SEARCHING IN THE PAST II:
GENERAL TRANSFORMS

Mark H. Overmars

RUU-CS-81-9

May 1981

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

SEARCHING IN THE PAST II: GENERAL TRANSFORMS

Mark H. Overmars

Technical Report RUU-CS-81-9

May 1981

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

Proposed running head:

SEARCHING IN THE PAST II

All correspondence to:

Mark H. Overmars
Dept. of Computer Science
University of Utrecht
P.O. Box 80.002
3508 TA Utrecht
the Netherlands

SEARCHING IN THE PAST II: GENERAL TRANSFORMS*

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

Abstract. Data structures normally are unable to remember the situation they held at moments in the past. We will give general methods to transform data structures into structures that remember their history, such that queries can be answered over the situation at any moment in the past. A first transformation is applicable to all searching problems for which dynamic data structures exist, although the resulting structures are not very efficient. Next, we concentrate on decomposable searching problems. We show that static data structures for decomposable searching problems can be transformed into efficient data structures for the in-the-past version of the problem, when only insertions occur. When the known data structure is dynamic, also deletions can be handled by the structure for the in-the-past problems.

Keywords and phrases: In-the-past searching, decomposable searching problem, C(n)-decomposability.

1. Introduction

A large class of problems that received considerable attention during the last few years is the class of so-called searching problems. A SEARCHING PROBLEM can be viewed as a problem in which one asks a question (often called a query) about an arbitrary object x (called the query object) with respect to a set of objects V. The most well-known example of a searching problem is the MEMBER SEARCHING problem that asks whether $x \in V$. Many searching problems come from such areas as database design and computational geometry (see e.g. Shamos [16]). Two prime examples are (i) the NEAREST NEIGHBOR SEARCHING problem that asks for the point in a multidimensional

---

* This work was supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

pointset that lies nearest to a given query point and (ii) the RANGE SEARCHING problem that asks for those elements (or their number) of a multidimensional pointset that lie in a given rectilinearly oriented range. Some searching problems do not really have a query object x. In a number of problems we just ask a question about V itself. These searching problems are called SET PROBLEMS. Examples of set problems are (i) the MAXIMUM problem that asks for the largest element of V and (ii) the CONVEX HULL problem that asks for the convex hull of a 2-(or higher) dimensional pointset.

"Solving" a searching problem Q consists of developing some data structure S for Q to represent the set V such that queries with different search objects can be answered efficiently. Such a data structure S can be static or dynamic. A STATIC structure is a structure that is once built for the set V and on which we only perform queries. For set problems a "static data structure" normally consists of the answer to the problem itself. A DYNAMIC data structure permits us to perform updates (insertions and/or deletions of elements in the set) efficiently as well. Developing a dynamic data structure for a searching problem Q is often harder than giving a static structure. We would like to be able to transform a known static data structure for Q into a dynamic structure for Q by means of some uniform method. The process of transforming a static structure into a dynamic structure is called DYNAMIZATION. During the last three years, several efforts were made to develop general methods for turning static solutions to searching problems into (efficient) dynamic solutions. Bentley [1] made the important observation that such a general approach to dynamization is especially relevant to the class of so-called decomposable searching problems.

<u>Definition</u> 1.1. A searching problem Q is called DECOMPOSABLE if and only if for any partition $A \cup B = V$ and any query object x, $Q(x,V) = \square(Q(x,A), Q(x,B))$, where $\square$ takes $O(1)$ to compute.

For example, the Member searching problem is decomposable with $\square = \vee$(<u>or</u>), the Nearest neighbor searching problem is decomposable with $\square$ = minimal distance, and the Range searching problem is decomposable with $\square = \cup$(union) or $\square = +$, where we choose the latter when we are only interested in the number of elements that lie in the range.

<u>Notation</u>. Let S be a structure for a searching problem, containing n points.

$Q_S(n)$ = the time required to perform a query on S,

$P_S(n)$ = the time required to build S,

$I_S(n)$ = the time required to perform an insertion on S (when applicable),
$D_S(n)$ = the time required to perform a deletion on S (when applicable),
$S_S(n)$ = the storage required for S.

We normally assume that all bounds are worst-case bounds. When averages are meant we add a superscript a. We assume that Q, I and D are non-decreasing, that P and S are at least linear and that all functions are smooth. (A function F is called smooth if $F(O(n)) = O(F(n))$.)

Bentley [1] gave a first method for transforming static data structures for decomposable searching problems into structures that permit us to perform insertions in good average time bounds, while the query time remains reasonable.

<u>Theorem</u> 1.2. (Bentley [1]) Let S be a static data structure for a decomposable searching problem Q. There exists a structure S' for Q such that

$$Q_{S'}(n) = \begin{cases} O(Q_S(n)) & \text{if } Q_S(n) = \Omega(n^{\varepsilon}),\ \varepsilon > 0 \\ O(\log n)Q_S(n) & \text{otherwise.} \end{cases}$$

$$I^a_{S'}(n) = \begin{cases} O(P_S(n)/n) & \text{if } P_S(n) = \Omega(n^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log n)P_S(n)/n & \text{otherwise.} \end{cases}$$

$$S_{S'}(n) = O(S_S(n)$$

Soon after, a number of other authors extended the result of Bentley [1] by giving more general dynamization methods, yielding different trade-offs between query and insertion time, by developing general deletion techniques, by changing average bounds into worst-case bounds and by proving lower bounds on the efficiency of the transforms. The most general - and up to constants, optimal - method was recently given by Overmars and van Leeuwen [14], gathering all insight gained so-far. (See also van Leeuwen and Overmars [18])

<u>Theorem</u> 1.3. (Overmars and van Leeuwen [14]) Let S be a static data structure for a decomposable searching problem Q and let f and g be two smooth, nondecreasing functions with $1 \le f(n), g(n) \le n$. There exists a dynamic structure S' for Q, such that

$$Q_{S'}(n) = O(f(n) + g(n))Q_S(n/g(n))$$

$$D_{S'}(n) = O(D_S(n/g(n)) + \log n + P_S(n)/n)$$

$$I_{S'}(n) = \begin{cases} O(\log n/\log \frac{f(n)}{\log n})P_S(n)/n & \text{if } f(n) = \Omega(\log n) \\ O(f(n).n^{\frac{1}{f(n)}}).P_S(n)/n & \text{if } f(n) = O(\log n) \end{cases}$$

$$S_{S'}(n) = O(S_S(n))$$

For static structures S, D_S normally will be P_S, but for a number of data structures reasonable deletion routines can be given. Good insertion techniques may not exist in either of these cases (see Overmars [12]).

Besides dynamization, some other transformations on data structures for searching problems have received attention, like the addition of range restriction capabilities (see Bentley [1] and Lueker [8]) and the transformation of searching problems to the corresponding "all pairs" and "all elements" problems (see Overmars [9]).

In this paper we will consider yet another transformation on searching problems. When we have a dynamically changing set of objects (in a dynamic data structure), it is in some instances important to be able to answer queries over the set as it was at some moment in the past. For example, for a salary administration in a data base, it might be important to be able to ask questions like: How many people had a salary ≥x at some date T. Most known data structures are unable to give this kind of information.

Definition 1.4. Let S be a dynamic data structure for a searching problem.

T_0 = the moment of time at which we initiated the (empty) S.
T_i = the moment just before the ith update is performed on S.

N will always denote the number of the next update (that will be performed at time T_N). In fact, without loss of generality, we will view T_N as being "now". In an ordinary dynamic data structure S we can perform updates at T_N and queries over the situation at T_N. To be able to solve Q "IN THE PAST" (also called the in-the-past version/variant of Q) we would like to transform S into a structure S' that also allows us to perform queries at moments in the past. Performing a query at moment T means performing the query over the pointset as it was after the ith update, assuming that $T_i < T \leq T_{i+1}$.

A first structure for answering queries in the past was given by Dobkin and Munro [4]. It was devised for solving the in-the-past version of the Kth ELEMENT SEARCHING problem, that asks for the kth element in an ordered set of points, and of the RANK SEARCHING problem, that asks for the rank of a given point in an ordered pointset. The structure was used for solving

some polyhedra problems. Very recently, Overmars [11] presented an improved, more efficient structure. He also gave structures for the in-the-past version of the member searching problem and of the range counting problem.

In this paper we focus on general methods for turning dynamic (static) data structures for searching problems into structures for solving the corresponding in-the-past searching problems. In section 2 we give a very general ("brute force") transformation, applicable when a dynamic structure is known for the searching problem at hand, yielding reasonable, but not very efficient, structures for in-the-past problems. In section 3 we show that static data structures for decomposable searching problems can be transformed into very efficient data structures for the corresponding in-the-past problems, but these structures only support insertions. In section 4 we show how to transform fully dynamic data structures for decomposable searching problems into fully dynamic in-the-past structures. In section 5 we give some concluding remarks and a number of directions for further research. Throughout the sections a number of examples are provided to demonstrate the efficiency of the methods presented when applied to particular searching problems. It shows, for example, that d-dimensional range searching in the past can be done within $O(\log^{d+1} N)$ query time, while the insertion time is bounded by $O(\log^{d} N)$ and the deletion time is bounded by $O(\log^{d+1} N)$.

