MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Range trees with slack parameter

Michiel Smid

MPI–I–91–102 April 1991

Im Stadtwald
W 6600 Saarbrücken
Germany

Range trees with slack parameter*

Michiel Smid
Max-Planck-Institut für Informatik
D-6600 Saarbrücken, Germany

April 5, 1991

Abstract

Range trees with slack parameter were introduced by Mehlhorn as a dynamic data structure for solving the orthogonal range searching problem. By varying the slack parameter, this structure gives many trade-offs for the complexity measures. In this note, a complete analysis is given for this data structure.

1 Introduction

In the *orthogonal range searching problem*, we are given a set S of n points in d-space. We have to store these points in a data structure such that *range queries* can be solved efficiently. Such a query consists of an axis-parallel hyper-rectangle $[a_1 : b_1] \times \ldots \times [a_d : b_d]$, and we have to report all points of S that lie in this rectangle, i.e., all points $p = (p_1, \ldots, p_d)$ in S that satisfy $a_1 \leq p_1 \leq b_1, \ldots, a_d \leq p_d \leq b_d$.

A well-known data structure for this problem is the *range tree*, introduced by Bentley [1] and Lueker [3]. See also Willard and Lueker [7].

Mehlhorn [4, page 44] gives a variation of this data structure, the *range tree with slack parameter*. The algorithms and analysis given there are, however, rather compact. Moreover, the analysis of the update algorithm is wrong. In this note, we give a complete analysis for this data structure.

Let T be a binary search tree. For each node v, let n_v denote the number of leaves in the subtree of v. This tree is called a BB[α]-tree, if $\alpha \leq n_w/n_v \leq 1 - \alpha$ for all nodes v and w such that v is the father of w. Here, α is a constant such that $0 < \alpha \leq 1/3$.

We use BB[α]-trees as leaf-search trees. That is, elements are stored in sorted order in the leaves. Internal nodes, i.e., non-leaf nodes, contain information to guide searches.

*This work was supported by the ESPRIT II Basic Research Actions Program, under contract No. 3075 (project ALCOM).

We assume that no two points in the set S have equal coordinates in any dimension. Later, we say something about degenerate cases.

Definition 1 Let S be a set of n points in d-space, and let m be a positive integer. A d-dimensional range tree with slack parameter m for the set S is defined as follows.

If $d = 1$, then it is a BB[α]-tree storing the elements of S in sorted order in its leaves. These leaves are linked by pointers, from left to right.

Let $d > 1$. Then the data structure consists of the following:

- A *main tree*, which is a BB[α]-tree storing the points of S in its leaves, sorted by their first coordinates.
- Each internal node in this main tree having a depth that is divisible by m, contains (a pointer to) an *associated structure*. Let v be such a node and let S_v be the points of S that are stored in the subtree of v. Then the associated structure of v is a $(d-1)$-dimensional range tree with slack parameter m for the set S'_v, which is obtained from S_v by deleting in each point the first coordinate.

2 Solving orthogonal range queries

Let $[a_1 : b_1] \times \ldots \times [a_d : b_d]$ be a query rectangle. The algorithm that finds all points of S that are in this rectangle works as follows.

If $d = 1$, then we search in the BB[α]-tree for the smallest value that is at least equal to a_1. Let w be the leaf in which this search ends. (If there is no value at least equal to a_1, the search ends in the rightmost leaf.) Starting in w, we walk to the right, following the pointers that connect the leaves, until we reach a value that is larger than b_1. All values encountered during this walk lie in $[a_1 : b_1]$ and are reported.

If $d > 1$, we do the following:

1. We search in the main tree for the leaf that contains the smallest (resp. largest) value that is at least (resp. at most) equal to a_1 (resp. b_1). Let w_a and w_b be the leaves in which the two searches end.

2. If w_b is to the left of w_a, the algorithm is finished and there are no answers.

3. If $w_a = w_b$, we check whether the point that is stored in this leaf lies in the query rectangle. If it does, we report it.

4. Otherwise, let u be the node at which the search for a_1 proceeds to the left son and the search for b_1 proceeds to the right son. We compute a set M of nodes in the main tree, as follows.

