for modified range trees (cf. Theorem 4). Consider the query algorithm for reduced range trees as described above. The time to find the roots r_i and r_j is proportional to the depth of tree T, which is $O(\log \log n)$. If $i = j$, we have to query the tree T_i, which takes time $O((\log \frac{n}{\log n})^2) = O((\log n)^2)$. If $i < j$, we query the trees T_i and T_j, which takes time $O((\log n)^2)$. Furthermore, the one-dimensional range queries in the associated structures of the roots of $T_{i+1}, \ldots, T_{j-1}$ take time $O(\log n \log \frac{n}{\log n}) = O((\log n)^2)$, since there are $O(\log n)$ such associated structures, and each has query time

$O(\log \frac{n}{\log n})$. Of course we have to add $O(t)$ to the total query time for reporting the answers. This proves the theorem. □

It follows that we have a new data structure for the orthogonal range searching problem, having the same performances as an ordinary range tree. The next theorem shows that this new data structure can be partitioned efficiently.

Theorem 11 *For a reduced range tree, there exists an $(O(n), 3, 2+o(1))$-partition.*

Proof. We put the tree T, together with the associated structures of the roots of the trees T_i in one part. This part has size $O(\log n + \log n \frac{n}{\log n}) = O(n)$. Also, each tree T_i, without the associated structure of its root, is put in one part. This gives us $O(\log n)$ parts, each of size $O(n)$. Clearly, a query passes through at most 3 parts. Also, if the data structure is not rebuilt, an update passes through at most 2 parts. If the structure is rebuilt, which happens at most once every $\Omega(n/\log n)$ updates, $O(\log n)$ parts are involved. Hence, on the average, an update passes through $2 + o(1)$ parts of the partition. □

Remark. We remarked after Theorem 6, that if a two-dimensional range tree is partitioned, in the restricted sense, such that each update passes through at most 2 parts, there must be a part of size $\Omega(n \log\log n)$. This is not in conflict with the partition of Theorem 11: A reduced range tree does not have the structure of a range tree, and therefore the lower bound does not apply. (Strictly speaking, the partition of Theorem 11 is not restricted: the associated structure of the root of T_i is not contained in the same part as the root itself. However, the data structure can easily be adapted such that the partition is restricted.)

3.3 General partitions

In this section we also partition the associated structures of the nodes of the main tree. This makes it possible to partition the structure in parts of size $o(n)$. For similar reasons as in Section 3.1, we again have to modify the definition of range trees.

Let $S = \{p_1 \leq p_2 \leq p_3 \leq \ldots \leq p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. Partition the set S into subsets $S_1 = \{p_1, p_2, \ldots, p_{h(n)}\}, S_2 = \{p_{h(n)+1}, \ldots, p_{2h(n)}\}, \ldots$, where $h(n) = \lceil \sqrt{n} \rceil$.

Definition 8 *A* balanced range tree, *representing the set S, is defined as follows.*

1. *Each set S_i is stored in an ordinary two-dimensional range tree T_i. Let r_i be the root of T_i. As usual, these roots are ordered according to $r_1 < r_2 < r_3 < \ldots$.*
2. *The roots r_i are stored in a perfectly balanced binary leaf search tree T. Let v be a node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$ (v may be a leaf of*

T). Then v contains an associated structure representing the set $S_{ij} = S_i \cup S_{i+1} \cup \ldots \cup S_j$, which has the following form. Let m be the cardinality of S_{ij} (note that $m = \Omega(\sqrt{n})$). We order $S_{ij} = \{q_1 \leq q_2 \leq \ldots \leq q_m\}$ according to their y-coordinates, and we partition it into sets $S_{ij1} = \{q_1, \ldots, q_{h(n)}\}$, $S_{ij2} = \{q_{h(n)+1}, \ldots, q_{2h(n)}\}, \ldots$. Then we store each set S_{ijk} in the leaves of a BB[α]-tree T_{ijk} (ordered of course according to their y-coordinates). The roots of the trees T_{ijk} are stored in the leaves of a perfectly balanced binary tree T_{ij}. Now the trees T_{ij} and $T_{ijk}, k = 1, 2, \ldots$, together form the associated structure of node v.

Again, the structure of balanced range trees is the same as that of ordinary range trees. The query and update algorithms of balanced range trees are similar to that of modified range trees (see Section 3.1). Clearly, an update will take $O((\log n)^2)$ time, as long as the sizes of the sets S_i and S_{ijk} remain $\Theta(\sqrt{n})$. We rebuild the entire data structure as soon as at least one set S_i or S_{ijk} contains either $1/2 \lceil \sqrt{n} \rceil$ or $2 \lceil \sqrt{n} \rceil$ points. Note that this means that in the worst case we have to rebuild the structure after about $\sqrt{n}$ updates. So the average update time of a balanced range tree is $O(\sqrt{n} \log n)$.

Theorem 12 *A balanced range tree, representing n points, can be built in $O(n \log n)$ time, and takes $O(n \log n)$ space to store. Using this tree, range queries can be solved in time $O((\log n)^2 + t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O(\sqrt{n} \log n)$.*

Proof. The theorem can be proved in the same way as Theorem 2. The bound on the update time follows from the above. □

Theorem 13 *For a balanced range tree, there exists an $(O(\sqrt{n}), O(\log n + \frac{t}{\sqrt{n}}), O(\log n))$-partition, where t is the number of answers to the query.*

Proof. Consider the main tree of a tree T_i. Each level of this tree, together with its associated structures, forms a part. So each tree T_i is partitioned into $O(\log n)$ parts, each of size $O(\sqrt{n})$. This gives us for all trees T_i together $O(\sqrt{n} \log n)$ parts of size $O(\sqrt{n})$. The tree T, without its associated structures, forms a part in its own, also of size $O(\sqrt{n})$. So we are left with the associated structures of the tree T. Each such structure is partitioned into trees T_{ijk}, of size $O(\sqrt{n})$, and the tree T_{ij}, also of size $O(\sqrt{n})$ (in general the latter tree will be much smaller). Since tree T contains $O(\sqrt{n})$ leaves, and since it has depth $O(\log n)$, the associated structures together are partitioned into $O(\sqrt{n} \log n)$ parts, each of size $O(\sqrt{n})$. If we do not have to rebuild the structure, an update will pass through exactly one tree T_i (which is partitioned into $O(\log n)$ levels), through tree T, and on each level of T through exactly one tree T_{ij} and one tree T_{ijk}. Hence in this case an update passes through $O(\log n)$ parts of the partition. If we do have to rebuild the structure, which happens at most once every $\Omega(\sqrt{n})$ updates, $O(\sqrt{n} \log n)$ parts are involved

in the update. Hence, on the average, an update passes through $O(\log n)$ parts. A query passes through tree T (with associated structures) and through at most 2 trees T_i. Clearly, in such a tree T_i, the number of visited parts is bounded by $O(\log n)$. Now consider the left path of the query in tree T. Let v be a node on this path. If v is a right son, no associated structure has to be queried. If it is a left son, we have to query the associated structure of its right brother. The number of parts of this associated structure, through which the query passes, is $1 + O(1 + \frac{t'}{\sqrt{n}})$: one tree T_{ij} and $O(1 + \frac{t'}{\sqrt{n}})$ trees T_{ijk}, where t' is the number of answers found in this associated structure. Since there are $O(\log n)$ such nodes v, it follows that the entire query passes through $O(\log n + \frac{t}{\sqrt{n}})$ parts. □

We give now a class of two-dimensional range trees, which can be partitioned into parts, such that an update passes through at most 3 parts. These range trees depend on two parameters $g(n)$ and $h(n)$. In this way we obtain a trade-off between the sizes of parts versus the amortized update time.

