Two models for the reconstruction problem for dynamic data structures

Michiel H.M. Smid, Leen Torenvliet
Peter van Emde Boas, Mark H. Overmars

RUU-CS-87-16
September 1987

Rijksuniversiteit Utrecht

Vakgroep Informatica

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

Two models for the reconstruction problem for dynamic data structures

Michiel H.M. Smid, Leen Torenvliet
Peter van Emde Boas, Mark H. Overmars

Technical Report RUU-CS-87-16
September 1987

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht
the Netherlands

TWO MODELS FOR THE RECONSTRUCTION PROBLEM FOR DYNAMIC DATA STRUCTURES*

by

Michiel H.M. Smid[1]
Leen Torenvliet[1]
Peter van Emde Boas[1]
Mark H. Overmars[2]

September 1987

abstract

The reconstruction problem for dynamic data structures is investigated: Given a dynamic data structure, which is entirely stored in main memory, construct a shadow administration, which is stored in the safe secondary memory, from which the original data structure can be reconstructed in case of calamity. Two realistic models of secondary memory are studied, in which the maintenance of the shadow structures are described and analyzed. In the Sequential Model, the shadow structure can be updated only by replacing the entire file by an updated one, whereas in the Indexed Sequential Model, the shadow administration is stored in a number of blocks, and it is possible to replace a block by another one. The models are illustrated with a number of examples. In both models, efficient shadow administrations can be obtained for set problems, whose answers can be merged efficiently. In the Indexed Sequential Model, it is possible to get efficient shadow structures for the data structures solving decomposable searching problems.

* The first author was supported by the Netherlands Organization for the Advancement of Pure Research (ZWO).
[1] Department of Computer Science, University of Amsterdam, Nieuwe Achtergracht 166, 1018 WV Amsterdam, The Netherlands.
[2] Department of Computer Science, University of Utrecht, P.O.Box 80.012, 3508 TA Utrecht, The Netherlands.

1. Introduction

1.1. The reconstruction problem for dynamic data structures

A large part of the research in the theory of data structures is concerned with the design of structures and algorithms solving searching problems. The efficiency of such data structures is often caused by the facts that they can be stored entirely in main memory, and that they are dynamic (i.e., it is possible to update them efficiently if elements are inserted or deleted). However, if we take the imperfectness of computer systems into account, both nice properties pass away. After a system crash, or as a result of errors in software, the contents of main memory can get lost. Another case, in which the data structure in main memory can get lost, is the regular termination of the application program which uses the structure. In case of an application which is executed on a system which is also used by other persons, the copy of the data structure in main memory will get lost between two runs of the application program. In both cases (system crash or regular termination), the data structure has to be reconstructed from the information stored in secondary memory. This information is called the shadow administration. This leads us to the following reconstruction problem:

We are given a dynamic data structure DS, which is entirely stored in main memory. The problem is to construct a shadow administration, which will be kept in secondary memory, from which the structure DS can be reconstructed. The goal is to design a shadow structure that requires little space and can be updated efficiently, such that the structure DS can be reconstructed efficiently from it.

This problem first appeared in Torenvliet and van Emde Boas [9], where the reconstruction of trie hashing functions is investigated. Solutions to the reconstruction problem might be useful in the following areas:

(i) The theory of data bases.

(ii) Computational geometry. Since in this area often data structures are used requiring more than linear space, it might be possible to improve asymptotically upon the storage requirements.

(iii) A system in which several processors at distinct times execute distinct tasks, and which communicate through message passing, might be even more sensitive for crashes than a uniprocessor system. To protect a calculation against the failure of processors, so-called checkpoints are built in on several places of the calculation. If such a checkpoint is reached, the complete state of all processors, and the interconnection pattern, is transported to secondary memory. If the system crashes, then the calculation can be continued at the last reached checkpoint. It is clear that much time and space can be saved by efficiently storing the information from each processor.