For in-the-past structures we use the following notation:

<u>Notation</u>. Let S be a structure for the in-the-past version of a searching problem at time T_N (hence, after the $N-1^{st}$ update).

$Q_S(N)$ = the time required to perform an in-the-past query on S,
$I_S(N)$ = the time required to insert a point in S,
$D_S(N)$ = the time required to delete a point in S,
$S_S(N)$ = the storage required for S.

2. <u>A very general approach</u>.

Given a structure S for a searching problem Q, a direct way to solve the in-the-past version of Q would be to build a static structure S_i of all points present after each update (after time T_i). This clearly yields a structure S' for solving Q in the past with

$$Q_{S'}(N) = O(\log N + Q_S(N))$$
$$I_{S'}(N) = P_S(N)$$

$$D_{S'}(N) = P_S(N)$$
$$S_{S'}(N) = O(N)S_S(N)$$

The query time of this structure is optimal ($O(\log N)$ is always needed to locate the moment of time among the T_i's and $Q_S(N)$ is needed to query the structure) but the update time is very bad. On the other hand, when nothing more is known about the searching problem, and there only exists a static structure for it, there is only one alternative: list with each update what point is inserted or deleted and when we want to perform a query at time T, walk along all updates preceding T, in this way collecting the points that remain present at T, build a structure out of them and perform the query. It yields a structure S' for solving Q in the past with

$$Q_{S'}(N) = O(N \log N + P_S(N) + Q_S(N))$$
$$I_{S'}(N) = O(1)$$
$$D_{S'}(N) = O(1)$$
$$S_{S'}(N) = S_S(N)$$

If the structure S for the searching problem is dynamic, we can do much better by, in some sense, mixing the two methods described. Instead of building another structure immediately after each update we do so only every time after a number of updates have occured. With the updates in between two structures we only list what point was inserted or deleted. When we want to perform a query over some moment of time T we start with the structure S_j nearest before T. On S_j we perform all updates that have taken place up to T, meanwhile keeping a complete record of all actions we perform. In this way we obtain a structure S_j' that contains the pointset as it was at moment T and that can be queried. Afterwards, we have to undo all the updates we performed on S_j to restore it to its original form, by unwinding the record of actions we created during the updating of the structure. When we make the time gaps between the structures large, we will get a high query time and a low (average) update time, and when we make the gaps small, we will get a low query time but a high (average) update time. Let f be a positive, nondecreasing, smooth integer function. Let $i_1 = 1$ and $i_{j+1} = i_j + f(i_j) + 1$. We build a dynamic data structure S_j of all points currently in the set after each time moment T_{i_j} (i.e., together with the i_j^{th} update). Hence between the structure build after T_{i_j} and the next structure there are $f(i_j)$ updates at which we have done nothing but listing the points that were inserted or deleted. To be able to build a structure S_j after time T_{i_j} efficiently, we have to keep track of all points currently in

the set. Therefore we add to the structure a simple balanced search structure DICT of all points present. Updating DICT takes at most O(log N). So with each update we have to do at least O(log N) work. For most updates this is all we have to do, but, with the updates at T_{i_j} we also have to do a lot of work for building the structure S_j. But, as the number of "cheap" updates is large (depending on the choice for f) the average update time will become reasonable.

Theorem 2.1. Let S be a dynamic data structure for a searching problem Q. For each positive, nondecreasing, smooth integer function f with $f(n) \leq n$ for all n, there exists a structure S' for solving the in-the-past version of Q, such that

$$Q_{S'}(N) = O(\log N + f(N)U_S(N) + Q_S(N))$$

$$I^a_{S'}(N) = O(\log N + P_S(N)/f(N))$$

$$D^a_{S'}(N) = O(\log N + P_S(N)/f(N))$$

$$S_{S'}(N) = O(N/f(N).S_S(N))$$

where $U_S(N) = \max(I_S(N), D_S(N))$.

Proof

Let us consider the update time first. With each update we have to do O(log N) work on updating DICT. The amount of work needed to build S_j (after time moment T_{i_j}, i.e., after the i_j^{th} update) is bounded by $P_S(i_j)$. Charging these costs to the $f(i_{j-1})$ updates that have taken place since the previous building of a structure makes for $P_S(i_j)/f(i_{j-1})$ steps per update. Because $f(n) \leq n$ for all n and f(n) is smooth, this is $O(P_S(i_j)/f(i_j))$ and, using also that P_S is at least linear, we can estimate this by $O(P_S(N)/f(N))$. The bound on the average update time follows.

When we want to perform a query at moment T, we first have to locate the structure S_j built after the largest T_{i_j} that is smaller than T. This can be done in O(log N) steps, using binary search. Next we have to perform at most f(N) updates on S_j, which take at most $U_S(N)$ time each. Keeping the record of actions can be done without any essential loss in efficiency. The query itself takes $Q_S(N)$ steps. Undoing the updates to restore S_j to its original form, using the records of actions, takes the same amount of work as the updates themselves. It follows that a query takes at most $O(\log N + f(N)U_S(N) + Q_S(N))$ steps.

In the same way as we charged the steps needed for building structures to updates, we can charge storage to updates. A structure S_j built after time T_{i_j} takes at most $S_S(i_j)$ storage. Sharing this over the previous $f(i_{j-1})$ updates that took only $O(1)$ storage makes for $S_S(i_j)/f(i_{j-1})$ storage per update. Again, because of the smoothness of f and the fact that S_S is at least linear, we can estimate this by $O(S_S(N)/f(N))$ storage per update. Hence the total amount of storage required by the S structures is bounded by $O(N/f(N).S_S(N))$. The storage required to list the updates and for DICT is bounded by $O(N)$. As $f(N) \leq N$ and S_S is at least linear it follows that $N \leq O(N/f(N).S_S(N))$. The bound on storage required follows.

□

When for a searching problem Q there exists only a half-dynamic data structure S (i.e., a structure that only supports insertions), we can apply the same technique to obtain a half-dynamic data structure S' for solving Q in the past, with the same bounds with respect to query- and insertion time and storage required, as stated in theorem 2.1.

The averages in update time appear because we build the S_j structures at once with the update at T_{i_j}. Instead of doing so we better spread the work over the next $f(i_{j-1})$ updates, each time doing $P_S(i_j)/f(i_{j-1})$ work on the construction. Because f is nondecreasing, it follows that S_j will be ready before we have to start building S_{j+1}. Hence with each update we do at most $O(\log N + P_S(N)/f(N))$ work. Hence, we can get the bound on update time, as stated in theorem 2.1. also as a worst-case bound. It remains to be shown that this does not increase the query time in order of magnitude. (It clearly does not increase the bound on storage required.) When we want to perform a query at moment T with $T_{i_j} < T \leq T_{i_{j+1}}$ it is possible that the structure S_j is not yet ready. If it is we just follow the normal procedure, but if it is not we cannot start performing updates on S_j. But we know that S_{j-1} is ready. So, we start with S_{j-1} and perform on it all updates upto T. These are at most $f(i_{j-1}) + f(i_j) \leq 2f(i_j)$ updates. Hence the amount of work is bounded by $2f(i_j)U_S(N) = O(f(N).U_S(N))$. Hence the query time remains of the same order of magnitude.

<u>Theorem</u> 2.2. Let S be a dynamic data structure for a searching problem Q. For each positive, nondecreasing, smooth integer function f with $f(n) \leq n$ for all n, there exists a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = O(\log N + f(N)U_S(N) + Q_S(N))$$
$$I_{S'}(N) = O(\log N + P_S(N)/f(N))$$
$$D_{S'}(N) = O(\log N + P_S(N)/f(N))$$
$$S_{S'}(N) = O(N/f(N).S_S(N))$$

Example: 2-dimensional convex hull searching.

In Overmars [10] a structure is described for maintaining the convex hull of a 2-dimensional pointset at the cost of $O(\log^2 n)$ per update, such that for all points x queries of the form: does x lie inside, outside or on the convex hull, can be answered within $O(\log n)$ steps. The structure takes $O(n)$ storage and can be built in $P_S(n) = O(n \log n)$ steps. Choosing $f(N) = \sqrt{N}/\log N$, we get a structure S' for in-the-past convex hull searching, such that

$$Q_{S'}(N) = O(\sqrt{N} \log N)$$
$$I_{S'}(N) = O(\sqrt{N} \log^2 N)$$
$$D_{S'}(N) = O(\sqrt{N} \log^2 N)$$
$$S_{S'}(N) = O(N.\sqrt{N} \log N)$$

Other trade-offs between query and update time (and storage required) can be obtained by different choices of f.