Definition 9 *Let $g(n)$ and $h(n)$ be integer functions, such that $1 \leq g(n) \leq n, 1 \leq h(n) \leq n$, and $g(n)h(n) \geq n/\log n$. A $(g(n), h(n))$*-range tree *is defined as follows.*

1. *Let $S = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. We partition the set S into subsets $S_1 = \{p_1, \ldots, p_{g(n)}\}, S_2 = \{p_{g(n)+1}, \ldots, p_{2g(n)}\}, \ldots$. Order the points of S according to their y-coordinates. Let $S = \{q_1 \leq q_2 \leq \ldots \leq q_n\}$ be the resulting set. We partition this set into subsets $V_1 = \{q_1, \ldots, q_{h(n)}\}, V_2 = \{q_{h(n)+1}, \ldots, q_{2h(n)}\}, \ldots$.*
2. *Each set S_i is stored in an ordinary two-dimensional range tree T_i. Let r_i be the root of T_i.*
3. *These roots are stored in the leaves of a perfectly balanced binary tree T. Let v be a node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$. Then v represents the set $S_{ij} = S_i \cup S_{i+1} \cup \ldots \cup S_j$. Let $I_v = \{k | S_{ij} \cap V_k \neq \emptyset\}$. Node v contains an associated structure, representing the set S_{ij}, which has the following form. There is a* top tree *T_v, which is a $BB[\alpha]$-tree, containing the set I_v in its leaves. Furthermore, each leaf k of this top tree, contains a $BB[\alpha]$-tree T_{vk}, containing in its leaves the points of $S_{ij} \cap V_k$, ordered according to their y-coordinates.*

In the above definition, the condition $g(n)h(n) \geq n/\log n$ is to assure that the data structure uses $O(n \log n)$ space. Observe that the associated structure of a node v of the tree T contains the points of $\bigcup_{k \in I_v}(S_{ij} \cap V_k) = S_{ij}$, ordered according to their y-coordinates. Also, for such a node v, we have $|I_v| = O(n/h(n))$. Finally, if v is the root of tree T, the set I_v contains all values of indices for which there is a set V_k (therefore the top tree T_v associated with the root is, and remains, perfectly balanced).

Since $(g(n), h(n))$-range trees have the same structure as ordinary range trees, the query algorithm for this data structure will be clear. An insertion or deletion of a point p is performed as follows. First we walk down tree T, to find the appropriate root r_i. During this walk, we have to update all associated structures we encounter on the search path. The first associated structure we encounter is that of the root r of T. We search in its top tree T_r, to find the set V_k in which p has to be inserted or deleted. Then we update the corresponding tree T_{rk}. Now consider a non-root node v of T, on our search path. We search in the top tree T_v for k (note that we know the value of k). If k is present in this top tree, we insert or delete p in the tree T_{vk}. If T_{vk} becomes empty, we delete k from the top tree T_v. Otherwise, if k is not present in the top tree (in this case point p has to be inserted), we insert it into it, together with a tree T_{vk} containing p. Finally, point p is inserted or deleted in the appropriate range tree T_i. In order to keep the data structure balanced, we rebuild it as soon as one set S_i contains either $1/2\,g(n)$ or $2\,g(n)$ points, or as soon as one set V_j contains either $1/2\,h(n)$ or $2\,h(n)$ points. Note that this rebuilding has to be done at most once every $\Omega(\min(g(n), h(n)))$ updates. The following theorem gives the complexity of a $(g(n), h(n))$-range tree.

Theorem 14 *Let $g(n)$ and $h(n)$ be as before. A $(g(n), h(n))$-range tree, representing n points, can be built in $O(n \log n)$ time, and takes $O(n \log n)$ space to store. Using this tree, range queries can be solved in $O((\log n)^2 + t)$ time, where t is the number of reported answers. Insertions and deletions in this tree can be done in amortized time $O((\log n)^2 + \frac{n \log n}{\min(g(n), h(n))})$.*

Proof. Each tree T_i represents $O(g(n))$ points. Hence it has size $O(g(n) \log g(n))$. Since there are $O(n/g(n))$ such trees, together they take $O(n \log g(n))$ space. The tree T takes $O(n/g(n))$ space. Each top tree T_v, where v is a node of T, has size $O(n/h(n))$. Hence all top trees together have size $O(\frac{n}{g(n)} \frac{n}{h(n)})$. Consider a fixed level of T. The trees T_{vk} of the associated structures on this level together represent the set S, and, hence, they have size $O(n)$. Since T has depth $O(\log \frac{n}{g(n)})$, all these trees T_{vk} together take $O(n \log \frac{n}{g(n)})$ space. Hence the space needed to store the entire data structure is bounded by $O(n \log g(n)) + O(\frac{n}{g(n)} \frac{n}{h(n)}) + O(n \log \frac{n}{g(n)}) = O(n \log n)$, since $g(n)h(n) \geq n/\log n$. The bound on the building time can be proved in the an analogous way. In each associated structure of a node in tree T, one-dimensional range queries can be solved in time $O(\log \frac{n}{h(n)} + \log h(n)) = O(\log n)$. Clearly, one-dimensional range queries in an associated structure of a tree T_i take $O(\log g(n)) = O(\log n)$ time. To solve a two-dimensional range query, we have to solve $O(\log n)$ one-dimensional range queries in associated structures. It follows that the query time of the data structure is $O((\log n)^2 + t)$. So we are left with the update time. Suppose the data structure is not rebuilt. Clearly, the update of range tree T_i takes amortized time $O((\log g(n))^2)$, and only one such tree has to be updated. Furthermore, the update of an associated structure

in T takes $O(\log n)$ time. Since $O(\log \frac{n}{g(n)})$ such associated structures are updated, the total update time is $O((\log g(n))^2 + \log n \log \frac{n}{g(n)}) = O((\log n)^2)$. Every $\Omega(\min(g(n), h(n)))$ updates, the data structure is rebuilt at most once. Such a rebuilding takes $O(n \log n)$ time. So the amortized update time of the data structure is $O((\log n)^2 + \frac{n \log n}{\min(g(n),h(n))})$. This proves the theorem. □

Theorem 15 *Let $g(n)$ and $h(n)$ be as before. For a $(g(n), h(n))$-range tree, there exists an $(O(f(n)), 5 + O(\frac{t}{h(n)}), 3 + O(\frac{n \log n}{f(n)} \frac{1}{\min(g(n),h(n))}))$-partition, where t is the number of answers to the query, and $f(n) = \max\left(g(n) \log g(n), \frac{n}{g(n)} \frac{n}{h(n)}, h(n) \log \frac{n}{g(n)}\right)$.*

Proof. Each tree T_i forms a part. This gives us $O(n/g(n))$ parts of size $O(g(n) \log g(n))$. Next we put the tree T together with all top trees T_v in one part. Tree T has size $O(n/g(n))$. There are $O(n/g(n))$ top trees, and each of them has size $O(n/h(n))$. So this part has size $O(\frac{n}{g(n)}) + O(\frac{n}{g(n)} \frac{n}{h(n)}) = O(\frac{n}{g(n)} \frac{n}{h(n)})$. Finally, for each fixed k, the trees T_{vk}, for v a node of T, are put together in one part. Consider a level of T. Let $v_1, v_2, \ldots, v_m$ be the nodes on this level. The trees $T_{v_1 k}, \ldots, T_{v_m k}$ together represent the set V_k, which has size $O(h(n))$. So for fixed k, all trees T_{vk} together have size $O(h(n) \log \frac{n}{g(n)})$, since tree T has depth $O(\log \frac{n}{g(n)})$. Since there are $O(n/h(n))$ possible values for k, this gives us $O(n/h(n))$ parts of size $O(h(n) \log \frac{n}{g(n)})$.