In order to be able to study, and analyze, the efficiency of shadow administrations, we have to make assumptions on how secondary memory is structured. In this paper we shall study two models of secondary memory. The first is the Sequential Model. In this model, information is stored in secondary memory as one sequential file. If we want to update this information, we have

to replace the entire file by the updated one. So in this model, after an update of the data structure, we have to replace the entire shadow administration. However, often only a small part of the shadow administration will actually have changed after an update. E.g., if our shadow structure is a balanced binary search tree, then an insertion or a deletion changes only O(log n) of the O(n) space. Therefore, in the second model, the Indexed Sequential Model, we assume that secondary memory consists of blocks. There is the ability of replacing a block by another one. Hence in this model we can maintain a shadow administration, just by replacing the actually changed blocks. Both models are realistic in practice. The first corresponds to the notion of sequential files, the second to indexed sequential files. It turns out that in both models it is possible to obtain efficient shadow administrations for data structures solving different types of searching problems. The emphasis in this paper will be on solutions for order decomposable set problems (see Overmars [5]) and decomposable searching problems (see e.g. Bentley [1]). (The Indexed Sequential Model is also used in Overmars et al. [6], where the maintenance in secondary memory of range trees is investigated.)

This paper is organized as follows. In Section 1.2, searching problems and complexity measures for their solutions are introduced. In Section 1.3, the general approach we use to design solutions to the reconstruction problem is given. We introduce complexity measures in which the performances of solutions will be expressed. In Section 2, the Sequential Model will be investigated. We give an efficient shadow administration for range trees. Also, efficient shadow structures are given for the data structures solving order decomposable set problems. In Section 3, we study the Indexed Sequential Model. Some basic methods, and efficient solutions for order decomposable set problems, are given. In Section 4, we study solutions for decomposable searching problems in the Indexed Sequential Model. We give four dynamization techniques, which transform "static" shadow administrations into "dynamic" ones. Finally, in Section 5, we give some concluding remarks.

1.2. Searching problems

In a searching problem, a question (also called a query) is asked about an object (called the query object) with respect to a set of objects. That is, if T_1, T_2 and T_3 are sets, then a searching problem is a mapping $PR : T_1 \times P(T_2) \rightarrow T_3$ (here we use the notation $P(T_2)$ to denote the power set of T_2). E.g., in the member searching problem, we are given a query object x, and a set V, and we are asked to decide whether or not x is an element of V. In this case $T_1=T_2$, $T_3=\{true, false\}$, and $PR(x,V)=(x\in V)$. Another example is the orthogonal range searching problem: We are given a set V of points in the plane, and a rectangle $([x_1:y_1],[x_2:y_2])$, and we are asked to determine all points $p=(p_1,p_2)$ in V, such that $x_1\leq p_1\leq y_1$ and $x_2\leq p_2\leq y_2$, i.e., all points of V, that are in the rectangle. Finally, in the nearest neighbor searching problem, we are given a set V of points in the plane, and a query point x, also in the plane, and we are asked to determine a point in V that is closest to x (closest with respect to the euclidean distance).

Given a set V in $P(T_2)$, a solution to the searching problem PR consists of a data structure DS, representing V, such that queries (i.e., PR(x,V) for x in T_1) can be computed efficiently. If the set V is given beforehand and does not change, then the data structure is called static. The structure is called dynamic, if it is possible to insert and delete elements. Let n=|V| be the cardinality of the set V. Then the complexity of data structure DS is given by the following functions of n: $P_{DS}(n)$, the preprocessing time required to build DS; $S_{DS}(n)$, the amount of space required to store DS; $Q_{DS}(n)$, the time required to answer a query using DS; and in case DS is a dynamic data structure, $I_{DS}(n)$, the time required to insert an element into DS; $D_{DS}(n)$, the time required to delete an element from DS. If the data structure DS is clear from the context, we just write P(n), S(n) etc. For more information about searching problems, the reader is referred to Overmars [5], and to the references given there.

1.3. The general situation

To study and analyze shadow administrations, we use the following conceptual model. It is not the best way of implementing the techniques, but it is easier to analyze and does not increase the complexity in order of magnitude. We store the following information:

(i) DS is a dynamic data structure solving the searching problem. We assume that DS is entirely stored in main memory.
(ii) SH is a shadow administration, from which the data structure DS can be reconstructed. This shadow administration is also stored in main memory.
(iii) In secondary memory, we store a copy CSH of the shadow administration SH.
(iv) Finally, there might be some extra information INF, which is used to update the shadow administration SH and its copy CSH efficiently. Since this extra information is not needed to reconstruct the data structure, and hence may get lost after a system crash, it is only stored in main memory.