When the bounds on the insertion and deletion time of the dynamic data structure S for Q are not of the same order of magnitude, we can sharpen the result of theorem 2.2. somewhat. Let f(n) and g(n) be two integer functions (positive nondecreasing and smooth). We build a S_j structure after moment $T_{i_1} = T_1$ and after a moment T_{i_j} if the number of insertions that have taken place since $T_{i_{j-1}}$ has become $f(i_{j-1}) + 1$ or if the number of deletions that have taken place since $T_{i_{j-1}}$ has become $g(i_{j-1}) + 1$. It follows that in between $T_{i_{j-1}}$ and T_{i_j} there have been at most $f(i_{j-1})$ insertions and $g(i_{j-1})$ deletions. Hence to perform a query at some moment T we have to do at most f(N) insertions and g(N) deletions on the last previous structure to obtain the situation at time T. When we build a S_j structure (at time T_{i_j}) because the number of insertions since $T_{i_{j-1}}$ has become $f(i_{j-1}) + 1$, we charge the costs to these $f(i_{j-1}) + 1$ insertions. When we build S_j because the number of deletions since $T_{i_{j-1}}$ has become $g(i_{j-1}) + 1$, we charge the costs to the $g(i_{j-1}) + 1$ deletions. It yields an average insertion time of $O(\log N + P_S(N)/f(N))$ and an average deletion time of $O(\log N + P_S(N)/g(N))$. These bounds can be changed into worst-case

bounds by spreading the work for building S_j over the next $f(i_{j-1})$ insertions when it had to be built because the number of insertions became too big or over the next $g(i_{j-1})$ deletions, respectively, otherwise. One easily shows that in this way there are at most two structures "under construction" and hence, that the query time remains of the same order of magnitude as in the average case. This leads to the following result.

Theorem 2.3. Let S be a dynamic data structure for a searching problem Q. For all positive, nondecreasing, smooth integer functions f and g with $f(n) \leq n$ and $g(n) \leq n$, there exist a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = O(\log N + f(N)I_S(N) + g(N)D_S(N) + Q_S(N))$$
$$I_{S'}(N) = O(\log N + P_S(N)/f(N))$$
$$D_{S'}(N) = O(\log N + P_S(N)/g(N))$$
$$S_{S'}(N) = O(N/\min(f(N),g(N)).S_S(N))$$

When the amount of storage required for the dynamic data structure S is smaller than the time required to build S, it is often possible to obtain better bounds for the update time by copying structures, rather than by rebuilding them from scratch. Let $COP_S(n)$ denote the time required to copy a S-structure containing n points. (In general we can take $COP_S(n) = S_S(n)$.) We will use the structure and algorithm described above for the average case (yielding theorem 2.3. with average rather than worst-case update time bounds) except that, when we need to build a structure S_j after moment T_{i_j}, we do so by copying the structure S_{j-1} and performing all updates that occurred between $T_{i_{j-1}}$ and T_{i_j} on the copy. Now, the copy will hold the situation after T_{i_j} and we can use it as S_j. (Note that we do not need our simple search structure DICT anymore.)

Theorem 2.4. Let S be a dynamic data structure for a searching problem Q. For all positive, nondecreasing, smooth integer functions f and g with both $f(n) \leq n$ and $g(n) \leq n$, there exist a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = O(\log N + f(N)I_S(N) + g(N)D_S(N) + Q_S(N))$$
$$I^a_{S'}(N) = O(COP_S(N)/f(N) + I_S(N))$$
$$D^a_{S'}(N) = O(COP_S(N)/g(N) + D_S(N))$$
$$S_{S'}(N) = O(N/\min(f(N), g(N)).S_S(N))$$

Proof

The bounds on the query time and on the amount of storage required follow from theorem 2.3. When we have to build a structure S_j because the number of insertions after $T_{i_{j-1}}$ has become too big, we charge the costs for copying S_{j-1} to the $f(i_{j-1})$ insertions that have taken place, otherwise we charge the costs to the $g(i_{j-1})$ deletions that have taken place. Each update we have to perform on the copied structure, we charge to the update itself. The average bounds for insertions and deletions follow.

□

Changing the averages into worst-case bounds is somewhat harder than in the previous cases. Instead of starting to build a structure S_j after T_{i_j}, taking care that it will be ready soon enough, we start with the work immediately after $T_{i_{j-1}}$ and we will take care that it is ready by the time we come to T_{i_j}. (The technique used is similar to the one used in the proof of theorem 1 in [14].) Let us assume S_{j-1} was indeed ready at moment $T_{i_{j-1}}$. We immediately start copying this structure, doing $COP_S(i_{j-1})/f(i_{j-1}) + I_S(i_j)$ work with each insertion and $COP_S(i_{j-1})/g(i_{j-1}) + D_S(i_j)$ work with each deletion. Assume the copy is ready after n insertions and d deletions and that we have got w time left with the last update. Then clearly

$$n.COP_S(i_{j-1})/f(i_{j-1}) + n.I_S(i_j) + d.COP_S(i_{j-1})/g(i_{j-1}) + d.D_S(i_j) - w = COP_S(i_{j-1})$$

and hence,

$$n.I_S(i_j) + d.D_S(i_j) \le (f(i_{j-1}) - n)COP_S(i_{j-1})/f(i_{j-1}) + w$$

and (*)

$$n.I_S(i_j) + d.D_S(i_j) \le (g(i_{j-1}) - d)COP_S(i_{j-1})/g(i_{j-1}) + w$$

As the structure S_j will have size at most i_j, the lefthand side of these formulas is an upperbound to the amount of work needed to perform the updates that have taken place since $T_{i_{j-1}}$ up to now and that have to be performed on the copied structure to make it the appropriate S_j. Let's spend the w work left immediately on performing updates (in the appropriate order). From this moment on we spend with each insertion $COP_S(i_{j-1})/f(i_{j-1}) + I_S(i_j)$ work on performing updates and with each deletion $COP_S(i_{j-1})/g(i_{j-1}) + D_S(i_j)$ work. We want that the process overtakes itself before $f(i_{j-1})$ insertions or $g(i_{j-1})$ deletions have taken place. When we perform an insertion the amount of work that still needs to be done increases by at most $I_S(i_j)$

(because also this insertion needs to be performed on S_j) and decreases by $COP_S(i_{j-1})/f(i_{j-1}) + I_S(i_j)$. Hence the work that needs to be done is at least $COP_S(i_{j-1})/f(i_{j-1})$ less than before the insertion. Similar, one can show that the amount of work that still needs to be done after a deletion is $COP_S(i_{j-1})/g(i_{j-1})$ less than before the deletion. From (*) and the fact that we already did w work on performing updates, it follows that the process will overtake itself within $f(i_{j-1}) - n$ insertions and also within $g(i_{j-1}) - d$ deletions and hence, that S_j will be up-to-date within at most $f(i_{j-1})$ insertions and also within $g(i_{j-1})$ deletions. From this moment on upto T_{i_j} we just perform the updates that come in on S_j, to keep it up to date.

Theorem 2.5. Let S be a dynamic data structure for a searching problem Q. For all positive, nondecreasing, smooth integer functions f and g with both $f(n) \leq n$ and $g(n) \leq n$, there exist a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = O(\log N + f(N)I_S(N) + g(N)D_S(N) + Q_S(N))$$
$$I_{S'}(N) = O(COP_S(N)/f(N) + I_S(N))$$
$$D_{S'}(N) = O(COP_S(N)/g(N) + D_S(N))$$
$$S_{S'}(N) + O(N/\min(f(N), g(N)).S_S(N))$$

Example: 2-dimensional convex hull searching.

As stated above, there exists a structure for the problem with an update time of $O(\log^2 n)$ that uses $O(n)$ storage. Hence, we can take $COP_S(n) = O(n)$. Choosing $f(N) = g(N) = \sqrt{N}/\log N$, we get a structure S' for in-the-past convex hull searching, such that

$$Q_{S'}(N) = O(\sqrt{N} \log N)$$
$$I_{S'}(N) = O(\sqrt{N} \log N)$$
$$D_{S'}(N) = O(\sqrt{N} \log N)$$
$$S_{S'}(N) = O(N \sqrt{N} \log N)$$

3. In-the-past searching for decomposable searching problems: a half-dynamic structure

The main property of time is that each next moment is larger (later) than all previous ones. We will first describe a data structure, called a RI-tree, that stores a number of moments of time in such a way that we can search for a specific time moment in $O(\log N)$ steps (N the number of stored moments) while insertions of new, later moments of time take

only O(1). Let PB_h denote the perfect binary tree of height h (containing 2^h points).

Definition 3.1. A RI-tree of height 0 consists of 1 point. A RI-tree of height $h > 0$ consists of a root which has PB_{h-1} as a left subtree and a RI-tree of height at most h-1 as a right subtree. Points are stored in the leaves of a RI-tree (see figure 1).

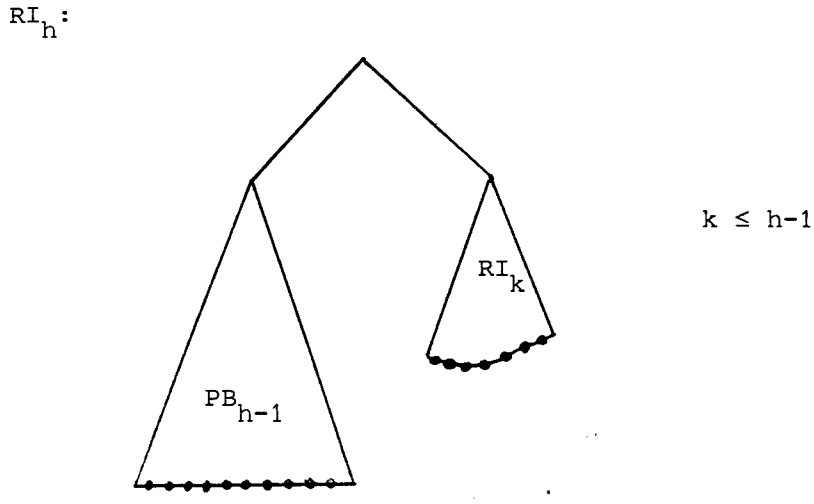


Figure 1.