To summarize, we have $O(n/g(n))$ parts of size $O(g(n) \log g(n))$, one part of size $O(\frac{n}{g(n)} \frac{n}{h(n)})$, and $O(n/h(n))$ parts of size $O(h(n) \log \frac{n}{g(n)})$. Then, in order to get the desired partition, we merge parts into $O(\frac{n \log n}{f(n)})$ new parts of size $O(f(n))$.

Now consider an insertion or a deletion of a point, such that the data structure is not rebuilt. Let V_k be the set in which the point is inserted or deleted. Then this update passes through exactly three parts: The part containing T and the top trees; the part containing the trees T_{vk}; and a part containing the appropriate range tree T_i. If the structure is rebuilt, $O(\frac{n \log n}{f(n)})$ parts are involved in the update. Since this has to be done at most once every $\Omega(\min(g(n), h(n)))$ updates, the average number of parts through which an update passes is at most $3 + O(\frac{n \log n}{f(n)} \frac{1}{\min(g(n),h(n))})$. The bound on the number of parts through which a query passes can be proved in the same way. □

As an example, take $g(n) = \lceil n/\log n \rceil$ and $h(n) = \lceil n/\log\log n \rceil$. Then we get a version of a range tree, having the same performances as an ordinary range tree:

Theorem 16 *Let $g(n) = \lceil n/\log n \rceil$ and $h(n) = \lceil n/\log\log n \rceil$. In a $(g(n), h(n))$-range tree, insertions and deletions can be performed in amortized time $O((\log n)^2)$. For this $(g(n), h(n))$-range tree, there exists an $(O(n), 5 + O(t \frac{\log\log n}{n}), 3 + o(1))$-partition , where t is the number of answers to the query.*

As another example, the following theorem chooses the functions $g(n)$ and $h(n)$, such that the sizes of parts are minimal.

Theorem 17 *Let $g(n) = h(n) = \lceil n^{2/3}/(\log n)^{1/3} \rceil$. In a $(g(n), h(n))$-range tree, insertions and deletions can be performed in amortized time $O(n^{1/3}(\log n)^{4/3})$. For this $(g(n), h(n))$-range tree, there exists an $(O((n \log n)^{2/3}), 5 + O(t \frac{(\log n)^{1/3}}{n^{2/3}}), 3 + o(1))$-partition, where t is the number of answers to the query.*

In the following section, we will show how the idea of Definition 9 can be generalized to range trees, which can be partitioned into arbitrary small parts, such that each update passes through $O(1)$ parts, and each query passes through $O(1 + t)$ parts, where t is the number of reported answers.

3.4 A partition with arbitrary small parts

In this section we give a partition of a two-dimensional range tree into parts of arbitrary small size, such that the number of seeks for an update is constant. In order to get such a partition, the structure of the range tree has to be changed slightly. In fact, only the associated structures will change because of an extra condition upon them, and some extra information is added.

Definition 10 *Let k be a positive integer. Let S be a set of n points in the plane. A k-*divided range tree*, representing the set S, consists of the following.*
There is a main tree*, which is a $BB[\alpha]$-tree, containing in its leaves the points of S, ordered according to their x-coordinates. Let v be an internal node of this main tree, representing the set S_v, and let i be the depth of v. Then node v contains an* associated structure*, which is defined as follows.*

1. *If $i = j\lceil \frac{1}{2k} \log n \rceil$ for some non-negative integer j, then the associated structure is a $BB[\alpha]$-tree, containing in its leaves the points of S_v, ordered according to their y-coordinates.*
2. *Otherwise, there is a non-negative integer j and an integer x, $1 \leq x \leq \lceil \frac{1}{2k} \log n \rceil - 1$, such that $i = j\lceil \frac{1}{2k} \log n \rceil + x$. Let u be that node in the main tree at depth $j\lceil \frac{1}{2k} \log n \rceil$, on the path towards node v. The associated structure of v is a binary tree, having the following form. The upper $(2k - j - 1)\lceil \frac{1}{2k} \log n \rceil$ levels are identical to those of the associated structure of u. Each node on level $(2k - j - 1)\lceil \frac{1}{2k} \log n \rceil$ contains a pointer to a $BB[\alpha]$-tree, containing in its leaves a subset of S_v, ordered according to their y-coordinates. For all these nodes, the sizes of these subsets of S_v are roughly equal. Also the entire associated structure of v contains the set S_v in its leaves, ordered according to their y-coordinates.*

Furthermore, each internal node of an associated structure contains

- *two mark bits which state whether the left and right subtree contain points of S;*

- *two extra pointers, one for the left, and one for the right subtree. Such an extra pointer points to the first node for which both subtrees contain points of S, or else (if no such node exists) to the only point of S in the subtree. If there are no points of S at all in the subtree, the pointer is not used.*

Next we define some concepts, which are used in the rest of this section.

Definition 11 *Consider a k-divided range tree for a set of n points.*

1. *A* tree part *is that part of a tree which starts at a node of depth $j\lceil \frac{1}{2k}\log n\rceil$, continues to a depth of $(j+1)\lceil\frac{1}{2k}\log n\rceil - 1$, and is connected. A tree part has at most $2^{\lceil \frac{1}{2k}\log n\rceil} < 2n^{\frac{1}{2k}} = O(n^{\frac{1}{2k}})$ nodes.*
2. *A* layer *is that collection of tree parts of a tree, which are located at the same depth. A perfectly balanced main tree has $2k$ layers.*
3. *A* group *is the collection of associated structures of the nodes of one tree part of the main tree.*
4. *Two (or more) tree parts of two (or more) associated structures are located at the* same position, *if the paths for reaching these tree parts are identical. In other words, when the same left-right decisions are taken in each associated structure in reaching the tree parts.*

First we shall show that k-divided range trees have the same performances as ordinary range trees.

Theorem 18 *A k-divided range tree, representing n points, can be built in time $O(n\log n)$ and takes $(n\log n)$ space to store. Using this tree, range queries can be solved in time $O((\log n)^2+t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^2)$.*

Proof. To build a k-divided range tree, consider the following algorithm:

1. Build an ordinary two-dimensional range tree.
2. For each internal node v (of the main tree) located at depth $i = j\lceil\frac{1}{2k}\log n\rceil + x$, where $j \in \{0,1,2,\ldots,2k-2\}$, and $x \in \{1,2,\ldots,\lceil\frac{1}{2k}\log n\rceil - 1\}$, do the following: copy the upper $(2k-j-1)\lceil\frac{1}{2k}\log n\rceil$ levels of the associated structure of the node having depth $j\lceil\frac{1}{2k}\log n\rceil$, which is located at the path to v. Complete this copy by traversing it and adding points, or sets of points grouped together into trees, to the lowest level of the copy, and setting the extra pointers and mark bits.

The first step takes $O(n\log n)$ time. The first part of the second step takes for each level $O(2^{(2k-j-1)\lceil\frac{1}{2k}\log n\rceil} \times 2^{j\lceil\frac{1}{2k}\log n\rceil + x}) = O(2^{x-\frac{1}{2k}\log n} \times 2^{\log n}) = O(n)$ time, since $x < \lceil\frac{1}{2k}\log n\rceil$. The second part of the second step takes for each level

$O(2^{(2k-j-1)\lceil\frac{1}{2k}\log n\rceil} \times 2^{j\lceil\frac{1}{2k}\log n\rceil+x}) + O(n) = O(n)$ time, because completing the associated structure can be done in time linear in the number of points to be added. Note that the points are ordered already in the first step. Since there are $O(\log n)$ levels, the total time for the second step is $O(n \log n)$. This shows the bound on the building time.