   Consider the path in the subtree of u that leads to a_1. All nodes that are right son of a node v on this path, where $v \neq u$ and in which the path proceeds to the left son of v, are put in M. Similarly, all nodes that are left son of a node w, where $w \neq u$ is on the right path in the subtree of u, and in which the

search proceeds to the right son of w, are put in M. Finally, we put the two leaves w_a and w_b in M.

For each node v in M, we have to find all points in its subtree whose second to last coordinates are contained in $[a_2 : b_2] \times \ldots \times [a_d : b_d]$. Note that in general, there is no associated structure in v. Therefore, we call the following procedure:

Find(v):
if v is a leaf or v contains an associated structure
then report v and stop
else let v_l and v_r be the left and right son of v;
    Find(v_l); **Find**(v_r)
fi
endFind

Let $v_1, \ldots, v_t$ be the sequence of nodes computed by this procedure. For $i = 1, \ldots, t$, if v_i is a leaf, we report the point in v_i if it lies in the query rectangle. Otherwise, if v_i is not a leaf, we do a range query in the associated structure of v_i with query rectangle $[a_2 : b_2] \times \ldots \times [a_d : b_d]$.

This concludes the query algorithm. We now prove the correctness of this algorithm and analyze its running time. Let $d > 1$. It is clear that the algorithm is correct if case 2 or 3 applies. Therefore, it remains to analyze case 4.

Claim 1 *For v a node in the main tree, let S_v be the points of S that are stored in the subtree of v. Then,*

$$\{p \in S : a_1 \leq p_1 \leq b_1\} = \bigcup_{v \in M} S_v,$$

where the right-hand side is a union of pairwise disjoint sets.

Proof: It is clear that the set on the right-hand side is a subset of that on the left-hand side. Let p be a point in the set on the left-hand side. Let w be the leaf that contains p. Consider the path in the main tree that leads to w. This path goes through node u. Assume w.l.o.g. that w is in the left subtree of u. If $w = w_a$, then $p \in S_w$ and $w \in M$ and, hence, p is in the set on the right-hand side. Otherwise, w is to the right of w_a. But then, there must be a node v, in which the path to w_a proceeds to the left son and the path to w proceeds to the right son. Then, $p \in S_v$ and $v \in M$. Thus, p is in the set on the right-hand side. This proves that both sets are equal.

Let p be a point in S and suppose that there are nodes v and v' in M such that $p \in S_v$ and $p \in S_{v'}$. Let w be the leaf that contains p. Then there are two different paths from w to the root: One path through v, the other through v'. This is a contradiction. Hence, the sets on the right-hand side are pairwise disjoint. ■

Claim 2 *Let v be a node in M, and let $v_1, \ldots, v_t$ be the nodes that are computed by the procedure* **Find**(v). *Then each node v_i is either a leaf or it contains an associated structure. Moreover,*

$$S_v = \bigcup_{i=1}^{t} S_{v_i},$$

where the right-hand side is a union of pairwise disjoint sets.

Proof: The proof is similar as that of Claim 1. ■

Claim 3 *At each level of the main tree, at most two nodes are contained in the set $M \setminus \{w_a, w_b\}$.*

Proof: The proof is left to the reader. ■

Lemma 1 *The query algorithm is correct and has a running time that is bounded by $O((2^m/m)^{d-1}(\log n)^d + k)$, where k is the number of reported answers.*

Proof: It is clear that the query algorithm is correct for $d = 1$. The correctness for the higher-dimensional case follows by induction on d, using Claims 1 and 2.

It is clear that for $d = 1$, the algorithm takes $O(\log n + k)$ time. Let $d > 1$. If case 2 or 3 applies, the algorithm takes $O(\log n)$ time and $k \leq 2$.