Note that there is exactly one way of storing n points in a RI-tree. Let $i = \lfloor \log n \rfloor$; If $n = 2^i$, the RI-tree of n points is a PB_i. Otherwise, it consists of a root with PB_i as a left subtree and a RI-tree of $n - 2^i$ points as a right subtree. See figure 2 for example of a RI-tree containing 11 points. It should be noted that the class of RI-trees is a proper subclass

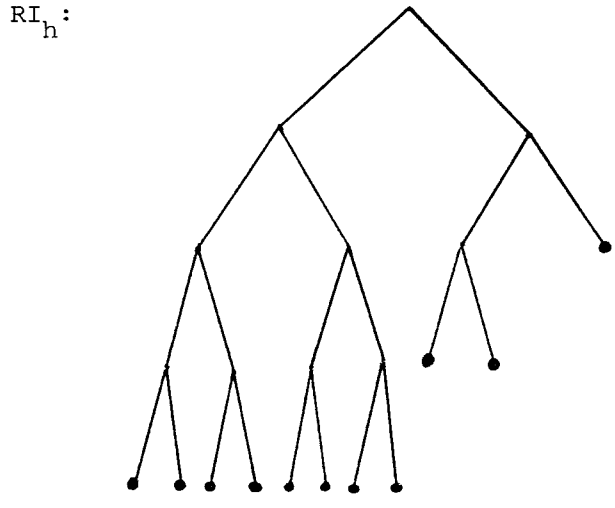
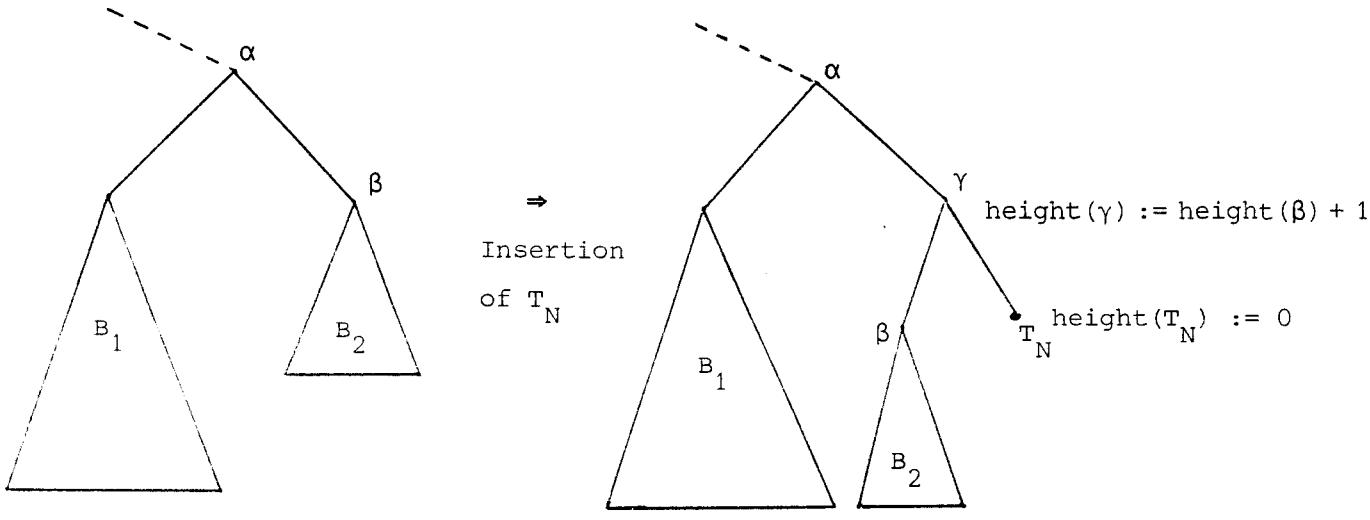
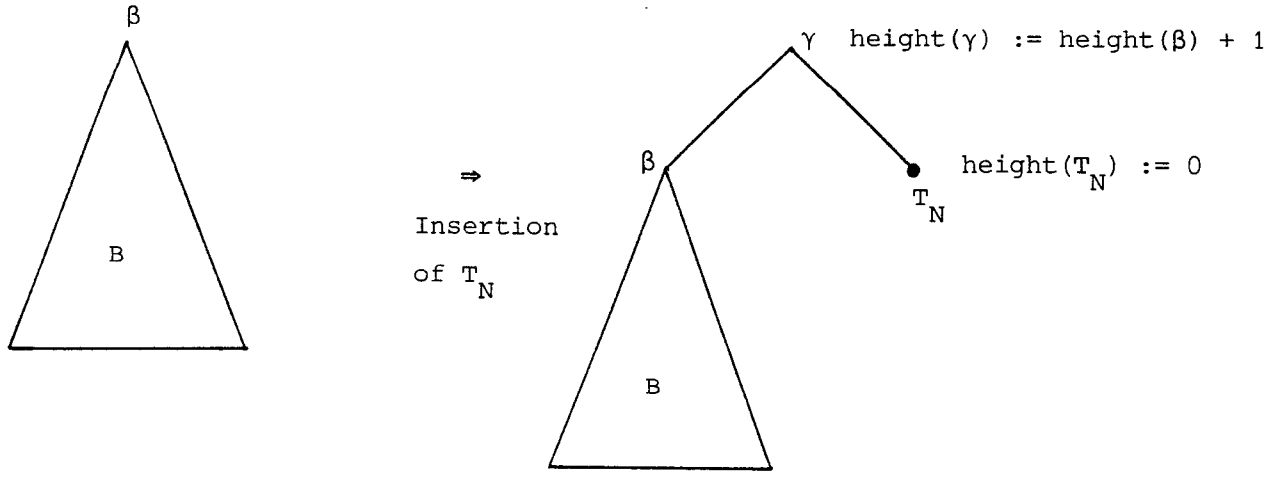


Figure 2.

of the class of leftist trees (see Knuth [7]). RI-trees clearly are balanced in a (very) strong sense and hence queries can be performed on them in O(log N) steps when the trees contain N points (moments of time). To insert a new moment of time T_N in a RI-tree (at the right side) we first determine the deepest internal node α on the rightmost path of the RI-tree with height(lson(α)) > height(rson(α)). Let β be the rightson of α. It follows that the subtree rooted at β is perfectly balanced (containing 2^k points, some k) and hence we can perform the actions displayed in figure 3a. When no such node α exists, the RI-tree must be perfectly balanced and we can perform the actions displayed in figure 3b. The time needed for performing these actions is clearly bounded by O(1). Hence,



a.



b.

Figure 3.

the time bound for insertions depends only on the time needed to locate the appropriate node α. To be able to find this node efficiently, we keep nodes α on the rightmost path with height(lson(α)) > height(rson(α)) in a stack, with the deepest node on top. In this way the appropriate node can be found in O(1) (it is the node on top of the stack). When we perform an insertion as displayed in figure 3a, we have to perform the following stack updates: (i) if height(lson(α)) > height(β) + 1 then leave α on the stack otherwise remove α, (ii) if height(β) > 0 then put γ on the stack. When we perform an insertion as displayed in figure 3b and height(β) > 0 we put γ on the stack. These operations suffice to keep the stack up to date. We need to maintain the heights of nodes but, as an insertion does not change the height of old nodes, this can be done in O(1) as well.

Theorem 3.2. Queries on a RI-tree of N points (moments of time) can be performed in O(log N) steps while an insertion at the right side takes only O(1).

One easily shows that it is also possible to delete points at the rightmost side in O(1) steps. The RI-tree has a number of interesting applications on its own. It can, for instance, be use to represent a stack which, besides the O(1) behaviour for popping and pushing elements, allows us to find the kth element on the stack in O(log N) (or even O(log k) using a double linked structure) steps.

We will use the RI-tree for a quite different task. Given a dynamically changing pointset for a decomposable searching problem Q, we store all moments of time T_i at which an update is made in the set in a RI-tree. In this section we will restrict ourselves to the case in which all updates are insertions. Let S be a static data structure known for the decomposable searching problem Q. To an internal node α of the RI-tree we associate a S-structure S_α of all points in the set that were present during the whole interval of time spanned by node α but that where not present during the whole interval of time spanned by f(α), the father of α. In the case all updates are insertions, it are exactly those points that were inserted at the moments of time in lson(f(α)) if α = rson (f(α)). Note that the S-structures of the nodes that are leftson of their father are empty. (see figure 4.)