The size of the copied part of an associated structure is $O(2^{(2k-j-1)\lceil\frac{1}{2k}\log n\rceil})$ and the entire associated structure represents $O(2^{(2k-j)\lceil\frac{1}{2k}\log n\rceil})$ points. Consequently, the size of an associated structure does not increase in order of magnitude (i.e. it still is linear in the number of ponts represented by it), and neither will the total size of the tree.

Consider the following query algorithm with query rectangle $([x_1 : y_1], [x_2 : y_2])$:

1. Perform a range query on the main tree with $[x_1 : y_1]$, as in the ordinary case, and select the associated structures to be queried.

2. For each associated structure selected in step 1, do the following: Select subtrees to be reported with $[x_2 : y_2]$ in the same way as has been done in step 1, using the ordinary pointers. Then report these subtrees if they are not empty by traversing them, using the extra pointers.

The first step selects $O(\log n)$ associated structures and takes $O(\log n)$ time. The first part of the second step takes $O(\log n)$ time for each associated structure. This gives a total time of $O((\log n)^2)$. The second part of the second step guarantees that each internal node visited in the traversal gives an extra answer to the query. Consequently, this step takes $O(t)$ time, where t is the number of answers to the query. Hence the total query time is $O((\log n)^2 + t)$.

So we are left with the bounds on the update time. First consider the case of an update, where the balance conditions are not disturbed. Observe that the main tree and the associated structures all have a depth bounded by $O(\log n)$. Furthermore, to adjust the extra information only $O(1)$ time is needed for each node on the path to the point to be inserted or deleted. In this case the update time follows in the same way as for ordinary range trees.

Now we will show how the balance conditions can be restored efficiently if the tree gets out of balance. Whenever the main tree, a BB[α]-tree, gets out of balance, the entire tree below the highest node which is out of balance will be rebuilt (partial rebuilding).

The associated structures are also BB[α]-tree. Here rebalancing will be done by means of single and double rotations. Rotations at a level below $(2k - j - 1)\lceil\frac{1}{2k}\log n\rceil$ are done in the usual way. Above this level (the part in which associated structures of one group must be identical), restoring balance is only considered in the largest associated structure of a group. Whenever this tree needs to be rebalanced, a rotation is performed and repeated in all the other associated structures

of the group. Hence only for rotations at the higher levels the update procedure is different from that of ordinary range trees. It follows that we need only consider how much time these rotations take. When a rotation needs to be done in the higher levels, this will take $O(n^{\frac{1}{2k}} \log n)$ time, since there are $O(n^{\frac{1}{2k}})$ associated structures in a group and walking down a path and rotating takes $O(\log n)$ time. In Willard and Lueker [12], it is shown that whenever a rotation needs to be done above level $(2k-j-1)\lceil\frac{1}{2k}\log n\rceil$, and there are $(2k-j)\lceil\frac{1}{2k}\log n\rceil$ levels, then there have been $\Omega(n^{\frac{1}{2k}})$ updates since the last time this node has been rebalanced. This can happen to $2k$ associated structures and to $O(\log n)$ nodes on the path. So this gives an extra update time of $O(n^{\frac{1}{2k}} \log n) \times 2k \times O(\log n)/\Omega(n^{\frac{1}{2k}}) = O((\log n)^2)$. This proves the theorem. □

Theorem 19 *For a k-divided range tree, there exists an* $(O(n^{\frac{1}{k}}), 4k(2k+1)-4+2t, k(2k+1)+o(1))$*-partition, where t is the number of answers to the query.*

Proof. We partition the k-divided range tree in the following way. One part of the partition will contain all the tree parts, which are located at the same position, of all associated structures of one group (cf. Definition 11). The lowest layer of the associated structures, which contain the points, are divided in the same manner, except that now only $n^{\frac{1}{2k}}/\lceil\frac{1}{2k}\log n\rceil$ consecutive tree parts of the associated structures of one group are put in one part. The tree part of the main tree to which the group belongs is added to the part in which the highest situated tree parts of the associated structures of the group are put.

First we will prove that each part has size $O(n^{\frac{1}{k}})$, and since no overlaps occur, the second requirement of partitions is also met. Consider the parts with the tree parts of the highest layers. One part only contains one tree part of each associated structure of one group. Since a group has $O(n^{\frac{1}{2k}})$ associated structures and a tree part has $O(n^{\frac{1}{2k}})$ nodes, one part will have size $O(n^{\frac{1}{k}})$. The part which contains the tree part of the main tree will have an additional size of $O(n^{\frac{1}{2k}})$, but will still have size $O(n^{\frac{1}{k}})$.

Now consider the parts containing the points. The leftmost tree part with points of the largest associated structure of a group has size $O(n^{\frac{1}{2k}})$. The leftmost tree parts of the two second largest associated structures together have the same size, since they contain the same points. A tree part of the main tree has $\lceil\frac{1}{2k}\log n\rceil$ levels, so when taking together $n^{\frac{1}{2k}}/\lceil\frac{1}{2k}\log n\rceil$ consecutive tree parts with points of each associated structure of one group this part of the partition will have size $\lceil\frac{1}{2k}\log n\rceil\, O(n^{\frac{1}{2k}})\, n^{\frac{1}{2k}}/\lceil\frac{1}{2k}\log n\rceil = O(n^{\frac{1}{k}})$.

Next we will show that an update passes through at most $k(2k+1)$ parts. (We remind the reader to the update algorithm of the data structure. See the proof of Theorem 18.) A path in the main tree passes through at most $2k$ tree parts, consequently this path involves at most $2k$ groups, having $2k, 2k-1, \ldots, 1$ layers. The important observation is this: if an update passes through several tree parts

of an associated structure of a group, then that update passes through the tree parts located at the same position of all the other associated structures of that group. This is a direct consequence of the higher $(2k-j-1)\lceil \frac{1}{2k}\log n \rceil$ levels being identical. Since tree parts of a group located at the same position are stored in one part of the partition, traversal of one tree part of the main tree will involve no more parts than there are layers in any associated structure of the group. Thus there are at most $2k + (2k-1) + (2k-2) + \ldots + 1 = k(2k+1)$ parts needed for an update, if the tree has not to be rebalanced.

Now consider the case where the main tree gets out of balance. Let v be the highest node in the main tree which is out of balance after an update. Then the entire tree below this node v is rebuilt. This rebuilding involves $O(\frac{n' \log n'}{n^{1/k}})$ parts when the subtree rooted at v represents n' points (n is the current number of points represented by the entire data structure). It was shown by Lueker [5], that there must have been $\Omega(n')$ updates since the last time v was the highest node out of balance. This holds for all of the $O(\log n)$ nodes which are situated on the search path of an update. So, on the average $O(\log n) \times O(\frac{n' \log n'}{n^{1/k}})/\Omega(n') = o(1)$ extra parts are needed for an update.

Next consider an associated structure which gets out of balance. Such a structure is rebalanced by means of rotations. For single as for double rotations at most three parts of the partition are changed. Rotations at a level below $(2k-j-1)\lceil \frac{1}{2k}\log n \rceil$ are done only in the associated structure which is directly involved and requires no extra parts to be passed. Above this level (the part in which associated structures of one group must be identical), restoring balance is only considered in the largest associated structure of a group. Whenever this tree needs to be rebalanced, a rotation is performed and repeated in all the other associated structures of the group. But the rotations all use the same (at most) three parts, thus $O(1)$ extra parts are involved. Also, the adjustment of the mark bits and extra pointers can be done without using extra parts. Hence, also in this case, the number of parts involved in an update does not increase (on the average).