So assume that case 4 applies. Let $Q(n, d)$ denote the query time, where we do not count the time needed to report the answers. The set M can be computed in $O(\log n)$ time. Let N be the set of nodes that are computed by calls to the procedure **Find**(v), where v ranges over M. The set N is computed in $O(|N|)$ time. If $w \in N$ is a leaf, we check the point that is stored in w. If $w \in N$ is not a leaf, we do a $(d-1)$-dimensional range query in its associated structure. For $w \in N$, let k_w be the number of points that are reported by w. Then the total query time in the d-dimensional range tree is bounded by

$$O\left(\log n + |N| + \sum_{w \in N : w \neq \text{leaf}} (Q(|S_w|, d-1) + k_w)\right).$$

Using Claims 1 and 2, it can be shown by induction on d that each point of S that is contained in the query rectangle is reported exactly once. Furthermore, note that $|S_w| \leq n$. Therefore, the total query time is bounded by $O(\log n + |N|\, Q(n, d-1) + k)$, if k is the total number of reported points.

In the rest of the proof we will show that $|N| = O(2^m(\log n)/m)$. Using this upper bound, the query time follows by induction on d.

For each node w in N, there is a node v in M such that w is computed by **Find**(v). How many nodes are computed by the procedure **Find**(v) for a fixed node v in M? Let $l \geq 1$ and $1 \leq i \leq m$ be such that v is at level $(l-1)m + i$. Then the procedure **Find**(v) delivers at most 2^{m-i} nodes to N.

Now let $l \geq 1$ be an integer, and consider all nodes in $M \setminus \{w_a, w_b\}$ that are at one of the levels $(l-1)m + 1, (l-1)m + 2, \ldots, lm$. By Claim 3, each of these levels

contains at most two nodes of $M \setminus \{w_a, w_b\}$. It follows that these nodes deliver at most

$$2(2^{m-1} + 2^{m-2} + \ldots + 1) \leq 2^{m+1}$$

nodes to N.

Since there are $O((\log n)/m)$ possible values for l, and since the leaves w_a and w_b deliver two nodes to N, we have shown that $|N| = O(2 + 2^{m+1}(\log n)/m) = O(2^m(\log n)/m)$. This completes the proof. ∎

3 Building the data structure

We give an algorithm that builds a perfectly balanced range tree with slack parameter m.

The definition gives a straightforward recursive algorithm. Let $P(n, d)$ denote the running time of this algorithm. Then $P(n, 1) = O(n \log n)$. If $d > 1$ and $l \geq 0$ is an integer such that $l = O((\log n)/m)$, then there are at most 2^{lm} associated structures at level lm. Each such associated structure is a $(d-1)$-dimensional range tree for $O(n/2^{lm})$ points. The main tree can be built in $O(n \log n)$ time. Therefore,

$$P(n, d) = O(n \log n) + \sum_{l=0}^{O((\log n)/m)} \sum_{i=1}^{2^{lm}} P(n/2^{lm}, d-1).$$

This recurrence solves to

$$P(n, d) = O\left(n \log n + n \frac{(\log n)^d}{m^{d-1}}\right).$$

We show how *presorting* can be used to improve the building time. The following algorithm builds a range tree with slack parameter m. This range tree contains more information than that of Definition 1. The extra information will be used in the update algorithm.

The building algorithm: Let S be a set of n points in d-space, where $d \geq 1$.

1. In the presorting step, we do the following. For $i = 1, \ldots, d$, we sort the points of S by their i-th coordinates, and store them in a list S^i. For each $1 < i \leq d$, we give each point in S^i a pointer to its copy in S^{i-1}.

2. If $d = 1$, we store the elements of S^1 in the leaves of a perfectly balanced binary search tree and link the leaves by pointers. Then, the algorithm is finished.

3. Assume that $d > 1$.

   (a) We build the main tree. That is, we store the points of S^1 in the leaves of a perfectly balanced binary search tree. In each node v, we store values $min(v)$ and $max(v)$, which are the minimal and maximal first coordinates of the points that are stored in its subtree.

(b) For $l = 0, 1, 2, \ldots$, we have to build associated structures for the internal nodes at level lm. Let $v_1, v_2, \ldots$ be the internal nodes at level lm, ordered from left to right. Each node v_i has an interval $[min_i : max_i]$ associated with it, where $min_i = min(v_i)$ and $max_i = max(v_i)$.

In each node v_i, we initialize empty lists $S_i^1, \ldots, S_i^d$.