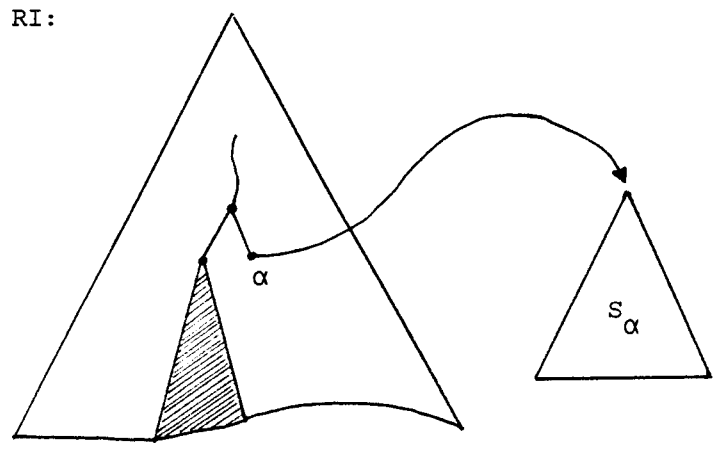


Figure 4.

When we want to perform a query with search object x at time T, we first search with T in the RI-tree, to find the latest T_i with $T_i < T$. One easily verifies that (i) each point of the set present at T is contained in exactly one associated structure S_α of a node α on the search path towards T_i, and (ii) each point contained in such an associated structure S_α was present at time T. So, the set of points present at T is partitioned over the structures S_α associated to nodes on the search path. To perform the query with x on the set, we query all these structures separately and then combine the answers using the composition operator □.

<u>Lemma</u> 3.3. The query time on the structure S' described above is bounded by

$$Q_{S'}(N) = \begin{cases} O(Q_S(N)) & \text{if } Q_S(N) = \Omega(N^\varepsilon),\ \varepsilon > 0. \\ O(\log N)Q_S(N) & \text{otherwise} \end{cases}$$

<u>Proof</u>

The depth of a RI-tree containing N points is $\lceil \log N \rceil$. It follows that, to perform an in-the-past query we have to perform queries on at most O(log N) associated S-structures, namely on at most one structure of size 1, one structure of size 2, one structure of size 4, ..., and one structure of size $2^{\lceil \log N \rceil - 1}$. It follows that the time needed to perform an in-the-past query is bounded by

$$O(\log N) + \sum_{i=0}^{\lceil \log N \rceil - 1} Q_S(2^i).$$

One easily verifies that this is bounded by $O(Q_S(N))$ if $Q_S(N) = \Omega(N^\varepsilon)$ for some $\varepsilon > 0$ and that it is bounded by $O(\log N).Q_S(N)$ otherwise.

□

<u>Lemma</u> 3.4. $S_{S'}(N) = \begin{cases} O(S_S(N)) & \text{if } S_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)S_S(N) & \text{otherwise.} \end{cases}$

<u>Proof</u>

Consider all S-structures associated with nodes with some fixed depth d. These are at most $\lfloor \frac{1}{2}.2^d \rfloor$ structures, each of size at most $2^{\lceil \log N \rceil - d}$. Hence the storage required for these structures is bounded by $\lfloor \frac{1}{2}.2^d \rfloor . S_S(2^{\lceil \log N \rceil - d})$. It follows that the total amount of storage required is bounded by

$$O(N) + \sum_{d=1}^{\lceil \log N \rceil} 2^{d-1} . S_S(2^{\lceil \log N \rceil - d})$$

One easily verifies that this is bounded by $O(S_S(N))$ if $S_S(N) = \Omega(N^{1+\varepsilon})$ for some $\varepsilon > 0$ and that it is bounded by $O(\log N)S_S(N)$ otherwise (S_S being at least linear).

□

It remains to be shown that insertions (at the rightside) in these augmented RI-trees can be performed efficiently. Using the method of inserting points in an ordinary RI-tree, as displayed in figure 3a and 3b, the only extra action that needs to be performed is building a S-structure S_{T_N} associated to the new inserted moment of time T_N, of all points inserted at the moments stored in the subtree rooted at β. All other associated structures remain the same. This building of S_{T_N} will sometimes be very costly (especially in the case of figure 3b when we need to build a structure of O(N) points) but mostly the S_{T_N} will be small. Hence, the average insertion time will remain low.

<u>Lemma</u> 3.5. $I^a_{S'}(N) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)P_S(N)/N & \text{otherwise} \end{cases}$

<u>Proof</u>

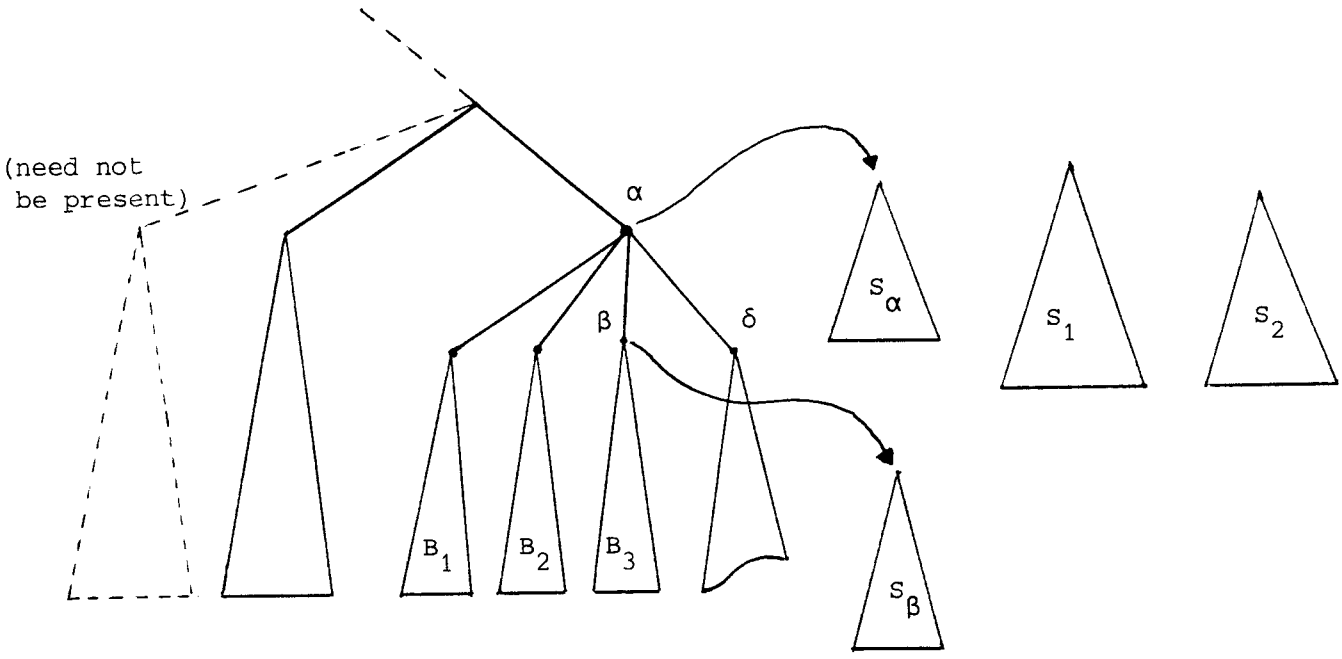
From the description of the insertion procedure it follows that associated S-structures that are once built will never be rebuilt or thrown away. After N insertions there are at most $\lfloor \frac{1}{2}.2^d \rfloor$ S-structures of size $2^{\lceil \log N \rceil - d}$ for each $1 \le d \le \lceil \log N \rceil$ (see the proof of lemma 3.4.) Hence the total time required for building associated S-structures during the first N insertions is bounded by

$$\sum_{d=1}^{\lceil \log N \rceil} 2^{d-1} . P_S(2^{\lceil \log \rceil N - d})$$

One easily verifies that this is bounded by $O(P_S(N))$ if $P_S(N) = \Omega(N^{1+\varepsilon})$ for some $\varepsilon > 0$ and that it is bounded by $O(\log N)P_S(N)$ otherwise (P_S being at least linear). As the updating of the RI-tree itself only takes $O(1)$ per update, the bound on the average insertion time follows.

□

It is possible to change the average bound on insertion time into a worst-case bound, but the method is quite complicated. It is very similar to the method described in Overmars and van Leeuwen [13] for changing the average time bounds in the dynamization of decomposable searching problems into worst-case bounds. We will only give a sketch of the method here. The details are left to the interested reader. We allow nodes on the rightmost path to have 1, 2 or 3 perfect binary left subtrees of the same depth. To each node α we associate a structure S_α containing all points that where present during the whole interval below α, but not during the whole interval below the father of α. As soon as a node α on the rightmost path gets 2 leftsons we start building a S-structure S_1 of all the points in those two left subtrees and the points in the current S_α. Let the height of α be i, we will see to it that this structure is ready after 2^i insertions. Let the number of points that come in this structure be k. With each insertion we do $P_S(k)/2^i = O(P_S(2^i)/2^i)$ work on the construction. After 2^{i-1} insertions α will get a third left subtree (this follows from the rest of the procedure). From this moment on we start building another S-structure S_2 of all the points in this new left subtree. This structure will get size 2^{i-1} and we do $P_S(2^{i-1})/2^{i-1} = O(P_S(2^i)/2^i)$ work with each insertion so it will be ready at the same time as S_1 (In the meantime no fourth leftson will occur!) At this moment we perform the actions as displayed in figure 5. Hence we add a new leftson γ to the father of α with B_1 and B_2 as left, respectively right subtree. S_α becomes the structure associated with γ (one easily verifies that it contains the appropriate points), and S_1 becomes the new S_α. Moreover we associate S_2 with δ. This maintains the structure correctly, provided that the father of α had less than 3 leftsons, and that indeed S_1 and S_2 were ready before α got a fourth leftson. (Note that α has only 1 leftson left, so there is enough room again for a new one). The proof goes along the same lines as the proofs for the correctness of the methods in [13].