Finally, we will show that a range query uses at most $4k(2k+1) - 4 + 2t$ parts. A query may select $4k$ groups of associated structures, $2k$ for each path of a boundary. In each associated structure now two paths are followed, one for each boundary. In the same way as has been proven for an update, now $2(4k + (4k-2) + (4k-4) + \ldots + 4) = 4k(2k+1) - 4$ parts are needed. The subtrees to be reported contain at most t points together, thus at most t internal nodes not on the paths to the boundaries are needed. It is possible that all the points and internal nodes are situated in different parts. The number of parts through which a query passes therefore is at most $4k(2k+1) - 4 + 2t$. □

Remark. Of course, if the number of answers to a query is about n, it is not possible that $2n$ seeks are needed for such a query, because the partition contains only $O(n^{1-\frac{1}{k}} \log n)$ parts.

4 Partitions of multi-dimensional range trees

We shall now generalize the results of the preceding sections to the multi-dimensional case. First we generalize Theorem 3 (the proof is left to the reader).

Theorem 20 *For a d-dimensional range tree, there exists an*

1. $(O(n(\log n)^{d-1}), 1, 1)$*-partition.*
2. $(O(1), O((\log n)^d + t), O((\log n)^d))$*-partition, where t is the number of answers to the query.*
3. $(O(n(\log n)^{d-2}), O(\log n), O(\log n))$*-partition.*

4.1 Restricted partitions

The restricted partitions of Section 3.1 can easily be generalized to the multi-dimensional case, as we will show now. In a restricted partition of a multi-dimensional range tree, only the main tree is partitioned. Just as in the two-dimensional case, a node of the main tree and its associated structure are contained in the same part. Since the associated structure of the root of the main tree, a $(d-1)$-dimensional range tree, has size $\Theta(n(\log n)^{d-2})$, this means that in a restricted partition there is a part of size $\Omega(n(\log n)^{d-2})$.

First, we define modified d-dimensional range trees. Let $S = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$ be a set of n points in d-dimensional space, ordered according to their first coordinates. We split this set into subsets $S_1 = \{p_1, \ldots, p_{h(n)}\}, S_2 = \{p_{h(n)+1}, \ldots, p_{2h(n)}\}, \ldots$, where $h(n) = \lceil n/\log n \rceil$.

Definition 12 *A* modified d-dimensional range tree, *representing the set S, is defined as follows.*

1. *Each set S_i is stored in an ordinary d-dimensional range tree T_i. Let r_i be the root of T_i.*
2. *The roots r_i are stored in the leaves of a perfectly balanced binary tree T. Let v be a node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$. Then v contains an associated structure, which is an ordinary $(d-1)$-dimensional range tree for the set $S_i \cup S_{i+1} \cup \ldots \cup S_j$, taking only the last $d-1$ coordinates into account.*

The query and update algorithms of a modified d-dimensional range tree are similar as in the two-dimensional case. Again, we completely rebuild the structure as soon as one set S_i contains either $1/2\lceil n/\log n \rceil$ or $2\lceil n/\log n \rceil$ points. The next theorem shows that this modified range tree has asymptotically the same complexity as an ordinary range tree.

Theorem 21 *A modified d-dimensional range tree, representing n points, can be built in time $O(n(\log n)^{d-1})$, and takes $O(n(\log n)^{d-1})$ space to store. In this tree, range queries can be solved in time $O((\log n)^d+t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^d)$.*

Proof. The proof is the same as in the two-dimensional case. Note that rebuilding of the structure takes $O(n(\log n)^{d-1})$ time, and has to be done at most once every $\Omega(n/\log n)$ updates. □

It will be clear that Theorems 5 and 7 generalize to the following ones (the proofs are the same).

Theorem 22 *For a modified d-dimensional range tree, there exists an $(O(n(\log n)^{d-2}),$ $\log\log n + O(1), \log\log n + O(1))$-partition.*

Theorem 23 *For a modified d-dimensional range tree, there exists an $(O(n(\log n)^{d-2}),$ $4\log^* n + O(1), \log^* n + O(1))$-partition.*

Remark. Just as in the two-dimensional case, the partition of Theorem 23 is optimal. That is, if a d-dimensional range tree is partitioned, in the restricted sense, into parts of size $O(n(\log n)^{d-2})$, then there is an update passing through $\Omega(\log^* n)$ parts. For a proof, see Part II [10].

Next, we define k-fold modified d-dimensional range trees.

Definition 13 *Let $S = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$ be a set of n points in d-dimensional space, ordered according to their first coordinates. A* k-fold modified d-dimensional range tree *is defined as follows.*

1. *For $k = 1$, a 1-fold modified d-dimensional range tree is an ordinary range tree for the set S.*
2. *Let $k > 1, m = \lceil n\frac{(\log)^k n}{(\log)^{k-1} n}\rceil$. Partition S into sets $S_1 = \{p_1, \ldots, p_m\}, S_2 = \{p_{m+1}, \ldots, p_{2m}\}, \ldots$. Then a k-fold modified d-dimensional range tree consists of the following. Each set S_i is stored in a $(k-1)$-fold modified d-dimensional range tree T_i. Let r_i be the root of T_i. These roots are stored in the leaves of a perfectly balanced binary tree T. Let v be a node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$. Then v contains an associated structure, which is an ordinary $(d-1)$-dimensional range tree for the set $S_i \cup S_{i+1} \cup \ldots \cup S_j$, taking only the last $d-1$ coordinates into account.*

Also in this case, the query and update algorithms are similar as in Section 3.1. We rebuild the data structure as soon as one set S_i contains either $1/2\,m$ or $2\,m$ points.

Theorem 24 *A k-fold modified d-dimensional range tree, representing n points, can be built in time $O(n(\log n)^{d-1})$, and takes $O(n(\log n)^{d-1})$ space to store. In this tree, range queries can be solved in time $O((\log n)^d + t)$, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^d)$.*

Proof. The proof is the same as that of Theorem 8. □

Hence the k-fold modified d-dimensional range tree has asymptotically the same complexity as an ordinary range tree.

Theorem 25 *Let k be a positive integer. For a k-fold modified d-dimensional range tree, there exists an $(O(n(\log n)^{d-2}(\log)^k n), 2k-1, k+o(1))$-partition.*

Proof. The proof is the same as that of Theorem 9. □

Remark. Also this partition is optimal: In each restricted partition of a d-dimensional range tree, such that each update passes through at most k parts, there is a part of size $\Omega(n(\log n)^{d-2}(\log)^k n)$. See Part II [10].

4.2 Multi-dimensional reduced range trees

The reduced range tree, which we defined in Section 3.2 for the two-dimensional case, also has a multi-dimensional generalization. In this section we will describe the structure of this multi-dimensional reduced range tree, prove that it has the same performances as an ordinary range tree, and give an efficient partition into parts of size $O(n)$.

Let S be a set of n points in d-dimensional space. In order to be able to partition a d-dimensional range tree into parts of size $O(n)$, it should represent $O(n/(\log n)^{d-1})$ points. To achieve this we partition S into $(\log n)^{d-1}$ sets S_i. (This partitioning is done in the same way as in the preceding sections.) Each S_i is represented by an ordinary d-dimensional range tree T_i. We could store the roots r_i of these T_i's in the leaves of a perfectly balanced binary tree T containing no associated structures, in which case we could give an $(O(n), 3, 2+o(1))$-partition in exactly the same way as in Section 3.2. The query time, however, then becomes $O((\log n)^{d-1} \times (\log \frac{n}{(\log n)^{d-1}})^{d-1}) = O((\log n)^{2d-2})$, and this is for $d > 2$ considerably worse than $O((\log n)^d)$, the time needed for a query in an ordinary range tree. We can avoid this high query time, by storing associated structures in every $\log\log n$-th layer in T. These associated structures may sometimes be too large to be put in one part of the partition. In that case we also have to split up these structures. This proces is repeated until the trees have size $O(n)$. We will now formalize this idea.