We simultaneously walk along the sorted list S^1 and the sorted sequence of nodes v_i. During this walk, we give each point p a pointer to node v_i—where i is such that the first coordinate of p lies in $[min_i : max_i]$—we store p at the end of the list S_i^1, and give p in S^1 a pointer to its copy in S_i^1.

Then we do the following for $j = 2, \ldots, d$: We walk along the list S^j. For each point p in this list, we follow the pointer to the position of p in S^{j-1}. Then we follow the pointer from S^{j-1} to the node v_i in whose associated structure point p belongs. We store p at the end of the list S_i^j, give p in S^j a pointer to node v_i and a pointer to its copy in S_i^j, and give p in S_i^j a pointer to its copy in S_i^{j-1}. (This copy in S_i^{j-1} can be found by following the pointer from p in S^{j-1} to p in S_i^{j-1}.)

Afterwards, we have at each node v_i a sequence of lists S_i^j, $j = 1, 2, \ldots, d$. The j-th list contains the points of S_{v_i} sorted by their j-th coordinates. Moreover, there are pointers from each point in S_i^j to its copy in S_i^{j-1}, $j = 2, \ldots, d$.

We build for each non-leaf v_i a $(d-1)$-dimensional range tree for the points in S_{v_i}, using the same algorithm recursively. Note that the presorting step is not necessary in recursive calls.

Consider a node v_i. If we are in v_i, we have access to d binary trees, where the j-th one contains the points of S_{v_i} sorted by their j-th coordinates. (For $j = 0$, this is the subtree of the main tree rooted at v_i, for $j = 1$, it is the main tree of the associated structure of v_i, for $j = 2$, it is the main tree of the associated structure that is stored with the root of the associated structure of v_i, etc.) We take care that each point in the j-th tree gets a pointer to its copy in the $(j-1)$-th tree, $j = 2, \ldots, d$. This can be done during the algorithm.

Lemma 2 *The above algorithm builds a d-dimensional range tree with slack parameter m in $O(n \log n + n((\log n)/m)^{d-1})$ time.*

Proof: The presorting step takes $O(n \log n)$ time. Let $P(n, d)$ denote the time needed to build the data structure provided we have the sorted lists $S^1, \ldots, S^d$, such that each $p \in S^j$ contains a pointer to its copy in S^{j-1}, $j = 2, \ldots, d$.

Then, $P(n, 1) = O(n)$, because a perfectly balanced tree for a sorted list of elements can be built in linear time.

Let $d > 1$. In step 3a, we build the main tree, which takes linear time. Consider

step 3b. For fixed l, this step takes an amount of time that is bounded by

$$O\left(n + \sum_{i=1}^{2^{lm}} P(n_i^l, d-1)\right),$$

where n_i^l is the number of points that are stored in the subtree rooted at the i-th node of level l. The total time for step 3b follows by summing this expression over the values $l = 0, 1, \ldots, O((\log n)/m)$. Hence,

$$P(n,d) = O\left(n\frac{\log n}{m} + \sum_{l=0}^{O((\log n)/m)} \sum_{i=1}^{2^{lm}} P(n_i^l, d-1)\right).$$

Note that $\sum_i n_i^l \leq n$. Using induction on d, it can easily be shown that $P(n,d) = O(n\,((\log n)/m)^{d-1})$. ∎

Lemma 3 *A d-dimensional range tree with slack parameter m for a set of n points has size $O(n\,((\log n)/m)^{d-1})$.*

Proof: The size $S(n,d)$ satisfies the same recurrence relation as $P(n,d)$ in the previous proof. ∎

4 Inserting and deleting points

Suppose we want to insert or delete point $p = (p_1, \ldots, p_d)$. Then we search in the main tree with p_1 for the position where p has to be inserted or deleted. If $d > 1$, we insert or delete the point $(p_2, \ldots, p_d)$ in the associated structures we encounter at the levels $0, m, 2m, \ldots$, using the same algorithm recursively.