(need not
be present)
α
β
δ
B_1
B_2
B_3
S_α
S_1
S_2
S_β

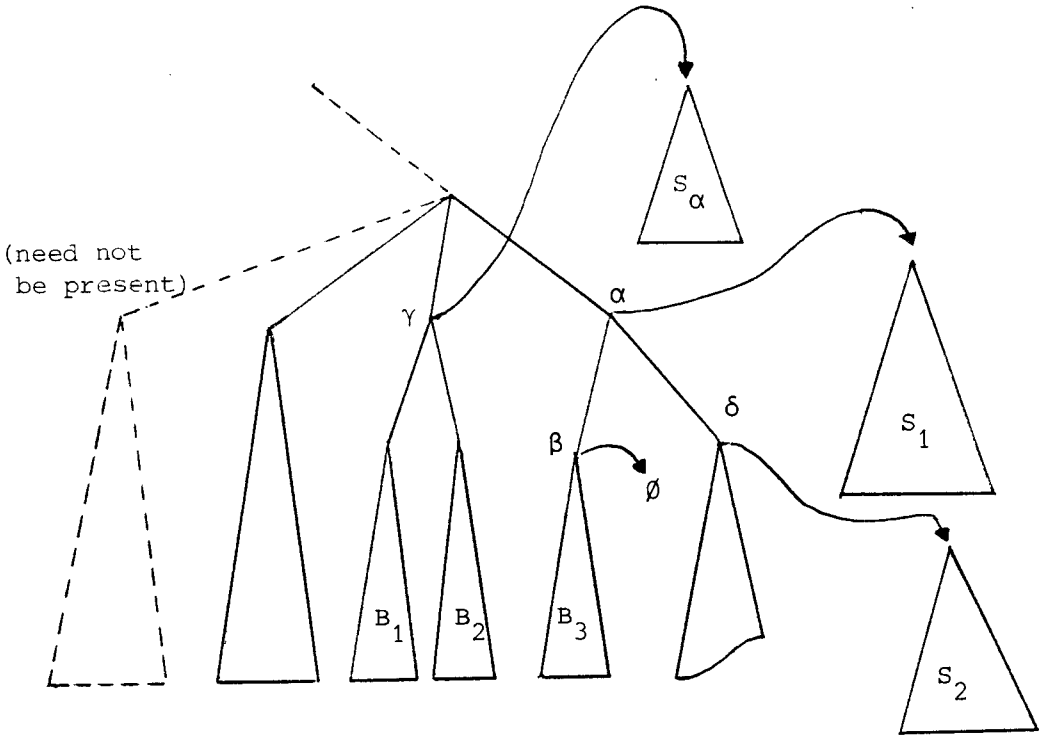

(need not
be present)
γ
α
β
δ
ø
B_1
B_2
B_3
S_α
S_1
S_2

Figure 5.

It is clear that the depth of the structure described is bounded by $\lceil \log N \rceil$ again. With an insertion, we have to do $O(P_S(2^i)/2^i)$ work for each node on the rightmost path of the tree. Hence, the total amount of work is bounded by

$$\sum_{i=1}^{\lceil \log N \rceil} O(P_S(2^i)/2^i) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = (N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)P_S(N)/N & \text{otherwise} \end{cases}$$

The query time and the amount of storage required, clearly remain of the same order of magnitude. Hence the bounds in lemma 3.5. is even valid as a worst-case bound.

Theorem 3.6. Let S be a static data structure for a decomposable searching problem Q. There exists a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = \begin{cases} O(Q_S(N)) & \text{if } Q_S(N) = \Omega(N^{\varepsilon}),\ \varepsilon > 0 \\ O(\log N)Q_S(N) & \text{otherwise} \end{cases}$$

$$I_{S'}(N) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)P_S(N)/N & \text{otherwise} \end{cases}$$

$$S_{S'}(N) = \begin{cases} O(S_S(N)) & \text{if } S_S(N) = \Omega(N^{1+\varepsilon}).\ \varepsilon > 0 \\ O(\log N)S_S(N) & \text{otherwise} \end{cases}$$

Proof

See lemma 3.3., 3.4. and 3.5. and the above discussion.

□

Examples:

(i) Nearest neighbor searching

For the 2-dimensional nearest neighbor searching problem there exists a static data structure S, based on the Voronoi diagram (see Shamos [16], Shamos and Hoey [17]) with a query time of $O(\log n)$ while $P_S(n) = O(n \log n)$ and $S_S(n) = O(n)$ (see Kirkpatrick [6]). Hence, there exists a half-dynamic data structure S' for solving the in-the-past version of the nearest neighbor searching problem, yielding

$$Q_{S'}(N) = O(\log^2 N)$$
$$I_{S'}(N) = O(\log^2 N)$$
$$S_{S'}(N) = O(N \log N)$$

(ii) Range searching

For the d-dimensional range searching problem, a static data structure is known with a query time of $O(\log^{d-1} n)$ (+ the number of answers,

which we ignore) and a building time of $O(n \log^{d-1} n)$, using $O(n \log^{d-1} n)$ storage (see Willard [19]). The problem is decomposable, hence we can apply theorem 3.6. to get a structure S' for range searching in the past, such that

$$Q_{S'}(N) = O(\log^d N)$$
$$I_{S'}(N) = O(\log^d N)$$
$$S_{S'}(N) = O(N \log^d N)$$

4. In-the-past searching for decomposable searching problems: a fully dynamic structure.

A subclass of decomposable searching problems are the so-called decomposable counting problems.

Definition 4.1. A decomposable searching problem Q is called a DECOMPOSABLE COUNTING PROBLEM if and only if for any set V and any subset $V_1 \subseteq V$, $Q(x,V\backslash V_1) = \Delta(Q(x,V), Q(x,V_1))$ where Δ can be computed in constant time.

For example, the RANGE COUNTING problem, that asks for the number of points in a multidimensional pointset that lie in a given range, is an order decomposable counting problem ($\Delta = -$).

Solving the in-the-past version of a decomposable counting problem can be done quite easily. We only make the restriction that points are never reinserted once they are deleted. Answering a query at time T can be done by performing the query on all points that have been inserted before T and "subtracting" the answer over all points that have been deleted before T, using the "subtraction" operator Δ. This suggests that we should use two structures MAIN and GHOST, both of the form described in the previous section, MAIN containing all points that have been inserted and GHOST containing all points that have been deleted. Hence, insertions we perform on MAIN and deletions we perform by inserting the point that needs to be deleted, in GHOST. To perform a query with object x at time T we perform a query with x at T on both MAIN and GHOST and subtract the answer over GHOST from the answer over MAIN.

Theorem 4.2. Let S be a static data structure for a decomposable counting problem Q. There exists a fully dynamic structure S' for solving Q in the past, such that

$$Q_{S'}(N) = \begin{cases} O(Q_S(N)) & \text{if } Q_S(N) = \Omega(N^{\varepsilon}),\ \varepsilon > 0 \\ O(\log N) Q_S(N) & \text{otherwise} \end{cases}$$

$$I_{S'}(N) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)P_S(N)/N & \text{otherwise} \end{cases}$$

$$D_{S'}(N) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)P_S(N)/N & \text{otherwise} \end{cases}$$

$$S_{S'}(N) = \begin{cases} O(S_S(N)) & \text{if } S_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)S_S(N) & \text{otherwise} \end{cases}$$

Proof

Both MAIN and GHOST contain at most N points. The bounds follow from theorem 3.6.

□

Example: d-Dimensional range counting ($d \geq 2$)

There exists a static data structure S for this problem with a query time of $O(\log^{d-1} n)$, $P_S(n) = O(n \log^{d-1} n)$ and $S_S(n) = O(n \log^{d-1} n)$ (see Willard [19]). Hence we can apply theorem 4.2. to obtain a fully dynamic data structure S' for the in-the-past version of the d-dimensional range counting problem, such that

$$Q_{S'}(N) = O(\log^d N)$$
$$I_{S'}(N) = O(\log^d N)$$
$$D_{S'}(N) = O(\log^d N)$$
$$S_{S'}(N) = O(N \log^d N)$$

Note that these bounds are exactly the same as the bounds for the structure for range counting in the past devised in Overmars [11]. His structure however also works in the 1-dimensional case.

Let us again concentrate on decomposable searching problems in general. It has been shown (Bentley and Saxe [2]) that there cannot exist a dynamization method for decomposable searching problems in general yielding both a low query time and a low deletion time bound. Hence, there clearly cannot exist a general method for turning static data structures for decomposable searching problems into efficient fully dynamic data structures for solving the problems in the past. Therefore, we assume in the sequel that the data structure S known for the decomposable searching problem is fully dynamic itself. We will show that in this case efficient fully dynamic in-the-past structures can be devised.