Definition 14 *Let n, d and k be integers, $k < d$, and let $S = \{p_1, p_2, \ldots, p_m\}$ be a set of m points in d-dimensional space, ordered according to their first coordinates. A d-*dimensional *(k)-*reduced range tree*, representing the tuple $< S, n >$ is defined as follows.*

1. *A d-dimensional (-1)-reduced range tree is empty.*
2. *A d-dimensional (0)-reduced range tree is an ordinary d-dimensional range tree for the set S.*
3. *A d-dimensional (k)-reduced range tree $(k \geq 1)$ has the following structure. We partition S into subsets $S_1 = \{p_1, p_2, \ldots, p_a\}$, $S_2 = \{p_{a+1}, p_{a+2}, \ldots, p_b\}, \ldots$, such that for each set S_i, we have $|S_i| < 2\lceil m/\log n \rceil$ and $|S_i| > \frac{1}{2}\lceil m/\log n \rceil$. The structure consists of:*
   - *(a) Each set S_i is stored in a d-dimensional $(k-1)$-reduced range tree T_i.*
   - *(b) The roots r_i of these T_i's are stored in the leaves of a perfectly balanced binary tree T.*
   - *(c) Each root r_i of these T_i's contains an associated structure T_i', which is a $(d-1)$-dimensional $(k-2)$-reduced range tree, representing the tuple $< \tilde{S}_i, n >$, where $\tilde{S}_i$ is the set S_i, taking only the last $d-1$ coordinates into account.*

Definition 15 *Let S be a set of n points in d-dimensional space. A d-*dimensional reduced range tree*, representing the set S, is a d-dimensional $(d-1)$-reduced range tree, representing the tuple $< S, n >$.*

The query and update algorithms for reduced range trees are presented in the proof of the following theorem.

Theorem 26 *A d-dimensional reduced range tree, representing a set of n points, can be built in time $O(n(\log n)^{d-1})$ and takes $O(n(\log n)^{d-1})$ space to store. Using this tree, range queries can be solved in time $O((\log n)^d + t)$, t being the number of answers to be reported. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^d)$.*

Proof. The bounds on the building time and the space requirements are obvious, since a reduced range tree is just an ordinary range tree with omission of some of the associated structures. Whether or not an associated structure has to be omitted can be decided in $O(1)$ time.

An update in a d-dimensional (k)-reduced range tree B is performed as follows. If $k = -1$, we do nothing. If $k = 0$, we use the update algorithm for an ordinary range tree. If $k \geq 1$ we search in T for the T_i and T_i' we have to update. Then we perform the update in T_i and T_i' using the same algorithm recursively. If after the update T_i, which initially represents $\lceil m/\log n \rceil$ points (m being the initial number

of points represented by B), contains $< \frac{1}{2}\lceil m/\log n\rceil$ or $> 2\lceil m/\log n\rceil$ points, we completely rebuild B. In this way the update time remains $O((\log n)^d)$. The proof is analogous to that in the ordinary case. The details are left to the reader.

A query in a d-dimensional (k)-reduced range tree, with query rectangle $([x_1 : y_1], \ldots, [x_d : y_d])$ is solved as follows. If $k = -1$, nothing has to be done. If $k = 0$, we use the query algorithm for an ordinary range tree. If $k > 0$, we do the following: Search with x_1 and y_1 in T. We now find roots r_i and r_j. If $i = j$ we only have to perform a query with $([x_1 : y_1], \ldots, [x_d : y_d])$ in T_i. Otherwise, if $i < j$, we have to :

1. perform a query with $([x_1 : \infty], [x_2 : y_2], \ldots, [x_d : y_d])$ in T_i;
2. perform a query with $([-\infty : y_1], [x_2 : y_2], \ldots, [x_d : y_d])$ in T_j;
3. perform queries with $([x_2 : y_2], \ldots, [x_d : y_d])$ in the trees T_l' for all $i < l < j$.

Let $Q(d, k, n, m)$ be the worst case time needed to perform a query in a d-dimensional (k)-reduced range tree, representing the tuple $< S, n >$, where m is the number of points in S. (We do not count in $Q(d, k, n, m)$ the number of reported answers.) Let $R(d, k, n, m)$ be the worst case query time for the same tree, for a query rectangle with one of the intervals being half-infinite (as in step 1. and 2. of the above query algorithm). (Again we do not count the number of answers.) Then it follows from the above algorithm, the correctness of which can be seen easily, that the following recurrence holds:

$$\begin{aligned} Q(d,-1,n,m) &= 0, \\ Q(d,0,n,m) &= O((\log m)^d), \\ Q(d,k,n,m) &= c\log\log n \;+\; 2R(d,k-1,n,\lceil m/\log n\rceil) \\ &\quad + \log n \times Q(d-1,k-2,n,\lceil m/\log n\rceil), \end{aligned}$$

for some constant c. Here the first term on the right hand side of $Q(d, k, n, m)$ is the time to find r_i and r_j; the second term is the time for step 1. and 2.; and the third term is the time for step 3. (note that at most $\log n$ queries are involved in this third step). Since a query with a rectangle, one of its intervals being half-infinite, can be seen as a special instance of an orthogonal range query (e.g. in step 1. of the above query algorithm, we can choose y_1 sufficiently large), we have

$$R(d,k,n,m) \leq Q(d,k,n,m).$$

(Note that Q denotes the worst case query time.) Hence

$$\begin{aligned} Q(d,k,n,m) &\leq c\log\log n + 2Q(d,k-1,n,\lceil m/\log n\rceil) \\ &\quad + \log n \times Q(d-1,k-2,n,\lceil m/\log n\rceil) \\ &\leq c\log\log n + 2Q(d,k-1,n,m) + \log n \times Q(d-1,k-2,n,m) \\ &\leq c\log\log n + 2Q(d,k-1,n,m) + \log n \times Q(d-1,k-1,n,m) \\ &\leq 2Q(d,k-1,n,m) + 2\log n \times Q(d-1,k-1,n,m), \end{aligned}$$

if n is sufficiently large. Now let c_j be the constant in $Q(j,0,n,n)$, i.e., choose c_j such that $Q(j,0,n,n) \leq c_j(\log n)^j$. These constants can be chosen such that $c_j \leq c_{j+1}$. Then it can be shown by induction on d and k, using the above recurrence for $Q(d,k,n,m)$, that $Q(d,k,n,n) \leq c_d 4^k(\log n)^d$. It follows that the query time is bounded above by $Q(d,d-1,n,n) \leq c_d 4^{d-1}(\log n)^d = O((\log n)^d)$. Of course, we have to add the number of reported answers. □

Theorem 27 *For a d-dimensional reduced range tree, there exists an* $(O(n), O((d^2 - d)(\log n)^{\lfloor \frac{d-1}{2} \rfloor}), O((\frac{1}{2} + \frac{1}{2}\sqrt{5})^d))$*-partition.*

Proof. Consider the following partition:

1. A d-dimensional (-1)-reduced range tree is empty, so it need not be stored.
2. Each d-dimensional (0)-reduced range tree is stored in a separate part.
3. A d-dimensional (k)-reduced range tree $(k \geq 1)$ is partitioned as follows: Store the tree T in a special part, which is going to contain all the trees containing only roots of other trees and no associated structures. Partition the T_i's and the T_i'''s recursively.

Claim 1 *Each part in the above partition has size* $O(n)$.