Note that in practice SH often is not necessary and changes can be made immediately on CSH. The distinction between SH and CSH makes it easier to estimate time bounds.

After a system crash, the contents of main memory (i.e., DS, SH and INF) will get lost. Therefore, we transport the copy CSH of the shadow administration to main memory, and we reconstruct from this copy the data structure DS, and the structures SH and INF. In case of an insertion or a deletion, the following steps are performed:
(i) The data structure DS is updated.
(ii) The structures SH and INF are updated.
(iii) The copy CSH in secondary memory is updated.
Steps (i) and (ii) take place in main memory, which is supposed to be a random access machine with real arithmetic. Therefore, all standard operations are allowed for these two steps of the update

procedure. The complexity of it will be expressed in computing time, which is just the usual measure to express the length of a computation. In step (iii), data in secondary memory has to be updated. In order to describe this update procedure, we have to make a model of secondary memory to make clear how it is structured, and what operations are possible. Also, we have to introduce complexity measures, in order to analyze the performances.

In this paper, we shall study two models of secondary memory, the Sequential Model and the Indexed Sequential Model. In both models, step (iii) of the update procedure is performed by transporting data from core to secondary memory. The complexity of this transport process is given by two quantities. The first quantity is the number of seeks that has to be done: The information, which has to be transported, is stored in a number of parts. Now for each such part we have to do one seek. E.g., in the Sequential Model, where the copy CSH is stored as one part, one seek has to be done. In both models, we assume that just one seek is necessary to transport the entire copy CSH from core to secondary memory (in case the file in secondary memory is built) or from secondary memory to core (in case of a system crash). The second quantity is the transport time. Let $T(S(n))$ be the time required to transport an amount of $S(n)$ data from core to secondary memory, or vice versa. It is reasonable to assume that $T(S(n))=O(S(n))$. However, the constant in this estimate will be incomparable with the constants in computing time. Therefore we treat this transport time as a separate time measure.

The complexity of the additional structures (i.e., SH, CSH and INF) will be expressed by:

(i) The total amount of storage they require.

(ii) The time required to update them. This time will be split in computing time, number of seeks, and transport time.

(iii) The time required to reconstruct the data structure DS, and the structures SH and INF. Also this time will be split in three parts. Note that here the number of seeks is equal to one, and that the transport time is $O(S'(n))$, where $S'(n)$ is the amount of space required to store CSH.

To finish this section, we introduce the basic notations, which will be used in this paper.

Notations :

For a data structure DS, we shall use the usual notations $P(n)$, $S(n)$, $Q(n)$, $I(n)$ and $D(n)$ to express its performances (cf. Section 1.2).

For the additional structures SH, CSH and INF of the data structure DS:

$S'(n)$ denotes the total amount of space required by these additional structures;

$P_s(n)$, $P_t(n)$ and $P_c(n)$ denote, respectively, the number of seeks, the transport time and the computing time, required to build the additional structures;

$I_s(n)$, $I_t(n)$ and $I_c(n)$ denote the complexity of an insertion into the additional structures (again split in number of seeks, transport time and computing time);

$D_s(n)$, $D_t(n)$ and $D_c(n)$ denote the complexity of a deletion from the additional structures;

$R_s(n)$, $R_t(n)$ and $R_c(n)$ denote the complexity of reconstructing the data structure DS.

Note that $P_s(n)=1$, $P_t(n)=O(S'(n))$, $R_s(n)=1$ and $R_t(n)=O(S'(n))$.

2. The Sequential Model

2.1. Description of the model

The first model of secondary memory we study is the Sequential Model. In this model, information is stored as one sequential file. An update of this file is performed by replacing the entire file by the updated one (the old file is discarded).

An efficient shadow administration in this model should have the following properties (for the notation, see Section 1.3):

(i) The additional structures require much less space than DS does. If the space requirements of the additional structures were about S(n), then we could better take the following simple shadow administration. There are no structures SH and INF, whereas CSH is a copy of the structure DS. This shadow structure also takes O(S(n)) space. After an update, the old copy can be discarded, and a new copy is transported to secondary memory. This update procedure takes only one seek and O(S(n)) transport time. After a system crash, we just transport the copy CSH to main memory, at the cost of one seek and O(S(n)) transport time. So there is no computing time in the complexity of the additional structure.