Assume that we have to insert p. Let w be the leaf in which the search ends and let q be the point that is stored in w. Then we give w two sons, one storing p and the other storing q. (The point with the smallest first coordinate is stored in the left son.) If $d > 1$ and w has a depth that is divisible by m, we build an associated structure for the set $\{p', q'\}$ which contains the points p and q without their first coordinates, and store it in w. In node w, we have access to d binary trees storing the sets S_w^j, $j = 1, \ldots, d$, sorted by their j-th coordinates. We store pointers from p and q in the tree for S_w^j to their copies in the tree for S_w^{j-1}, $j = 2, \ldots, d$.

A deletion can be handled similarly.

Afterwards, we have to rebalance the main tree. We use the *partial rebuilding technique.* That is, we walk back to the root of the main tree and find the highest node v that does not satisfy the BB[α]-condition. Then we *rebalance* at node v: We rebuild the subtree rooted at v as a perfectly balanced range tree with slack parameter m. Note that in this subtree, we store associated structures at levels $lm - d_v, lm - d_v + m, lm - d_v + 2m, \ldots$, where d_v is the depth of v in the (entire) main tree and $l = \lceil d_v/m \rceil$.

We bound the time for a rebalancing operation at node v. Let n_v be the number of points that are stored in the subtree of v. By Lemma 2, this rebalancing operation can be done in $O(n_v \log n_v + n_v((\log n_v)/m)^{d-1})$ time. The first term is the time for the presorting step. Note that in the highest associated structures in the subtree of v, subsets are stored that are ordered already. The next claim shows that we can use this information to rebuild the subtree of v faster.

Claim 4 *A rebalancing operation at node v having n_v points in its subtree can be done in $O(n_v)$ time if $d = 1$, and in $O(n_v m + n_v((\log n_v)/m)^{d-1})$ time if $d > 1$.*

Proof: Let S_v be the set of points that are stored in the subtree of v. The claim is clear for $d = 1$, because the points of S_v are stored in sorted order in the subtree of v.

Let $d > 1$. We saw already that if we have lists S_v^j, $1 \leq j \leq d$, containing the points of S_v sorted by their j-th coordinates, and if we have pointers from each $p \in S_v^j$ to its copy in S_v^{j-1} for $j = 2, \ldots, d$, then the range subtree can be built in $O(n_v((\log n_v)/m)^{d-1})$ time. (In this subtree, associated structures are stored at levels that differ by a multiple of m. The highest of these levels is not necessarily the level of the root of the subtree. This does not affect the running time of the building algorithm.)

Therefore, we only have to show how we can obtain these sorted lists and the pointers between them in $O(n_v m)$ time.

Call the procedure **Find**(v) that was defined in the query algorithm. This gives a sequence of nodes $v_1, \ldots, v_t$, ordered from left to right, that are either leaves or that contain an associated structure. By Claim 2, $S_v = \bigcup_{i=1}^{t} S_{v_i}$, where the sets on the right-hand side are pairwise disjoint. Note that $t \leq n_v$ and $t \leq 2^m$. Hence, these nodes are found in $O(t) = O(n_v)$ time.

Each non-leaf v_i is root of a subtree of the main tree. This subtree stores the points of S_{v_i} sorted by their first coordinates. Moreover, v_i contains a $(d-1)$-dimensional range tree for the set S_{v_i}. If we are in v_i, then we can obtain the sequences $S_{v_i}^j$, $j = 1, \ldots, d$, where each $S_{v_i}^j$ contains the points of S_{v_i} sorted by their j-th coordinates, and each point in $S_{v_i}^j$ contains a pointer to its copy in $S_{v_i}^{j-1}$ for $j = 2, \ldots, d$. This can be done in $O(\sum_i |S_{v_i}|) = O(n_v)$ time.

We give each leaf v_i a sequence of lists $S_{v_i}^j$, $j = 1, \ldots, d$, storing the only point of S_{v_i}. We link these lists by pointers. This can be done in $O(t) = O(n_v)$ time.

Finally, for $j = 1, \ldots, d$, we merge the lists $S_{v_1}^j, \ldots, S_{v_t}^j$ into a sorted list S_v^j. We take care that each $p \in S_v^j$ gets a pointer to its copy in S_v^{j-1}, $j = 2, \ldots, d$. This can be done in $O(n_v \log t) = O(n_v m)$ time.