Proof. It is easy to prove by induction, that a d_1-dimensional (k)-reduced range tree, representing the tuple $< S, n >$, where S is a set of n points, represents $O(n/(\log n)^{d_1-k-1})$ points. Hence such a tree needs $O(\frac{n}{(\log n)^{d_1-k-1}} \times (\log(\frac{n}{(\log n)^{d_1-k-1}}))^{d_1-1}) = O(n(\log n)^k)$ space to store. It follows that (0)-reduced range trees have size $O(n)$.

It remains to prove that our 'special part' also has size $O(n)$. Let $g(d,n)$ be the size of this part for a d-dimensional reduced range tree representing n points. Then

$$\begin{aligned} g(d,n) &\leq \log n + \log n \times (g(d-1,n) + g(d-2,n)) \text{ if } d \geq 2, \\ g(0,n) &= 0, \\ g(1,n) &= 0. \end{aligned}$$

It follows that $g(d,n) = O((\log n)^{d-1}) = O(n)$. This proves the claim. □

Claim 2 *When performing an update in a d-dimensional (k)-reduced range tree, partitioned as described above, we visit* $O((\frac{1}{2} + \frac{1}{2}\sqrt{5})^d)$ *parts on the average.*

Proof. Suppose we have to perform an update in a d-dimensional (k)-reduced range tree. The algorithm we use has been described in the proof of Theorem 26. Note that if no rebuilding has to be done, the number of seeks needed for an update in a d-dimensional (k)-reduced range tree only depends on the value of k.

Let s_k be the number of parts, and a_k the number of (0)-reduced range trees, through which an update passes, in case no rebuilding is necessary. We then have

$$\begin{aligned} s_k &= a_k + 1, \\ a_k &= a_{k-1} + a_{k-2} \text{ if } k \geq 2, \\ a_0 &= 1, \\ a_1 &= 1. \end{aligned}$$

It follows that s_{d-1}, the number of parts we visit when updating a d-dimensional reduced range tree, is $O((\frac{1}{2} + \frac{1}{2}\sqrt{5})^d)$, if no rebuilding has to be done. (Here $(\frac{1}{2} + \frac{1}{2}\sqrt{5})^d$ is an approximation to the d-th Fibonacci number.)

Now we have to charge the costs (seeks) we make when rebuilding the tree. Suppose we have to rebuild a d_1-dimensional (k)-reduced range tree B, where $k \geq 1$. Let t_k be the number of parts, and b_k the number of (0)-reduced range trees which are involved. Then

$$\begin{aligned} t_k &= b_k + 1, \\ b_k &= \log n \times (b_{k-1} + b_{k-2}) \text{ if } k \geq 2, \\ b_0 &= 1, \\ b_1 &= \log n. \end{aligned}$$

It follows that $t_k = O((\log n)^k)$. But when we have to rebuild B, there must have been $\Omega(n/(\log n)^{d_1-k-1})$ updates in B since the last time B has been rebuilt. Dividing these costs among the updates gives us $O(\frac{(\log n)^{d_1-1}}{n}) = O(\frac{(\log n)^{d-1}}{n})$ seeks per update. An update can be assigned costs from every reduced range tree on the search path for this update. This number of trees being $O((\frac{1}{2} + \frac{1}{2}\sqrt{5})^d)$, we have to charge every update for an extra $O((\frac{1}{2}+\frac{1}{2}\sqrt{5})^d \times \frac{(\log n)^{d-1}}{n}) = o(1)$ seeks for rebuilding. So the number of seeks per update is $O((\frac{1}{2} + \frac{1}{2}\sqrt{5})^d)$ on the average.
□

Claim 3 *When performing a query in a d-dimensional reduced range tree, partitioned as described above, we visit* $O((d^2 - d)(\log n)^{\lfloor \frac{d-1}{2} \rfloor})$ *parts.*

Proof. Let $S(d,k)$ be the number of (0)-reduced range trees needed for a query on a d-dimensional (k)-reduced range tree. Note that the total number of seeks needed for performing a query on a d-dimensional reduced range tree is thus $S(d,d-1)+1$. We then have the following recurrence:

$$\begin{aligned} S(d,k) &\leq 2 + 2\sum_{i=2}^{k-1} \log n \times S(d-1,i-2) + \log n \times S(d-1,k-2), \quad (*) \\ S(d,-1) &= 0, \\ S(d,0) &= 1. \end{aligned}$$

From this it can be shown that $S(d,k) = O((k^2+k)(\log n)^{\lfloor \frac{k}{2} \rfloor})$, which gives us the required bound. □

Remark: To get $(\log n)^{\lfloor \frac{d-1}{2} \rfloor}$ instead of $(\log n)^{\lceil \frac{d-1}{2} \rceil}$ in the above bound, we in fact also need $S(d,1) = 2$. This is not true for the partition given above, but it is not hard to change the partition slightly, so that $S(d,1) = 2$ holds. In the 2-dimensional case we already had $S(2,1) = 2$, and in the same way we can get $S(d,1) = 2$ for multi-dimensional reduced range trees.

Now combining the three claims proves Theorem 27. □

Remark: The number of seeks can be high, as the above theorem shows, but in practice this will seldom be the case. The number of seeks is namely strongly dependent on the number of answers in the first coordinates. When for example the number of answers in the first coordinate is $\leq n/(\log n)^{d-1}$ only two seeks are needed, and (*) is an equality only when the number of answers in the first coordinate is $\geq n - 2n/(\log n)^{d-1}$.

4.3 Multi-dimensional k-divided range trees

The d-dimensional k-divided range tree generalizes its two-dimensional counterpart. Now every structure of dimension less then d has the property that the upper levels are identical.

Definition 16 *A* d-dimensional k-divided range tree *representing a set S of n points, consists of a main tree, in which every internal node has an associated structure. The main tree is a BB[α]-tree containing in its leaves the points ordered according to their first coordinates. The associated structure of an internal node v, located at depth i ($i = 0,1,2,\ldots$) is defined as follows:*

1. *If $i = j\lceil \frac{1}{dk}\log n \rceil$ for some non-negative integer j, then the associated structure is a $(d-1)$-dimensional k-divided range tree representing the points of S below v, but divided in layers with a depth of $\lceil \frac{1}{dk}\log n \rceil$.*
2. *Otherwise, there is a non-negative integer j and an integer x, $1 \leq x \leq \lceil \frac{1}{dk}\log n \rceil - 1$, such that $i = j\lceil \frac{1}{dk}\log n \rceil + x$. Let u be that node in the main tree at depth $j\lceil \frac{1}{dk}\log n \rceil$, located on the path towards v. The associated structure of v consists of the following: The upper $(dk-j-1)\lceil \frac{1}{dk}\log n \rceil$ levels are identical to those of u. The lower levels, which contain the points in the leaves, complete the associated structure of v, as in Definition 10.*

A one-dimensional k-divided range tree does not have associated structures, but has the following extra information:

- *two mark bits which state whether the left and right subtree contain points of S;*
- *two extra pointers, one for the left, and one for the right subtree. Such an extra pointer points to the first node for which both subtrees contain points of S, or else (if no such node exists) to the only point of S in the subtree. If there are no points of S at all in the subtree, the pointer is not used.*

Theorem 28 *A d-dimensional k-divided range tree, representing n points, can be built in time $O(n(\log n)^{d-1})$, and takes $O(n(\log n)^{d-1})$ space to store. Using this tree, range queries can be solved in time $O((\log n)^d + t)$ where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^d)$.*

Proof. This can be shown in a similar way as in the two-dimensional case. □

In the same way as in Definition 11, we define the concepts of a *tree part*, a *layer*, a *group*, and the notion of tree parts to be located at the *same position*. Tree parts of d-dimensional k-divided range trees have depth $\lceil \frac{1}{dk} \log n \rceil$ and contain $O(n^{\frac{1}{dk}})$ nodes. A perfectly balanced main tree has dk layers.