We will first devise a method for getting good average update time bounds. The structure we use is exactly the same as the structure described in the previous section (yielding an average insertion time bound) except that the associated structures S_α are dynamic. Also the insertion procedure carries over, except that the structure S_{T_N} is built only from those points in the subtree rooted at β that are still present (not yet deleted). To delete a point p we first insert T_N in the RI-tree in exactly the same way as when we inserted a point (and so we have to built some structure S_{T_N}). Let α be the node on the rightmost path containing p. (Such node α exists because p was present up to now and there is exactly one such node.) To be able to find α efficiently, we add to the structure a dictionary DICT in which we keep this information for each point present. Hence with each insertion and deletion we have to update DICT which takes at most O(log N). Moreover we have to update DICT when we build structures but, when we also keep with each point a pointer to its location in DICT, the time required for this is dominated by the building time. After we have found the appropriate node α we have to delete p from S_α because p is no longer present during the whole interval of time below α. But p was present during the whole interval of time spanned by lson(α) and by each node that is leftson of some node on the rightmost path below α. Hence we have to insert p in the S-structures associated with these nodes (see figure 6).

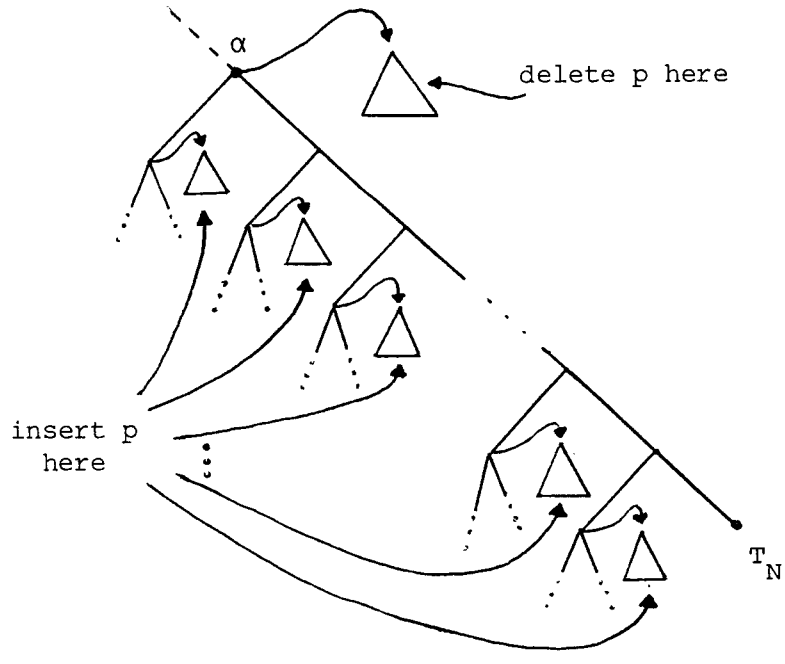


Figure 6.

One easily verifies that in this way the structure is maintained correctly. The bounds on query time, average insertion time and storage required do not change (in order of magnitude). The deletion time is given by the following lemma:

<u>Lemma</u> 4.3. $D^a_{S'}(N) = \begin{cases} O(I_S(N) + D_S(N)) & \text{if } I_S(N) = \Omega(N^\varepsilon),\ \varepsilon > 0 \\ O(\log N . I_S(N) + D_S(N)) & \text{otherwise} \end{cases}$

<u>Proof</u>

Inserting T_N in the structure takes (average) $O(P_S(N)/N)$ if $P_S(N) = \Omega(N^{1+\varepsilon})$ for some $\varepsilon > 0$ and $O(\log N)P_S(N)/N$ otherwise, according to lemma 3.5. As $P_S(N)/N \le I_S(N)$ this can be estimated by $O(I_S(N))$ if $I_S(N) = \Omega(N^\varepsilon)$ for some $\varepsilon > 0$ and by $O(\log N)I_S(N)$ otherwise. Finding the appropriate node α takes only $O(\log N)$ steps. Deleting the point p from the associated structure S_α takes at most $O(D_S(N))$ steps. As the depth of the RI-tree is bounded by $\lceil \log N \rceil$ we have to perform an insertion of p on associated structures of size at most 2^i for at most each i with $0 \le i \le \lceil \log N \rceil - 1$. This takes a total of

$$\sum_{i=0}^{\lceil \log N \rceil - 1} I_S(2^i) = \begin{cases} O(I_S(N)) & \text{if } I_S(N) = \Omega(N^\varepsilon),\ \varepsilon > 0 \\ O(\log N) I_S(N) & \text{otherwise} \end{cases}$$

The bound follows.

□

Changing the average time bounds on insertion and deletion time into worst-case bounds is again possible. We will use the structure for worst-case bounds described in the previous section again. Note that in the case both insertions and deletions occur, after the "rotation" displayed in figure 5, not only the structures at γ, α and δ need to be rebuilt, but also the structures at lson(γ), rson(γ) and β. Hence, at the moment α gets two left subtrees we start building the structure S_1 that will become associated with α, the structure that must be associated with γ (this structure will no longer necessarily be the old S_α) and the structures that will become associated with lson(γ) and rson(γ). As soon as the third left subtree of α comes in, we start building the structure S_2 that will become associated to δ and the structure that must be associated with β. We can take care, doing with each update $O(P_S(2^i)/2^i)$ work on the construction (i is the height of α) that all structures are ready after a total of 2^i updates have taken place and hence, before a fourth left subtree would

come in. There is only one problem. Deletions that take place in the meantime might cause that some points need to be deleted in the new S_α and S_δ we are busy building and that some points need to be inserted in the new S_γ and S_β. We cannot perform these updates on the structures under construction. (We of course do perform them on the old structures such that the whole structure S' remains appropriate for query answering.) Therefore, we add to each of these structures a buffer BUF. When we need to perform an update on the structure we put this update in BUF. To make sure that there will be enough time left to perform this buffered update after the construction of the structure is finished and before it must be ready (i.e. before 2^i updates have taken place since the construction began) we do $D_S(2^{i+1})$ or $I_S(2^{i+1})$ work on the construction with this deletion or insertion respectively. It follows that the construction will be finished $D_S(2^{i+1})$ or $I_S(2^{i+1})$ steps earlier respectively, and, as the structure will have size at most 2^{i+1}, there is enough time left to perform the buffered update. (This method is similar to the method described in much more detail in [13].) Hence, with a deletion we have to do $O(P_S(2^i)/2^i)$ work on constructing new structures for each node on the rightmost path (like we do with each insertion). Next we search for the node α whose associated structure contains the point p that must be deleted. We delete it here, and moreover we put the point in the buffer BUF of the new structure for α we are busy building and we do $D_S(2^i)$ work on the construction of this new S_α (i the height of α). Next we insert the point in the structures associated to nodes that are leftson of a node on the rightmost path below α (or of α itself). Moreover we put the point in the appropriate buffers of structures we are busy building and we speed up the construction of these structures doing $I_S(2^i)$ work extra (i the height). This will maintain the structure S' correctly. (The details are left to the interested reader, who is suggested to read [13] first). It shows that the average bounds on insertion and deletion time are also valid as worst-case bounds.

<u>Theorem</u> 4.4. Let S be a dynamic data structure for a decomposable searching problem Q. There exists a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = \begin{cases} O(Q_S(N)) & \text{if } Q_S(N) = \Omega(N^\varepsilon),\ \varepsilon > 0 \\ O(\log N)Q_S(N) & \text{otherwise} \end{cases}$$

$$I_{S'}(N) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)P_S(N)/N & \text{otherwise} \end{cases}$$

$$D_{S'}(N) = \begin{cases} O(I_S(N) + D_S(N)) & \text{if } I_S(N) = \Omega(N^{\varepsilon}),\ \varepsilon > 0 \\ O(\log N.I_S(N) + D_S(N)) & \text{otherwise} \end{cases}$$

$$S_{S'}(N) = \begin{cases} O(S_S(N)) & \text{if } S_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N).S_S(N) & \text{otherwise} \end{cases}$$

Proof

This follows from theorem 3.6., lemma 4.2. and the above discussion.

□

Examples:

(i) Nearest neighbor searching

In Overmars [10] a dynamic structure for nearest neighbor searching is given, yielding a query time of $O(\log n)$ and an insertion and deletion time of $O(n)$, while $P_S(n) = O(n \log n)$ and $S_S(n) = O(n \log n)$. Applying theorem 4.4. yields a fully dynamic data structure S' for nearest neighbor searching in the past, such that

$$Q_{S'}(N) = O(\log^2 N)$$
$$I_{S'}(N) = O(\log^2 N)$$
$$D_{S'}(N) = O(N)$$
$$S_{S'}(N) = O(N \log^2 N)$$

To get other trade-offs between query and deletion time we can first apply theorem 1.3. to the structure S for the ordinary problem. When we choose e.g. $f(n) = g(n) = \sqrt{n}/\sqrt{\log n}$, then theorem 1.3. yields a structure S' for the ordinary problem, such that

$$Q_{S'}(n) = O(\sqrt{n} \log n)$$
$$I_{S'}(n) = O(\log n)$$
$$D_{S'}(n) = O(\sqrt{n} \log n)$$
$$S_{S'}(n) = O(n \log n)$$

One easily shows that $P_{S'}(n) = O(n \log n)$. When we apply theorem 4.4. on this structure we get a structure S" for nearest neighbor searching in the past, such that

$$Q_{S''}(N) = O(\sqrt{N} \log N)$$
$$I_{S''}(N) = O(\log^2 N)$$
$$D_{S''}(N) = O(\sqrt{N} \log N)$$
$$S_{S''}(N) = O(N \log^2 N)$$

The storage required can even be decreased to $O(N \log N \log\log N)$ using a result of Gowda and Kirkpatrick [5].