We have shown that we can obtain the sorted lists and the pointers between them in $O(n_v + n_v m) = O(n_v m)$ time. This completes the proof. ■

Next, we show that expensive rebalancing operations do not occur too often.

Claim 5 *Let v be a node in a BB[α]-tree that is in perfect balance. Let n_v be the number of leaves in the subtree of v at the moment it gets out of balance. Then there have been at least $(1 - 2\alpha)n_v - 2$ updates in the subtree of v.*

Proof: The proof given here is due to Overmars [6]. Let n'_v, n'_{lv} and n'_{rv} be the number of leaves in the subtree of v, the left son of v and the right son of v, respectively, at the moment that v is in perfect balance. Assume w.l.o.g. that $n'_{lv} \leq n'_{rv}$. Clearly, the fastest way for node v to get out of balance, is by deleting objects from its left subtree, and by inserting objects into its right subtree. Suppose that at the moment when v gets out of balance, N_i insertions have taken place in the right subtree of v, and N_d deletions in the left subtree of v. Let n_v (resp. n_{lv}) be the number of leaves in the subtree of v (resp. the left son of v) at the moment v gets out of balance. Then $n_v = n'_v + N_i - N_d$ and $n_{lv} = n'_{lv} - N_d = \lfloor n'_v/2 \rfloor - N_d$. Since node v is out of balance at this moment, we have $n_{lv}/n_v < \alpha$. It follows that

$$\alpha n_v > n_{lv} = \lfloor \frac{n'_v}{2} \rfloor - N_d > \frac{n'_v}{2} - 1 - N_d = \frac{n_v - N_i + N_d}{2} - 1 - N_d = \frac{n_v - (N_i + N_d)}{2} - 1.$$

Thus, $N_i + N_d > (1 - 2\alpha)n_v - 2$, i.e., there have been at least $(1 - 2\alpha)n_v - 2$ updates in the subtree of v. ∎

Now we are ready to bound the amortized update time. Note that the slack parameter m only makes sense if $m = O(\log n)$.

Lemma 4 *In a d-dimensional range tree with slack parameter m such that $m = O(\log n)$, points can be inserted and deleted in $O(\log n)$ amortized time if $d = 1$, and $O((\log n)^d/m^{d-1} + (\log n)^{d-1}/m^{d-3})$ amortized time if $d > 1$.*

Proof: First we bound the amortized costs for rebalancing operations that are caused by nodes of the main tree. Consider a node v that is on the search path in the main tree. By Claims 4 and 5, this node contributes an amount of $O(1)$ to the amortized rebalancing costs if $d = 1$, and $O(m + ((\log n_v)/m)^{d-1}) = O(m + ((\log n)/m)^{d-1})$ if $d > 1$. Since there are $O(\log n)$ such nodes v, it follows that the amortized rebalancing time for one single update—caused by nodes of the main tree—is bounded by $O(\log n)$ if $d = 1$, and $O(m \log n + (\log n)^d/m^{d-1})$ if $d > 1$.

Let $U(n, d)$ denote the amortized update time. Then $U(n, 1) = O(\log n)$, because the update algorithm takes $O(\log n)$ time before the rebalancing step, and we just saw that the amortized time for rebalancing is also bounded by $O(\log n)$. For $d > 1$, we have

$$U(n, d) = O(\log n + U(n, d - 1)(\log n)/m) + O(m \log n + (\log n)^d/m^{d-1}). \quad (1)$$

We claim that

$$U(n, d) = O((\log n)^d/m^{d-1} + (\log n)^{d-1}/m^{d-3}) \text{ if } d \geq 2. \quad (2)$$

Equation (2) holds for $d = 2$, because (1) implies that

$$U(n, 2) = O(\log n + (\log n)^2/m + m \log n + (\log n)^2/m) = O((\log n)^2/m + (\log n)/m^{-1}).$$

The proof for $d > 2$ can easily be given by induction, using (1) and the fact that $m = O(\log n)$. ∎

5 The final result

Before we give the final result, we say something about degenerate point sets, i.e., sets where two or more points have the same coordinates in some dimension. If such a set is stored in the leaves of a binary search tree sorted by, say, their j-th coordinates, then points with equal j-th coordinates are stored in lexicographical order. The search information that is stored in an internal node consists of a d-dimensional point instead of just a j-th coordinate.