To partition this range tree, one part will contain all tree parts of one-dimensional structures that are located at the same position and belong to one tree part of the main tree. The tree parts of k-dimensional structures are added to that part of the partition, in which the tree parts of $(k-1)$-dimensional structures are situated to which they give access directly in the range tree. Thus first, tree parts of one-dimensional structures are divided into parts, then tree parts of two-dimensional structures are added, and so on, until the tree parts of the main tree are divided. The maximal size of a part becomes $O(n^{\frac{1}{dk}} + (n^{\frac{1}{dk}})^2 + \ldots + (n^{\frac{1}{dk}})^d) = O(n^{\frac{1}{k}})$. The path traversed during an update uses dk tree parts of the main tree, $dk \times dk$ tree parts of $(d-1)$-dimensional structures, and finally $(dk)^{d-1} \times dk$ tree parts of the one-dimensional structures. So in total $(dk)^d$ tree parts, and hence $(dk)^d$ parts of the partition are involved in an update. This leads to the following theorem (the proof is left to the reader).

Theorem 29 *For a d-dimensional k-divided range tree, there exists an $(O(n^{1/k}), 2(2dk)^d + 2t, (dk)^d)$-partition, where t is the number of answers to the query.*

5 Storage considerations

Up to now we did not consider the amount of space used in secondary memory. It might seem that this is exactly the same amount as if the data structure were stored in core, but this is not true. When a part is changed during an update, the new part has to replace the old corresponding part in secondary memory. This

new part only fits in the old space, if its size is not larger. But sizes of parts grow when n grows. When the part does not fit into the same slot in secondary memory, we either have to find a new slot for it, or we have to split it. The first solution creates holes in the file and, hence, increases the amount of storage in secondary memory. The second solution increases the number of seeks necessary to write the part, something we clearly want to avoid.

To solve this problem, we will reserve larger slots for storing parts than is actually necessary. In this way, the slot will have enough room to store the part, even when it grows. To be more precise, consider an $(F(n), G(n), H(n))$-partition. We assume that $F(n)$ behaves smoothly in the sense that $F(O(n)) = O(F(n))$, and that $F(n)$ is non-decreasing. Now assume at some moment, at which the set represented by the data structure contains n_0 points, we rebuild the entire data structure in secondary memory. Rather than using slots of size $F(n_0)$, we use slots of size $F(2n_0)$. As a result, as long as n (the current number of points) is at most $2n_0$, parts still fit in their slots. At the moment when $n = 2n_0$, we rebuild the entire data structure in secondary memory. When n becomes very small, because of a large number of deletions, the amount of storage in secondary memory also becomes too large. To avoid this, we also rebuild the entire structure when $n \leq n_0/2$.

Theorem 30 *The data structure can be stored in secondary memory, using $O(S(n))$ storage, without increasing the average update costs in order of magnitude.*

Proof. The number of parts is $O(S(n)/F(n))$. Each part requires $F(2n_0) \leq F(4n) = O(F(n))$ storage. The storage bound follows. When the entire structure has to be rebuilt, there must have been $\Omega(n_0)$ updates. Clearly, the rebuilding of a structure of n points takes time at most the time required for n insertions. As $n = O(n_0)$, the average update time will never be increased by more than a constant factor. □

A second problem with storage might be that the physical block size is larger than the slots we need. This will occur when n is small and/or the parts are small (like in Section 3.4). In this case, we can pack a number of slots into one physical record in the usual ways for structures in secondary memory.

6 Concluding remarks

In this paper we have given a number of methods to partition range trees, such that queries and updates pass through only a small number of parts. This enables us to store range trees in secondary memory and to query and maintain them efficiently. This is very useful in case the structure does not fit in main memory, or if we want to maintain a shadow administration to be able to reconstruct the structure after

a system crash. We have shown e.g. that we can maintain a two-dimensional range tree in secondary memory, such that in each update at most $k(2k+1)$ parts of the data structure, each of size $O(n^{1/k})$, have to be transported to main memory and vice versa. Also, to perform a range query, at most $4k(2k+1)-4+2t$ parts of the structure, again each of size $O(n^{1/k})$, are involved (here t is the number of answers to the query). In Part II of this paper [10], lower bounds are given, from which it follows that several of our partitions are optimal. In particular, the restricted partitions of Sections 3.1 and 4.1 are optimal.

We have shown in Sections 3.2 and 4.2, that it is useful to change range trees, to get new data structures for the range searching problem, for which more efficient partitions exist. These new structures have the same performances as ordinary range trees. For example, it is possible to partition a variant of a two-dimensional range tree into parts of size $O(n)$, such that each update passes through 2 parts (hence 2 seeks are necessary), and each query passes through at most 3 parts (which gives at most 3 seeks to perform the query).

The techniques presented in this paper also apply to other data structures. In fact, any data structure that has the form of an augmented BB[α]-tree, with some reasonable properties of the query and update algorithms, can be partitioned in the way described in this paper. Examples of such structures are segment trees (see e.g. Preparata and Shamos [9]), structures solving set maintaining problems like maintaining a convex hull, maintaining a Voronoi diagram, etc. (see e.g. Overmars [8]), and structures for adding range restrictions to searching problems (see e.g. Bentley [2], Willard and Lueker [12]).

Acknowledgement

We would like to thank Leen Torenvliet and Peter van Emde Boas for their continuous support. Furthermore we thank Leen Torenvliet for the technical assistance during the preparation of the paper.

References

[1] R. Bayer and E.M. McCreight. *Organisation and Maintenance of Large Ordered Indexes.* Acta Informatica **1** (1972), pp. 173-189.

[2] J.L. Bentley. *Decomposable Searching Problems.* Inform. Proc. Lett. **8** (1979), pp. 244-251.

[3] N. Blum and K. Mehlhorn. *On the Average Number of Rebalancing Operations in Weight-Balanced Trees.* Theor. Comp. Sci. **11** (1980), pp. 303-320.

[4] D. Comer. *The Ubiquitous B-tree.* Computing Surveys **11** (1979), pp. 121-137.

[5] G.S. Lueker. *A Data Structure for Orthogonal Range Queries.* Proc. 19-th Annual IEEE Symp. on Foundations of Computer Science, 1978, pp. 28-34.

[6] J. Nievergelt, H. Hinterberger and K.C. Sevcik. *The Grid File: An Adaptable, Symmetric Multikey File Structure.* ACM Trans. Database Systems **9** (1) (1984), pp. 38-71.

[7] J. Nievergelt and E.M. Reingold. *Binary Search Trees of Bounded Balance.* SIAM J. Computing **2** (1973), pp. 33-43.

[8] M.H. Overmars. *The Design of Dynamic Data Structures.* Springer Lecture Notes in Computer Science, Vol. 156, Springer Verlag, 1983.

[9] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer Verlag, 1985.

[10] M.H.M. Smid and M.H. Overmars. *Maintaining Range trees in Secondary Memory, Part II: Lower Bounds.* Report FVI-87-15, University of Amsterdam, 1987.

[11] M.H.M. Smid, L. Torenvliet, P. van Emde Boas and M.H. Overmars. *Two Models for the Reconstruction Problem for Dynamic Data Structures.* Report FVI-87-13, University of Amsterdam, 1987.

[12] D.E. Willard and G.S. Lueker. *Adding Range Restriction Capability to Dynamic Data Structures.* Journal of the ACM **32** (1985), pp. 597-617.