(ii) Range searching

For d-dimensional range searching a dynamic data structure S is known with a query time of $O(\log^d n)$ (+ the number of answers) an update time of $O(\log^d n)$, a building time of $O(n \log^{d-1} n)$ while $S_S(n) = O(n \log^{d-1} n)$ (see Lueker [8] and Willard [20]). Applying theorem 4.4. yields a structure S' for d-dimensional range searching in the past, such that

$$Q_{S'}(N) = O(\log^{d+1} N)$$
$$I_{S'}(N) = O(\log^d N)$$
$$D_{S'}(N) = O(\log^{d+1} N)$$
$$S_{S'}(N) = O(N \log^d N)$$

5. Concluding remarks

We have given a number of general techniques for transforming data structures for searching problems into structures for the in-the-past versions of those searching problems. First, a very general, brute force technique was given, applicable to every searching problem for which a dynamic data structure is known. Although the bounds are reasonable, the efficiency of the resulting structures is not very good, compared with the bounds of the structures known for the ordinary problems. For decomposable searching problems much better results were obtained. Given a static data structure for a decomposable searching problem we showed how to transform it into a structure for solving the corresponding in-the-past problem with an extra factor of only O(log N) in query time. This structure only permits us to perform insertion. When a dynamic data structure for the ordinary problem is known, the resulting structure handles deletions as well. An often considered (but nowhere mentioned) extension of the class of decomposable searching problems is the notion of C(n)-decomposability.

Definition 5.1. A searching problem Q is called C(n)-DECOMPOSABLE if and only if for any partition $A \cup B = V$ and any query object x, $Q(x,V) = \square(Q(x,A),Q(x,B))$ where $\square$ takes at most O(C(n)) time to compute when V contains n points.

Clearly, the decomposable searching problems are O(1)-decomposable. There are a number of searching problems that are C(n)-decomposable for some $C(n) \neq O(1)$. For instance, the 3-dimensional convex hull problem is O(n)-decomposable. Most results known for decomposable searching problems, including the ones described here carry over to C(n)-decomposable searching

problems in a very easy way namely by replacing $Q_S(n)$ in the righthand sides of the query time bounds on transformed structures by $(Q_S(n) + C(n))$. Theorem 3.6. and 4.4. become respectively:

Theorem 5.2. Let S be a static data structure for a C(n)-decomposable searching problem Q. There exists a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = \begin{cases} O(Q_S(N) + C(N)) & \text{if } (Q_S(N) + C(N)) = \Omega(N^{\varepsilon}),\ \varepsilon > 0 \\ O(\log N)(Q_S(N) + C(N)) & \text{otherwise} \end{cases}$$

$$I_{S'}(N) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)P_S(N)/N & \text{otherwise} \end{cases}$$

$$S_{S'}(N) = \begin{cases} O(S_S(N)) & \text{if } S_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)S_S(N) & \text{otherwise} \end{cases}$$

Theorem 5.3. Let S be a dynamic data structure for a C(n)-decomposable searching problem Q. There exists a structure S' for solving Q in the past, such that

$$Q_{S'}(N) = \begin{cases} O(Q_S(N) + C(N)) & \text{if } (Q_S(N) + C(N)) = \Omega(N^{\varepsilon}),\ \varepsilon > 0 \\ O(\log N)(Q_S(N) + C(N)) & \text{otherwise} \end{cases}$$

$$I_{S'}(N) = \begin{cases} O(P_S(N)/N) & \text{if } P_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N).P_S(N)/N & \text{otherwise} \end{cases}$$

$$D_{S'}(N) = \begin{cases} O(I_S(N) + D_S(N)) & \text{if } I_S(N) = \Omega(N^{\varepsilon}),\ \varepsilon > 0 \\ O(\log N.I_S(N) + D_S(N)) & \text{otherwise} \end{cases}$$

$$S_{S'}(N) = \begin{cases} O(S_S(N)) & \text{if } S_S(N) = \Omega(N^{1+\varepsilon}),\ \varepsilon > 0 \\ O(\log N)S_S(N) & \text{otherwise} \end{cases}$$

Using a result of Preparata and Hong [15] on merging 3-dimensional convex figures, (see also Bentley and Shamos [3]) theorem 5.2. leads to an in-the-past structure for the 3-dimensional convex hull problem, yielding a query time of O(N) and an insertion time of $O(\log^2 N)$ while the storage required is bounded by O(N log N).

We would like to end this paper by giving a number of topics for further research.

(i) Direct solutions for solving in-the-past versions of searching problems have already been given for member searching, k^{th} element/rank searching and range searching ([11]). It seems interesting to try and

devise structures for in-the-past versions of other searching problems like nearest neighbor searching and convex hull searching.

(ii) For dynamization of decomposable searching problems whole classes of methods have been devised to obtain different trade-offs between query and update times. It seems reasonable that similar results can be obtained for the type of transformations considered in this paper.

(iii) Lowerbounds on the efficiency and storage required of the transformations remain to be proven.

(iv) In Overmars [10] another class of searching problems, called C(n)-order decomposable set problems, is defined and a general dynamization technique is described for these problems. It may be possible to give general methods to solve the in-the-past versions of these problems. Also, there might be other classes of searching problems for which general methods, for solving the corresponding in-the-past problems, exist.

(v) It is possible to extend the notion of searching in the past to performing queries over intervals of time rather than over a moment of time. Different interpretations of this can be given, for example, we want to perform the query over the elements that were present during the whole interval of time, or on those that were present at least once during the interval. Also for these in-the-past versions of searching problems, structures and general transformations can be devised.

6. References

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

[2] Bentley, J.L. and J.B. Saxe, Decomposable searching problems I: static to dynamic transformation, J. of Algorithms 1 (1980) 301-358.

[3] Bentley, J.L. and M.I. Shamos, Divide and conquer for linear expected time, Inform. Proc. Lett. 7 (1978) 87-91.

[4] Dobkin, D.P. and J.I. Munro, Efficient uses of the past, Proc. of the 21th IEEE Symp. on Foundations of Computer Science, 1980, 200-206.

[5] Gowda, I.G. and D.G. Kirkpatrick, Exploiting linear merging and extra storage in the maintenance of fully dynamic geometric data structures, Proc. of the 19th Annual Allerton Conference on Communication, Control and Computing, 1980.

[6] Kirkpatrick, D.G., Optimal search in planar subdivisions, preprint, Dept. of Computer Science, UBC, Vancouver, 1979.

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

[8] Lueker, G.S., A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems, Techn. Rep. #129, Dept. of Inform. and Computer Science, University of California, Irvine, 1978.

[9] Overmars, M.H., General methods for "all elements" and "all pairs" problems, Inform. Proc. Lett. 12 (1981) 99-102.

[10] Overmars, M.H., Dynamization of order decomposable set problems, Techn. Rep. RUU-CS-80-9, Dept. of Computer Science, University of Utrecht, 1980. (To appear in J. of Algorithms)

[11] Overmars, M.H., Searching in the past I, Techn. Rep. RUU-CS-81-7, Dept. of Computer Science, University of Utrecht, 1981.

[12] Overmars, M.H., Transforming semi-dynamic data structures into dynamic structures, to appear, 1981.

[13] Overmars, M.H. and J. van Leeuwen, Dynamization of decomposable searching problems yielding good worst-case bounds, in: P. Deussen (ed.), Theoretical Computer Science (5th GI-conf.), Lect. Notes in Comp.Sci. 104, Springer-Verlag, 1981, 224-233.

[14] Overmars, M.H. and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching problems, Techn. Rep. RUU-CS-80-10, Dept. of Computer Science, University of Utrecht, 1980. (To appear in Inform. Proc. Lett.)

[15] Preparata, F.P. and S.J. Hong, Convex hulls of finite sets of points in two and three dimensions, C. ACM 20 (1977) 87-93.

[16] Shamos, M.I., Computational geometry, Ph.D.Thesis, Yale University, 1978 (to be published).

[17] Shamos, M.I. and D. Hoey, Closest-point problems, Proc. of the 16th Annual IEEE Symp. on Foundations of Computer Science, 1976, 151-162.

[18] van Leeuwen, J. and M.H. Overmars, The art of dynamizing, Techn. Rep. RUU-CS-81-8, Dept. of Computer Science, University of Utrecht, 1981. (To appear in the Proc. of MFCS '81, Springer-Verlag, 1981.)

[19] Willard, D.E., New data structures for orthogonal queries, TR-22-78, Aiken Computation Lab., Harvard University, 1978.

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