The algorithms are only slightly changed. For example, in a query with first interval $[a_1 : b_1]$, we search for the leftmost leaf that contains a point whose first coordinate is at least equal to a_1. It should be clear that all claims and lemmas still hold.

As a second remark, until now we assumed that the slack parameter m is independent of n. What happens if m varies with n? Let $f(n)$ be an integer function, which will play the role of the slack parameter. Clearly, it only makes sense to consider functions f such that $f(n) = O(\log n)$. We assume that f is non-decreasing and smooth, i.e., $f(\Theta(n)) = \Theta(f(n))$. Let n_0 be the number of points at the start of the algorithm. Then we build the range tree with slack parameter $f(n_0)$. After $n_0/2$ updates we rebuild the entire data structure, taking the slack parameter equal to $f(n_1)$, where n_1 is the number of points at that moment.

Between two rebuilding operations, the current number n of points always satisfies $n_0/2 \leq n \leq 3n_0/2$. Hence, $f(2n/3) \leq f(n_0) \leq f(2n)$.

Since the entire data structure is rebuilt only once every $\Theta(n)$ updates, it follows from Lemma 2—or Claim 4—that we have to add $O(\log n + ((\log n)/f(n))^{d-1})$ to the amortized update time. This amount is bounded by the overall amortized update time bound of Lemma 4. Therefore, Lemma 4 remains valid.

In the next theorem, we state our final result. The proof follows from the previous sections and from the above remarks.

Theorem 1 *Let S be a set of n points in d-space, where $d \geq 2$, and let $f(n)$ be a smooth non-decreasing integer function such that $f(n) = O(\log n)$. A range tree with slack parameter $f(n)$ for the set S*

1. *has size $O(n\,((\log n)/f(n))^{d-1})$,*
2. *can be built in $O(n \log n + n\,((\log n)/f(n))^{d-1})$ time,*
3. *has a query time of $O((2^{f(2n)}/f(n))^{d-1}(\log n)^d + k)$, where k is the number of reported points,*
4. *has an amortized update time of $O((\log n)^d/f(n)^{d-1} + (\log n)^{d-1}/f(n)^{d-3})$.*

By choosing different functions f, we get many interesting trade-offs. For example, for $f(n) = 1$, we get the well-known result of [3,7]. (Note that for this case the query and update times can be improved by using dynamic fractional cascading, see Mehlhorn and Näher [5].)

If $f(n) = \lceil \epsilon(\log n)/(d-1) \rceil$, we get a data structure of linear size, in which queries can be solved in $O(n^\epsilon \log n + k)$ time, and in which points can be inserted and deleted in $O((\log n)^2)$ amortized time.

Particularly interesting is the case $f(n) = \lceil \epsilon(\log \log n)/(d-1) \rceil$. For this function, the range tree has a query time of $O((\log n)^{d+\epsilon}/(\log \log n)^{d-1} + k)$, an amortized update time of $O((\log n)^d/(\log \log n)^{d-1})$ and a size of $O(n\,((\log n)/\log \log n)^{d-1})$.

Chazelle [2] has shown that any data structure that solves the range searching problem on a pointer machine with $O(\mathit{polylog}(n) + k)$ query time, must have a size of $\Omega(n\,((\log n)/\log \log n)^{d-1})$. Note that our range tree can be implemented on a pointer machine. It follows that it is optimal for the latter choice of f.

References

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

[2] B. Chazelle. *Lower bounds for orthogonal range searching: I. The reporting case.* J. ACM **37** (1990), pp. 200-212.

[3] 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.

[4] K. Mehlhorn. *Data Structures and Algorithms, Volume 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, 1984.

[5] K. Mehlhorn and S. Näher. *Dynamic fractional cascading.* Algorithmica **5** (1990), pp. 215-241.

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

[7] D.E. Willard and G.S. Lueker. *Adding range restriction capability to dynamic data structures.* J. ACM **32** (1985), pp. 597-617.