(ii) SH and INF can be updated efficiently. Note that the update time of CSH is completely determined by the space requirement of SH.

(iii) Finally, the reconstruction computing time is much smaller than P(n), the preprocessing time of DS. If the reconstruction time were about equal to P(n), then we could better take the following shadow administration, consisting of just the elements represented by the data structure. That is, let SH (and CSH) be a list, containing the elements in sorted order. Furthermore, let INF be a balanced leaf search tree, containing the elements in its leaves, also in sorted order. Each leaf of this tree contains a pointer to the corresponding element in the list SH. This shadow administration takes only O(n) space. To reconstruct the structures after a crash, we just transport the list CSH to core, and we build from scratch the structure DS. So reconstruction takes one seek, O(n) transport time, and also O(P(n)) computing time. (Note that this computing time can be much smaller, since we have the elements in sorted order (cf. Section 2.2).) To update the additional structures after an insertion or a deletion, we first update the tree INF, in O(log n) computing time. Then, using the pointers from INF to SH, we know where the element has to be inserted or deleted. So the list SH can be updated in O(1) computing time. Then we transport a copy CSH of the resulting list SH to secondary memory in one seek and O(n) transport time.

Let DS be a dynamic data structure, requiring S(n) storage and P(n) preprocessing time. Now suppose we have additional structures SH, CSH and INF, requiring S'(n) storage, such that DS can be reconstructed in $R_C(n)$ computing time. We assume that $S'(n)=o(S(n))$ and $R_C(n)=o(P(n))$ (see (i) and (iii) above). It is reasonable to assume that $S'(n)=\Omega(n)$ (the additional structures will contain at least the elements represented by the data structure). So $S'(n)=o(S(n))$ implies $n=o(S(n))$. To reconstruct DS, we have to walk through $\Omega(S(n))$ space, and hence $R_C(n)=\Omega(S(n))$. So if

$R_C(n)=o(P(n))$, then $S(n)=o(P(n))$. Hence shadow administrations, requiring $o(S(n))$ storage, from which the data structure can be reconstructed in $o(P(n))$ computing time, might exist for data structures with $n=o(S(n))$ and $S(n)=o(P(n))$.

2.2. Range trees

In this section, we give an efficient shadow administration for range trees with a slack parameter (cf. Mehlhorn [3]).

Definition 2.1: Let m be a positive integer, and let α be a real number satisfying $2/11<\alpha\leq 1-1/2\sqrt{2}$. Let V be a set of n points in the plane. A <u>range tree T with slack parameter m</u> for set V is defined as follows: T consists of a BB[α]-tree, which contains in its leaves the elements of V, ordered according to their x-coordinates. Each node v of T, whose depth is a multiple of m, contains a BB[α]-tree, which has in its leaves the subset of V represented by node v, ordered according to their y-coordinates.

Range trees are used to solve the orthogonal range searching problem (see Section 1.2). The following theorem gives the complexity of this data structure (for a proof, the reader is referred to Mehlhorn [3]).

Theorem 2.1: A range tree with slack parameter m, representing a set of n points in the plane, has performances:

(i) $S(n)=O((n \log n) / m)$.

(ii) $P(n)=O(n \log n) + O((n \log n) / m) = O(n \log n)$. Here the first term is the time required to sort the set V to both coordinates, whereas the second term is the actual building time of the structure.

(iii) $I(n)$ and $D(n)$ are both $O(\log^2 n)$, on the average.

(iv) Using this structure, orthogonal range queries can be solved in time $O(\log^2 n \cdot 2^m/m)$.

It follows from the above theorem, that if we have in secondary memory the set V ordered according to both their x- and their y-coordinates, then we save $O(n \log n)$ computing time in reconstructing the data structure. Therefore, we take the following shadow administration for the range tree. Let SH (and CSH) consist of two lists; one list contains the elements of V ordered according to their x-coordinates, the other list contains them ordered according to their y-coordinates. Furthermore, let the structure INF consist of two balanced leaf search trees, having the elements in the lists in sorted order in their leaves. Each leaf of such a tree contains a pointer to the corresponding element in the list. Suppose element p is inserted or deleted in the set V. Then we insert or delete p in the trees of the structure INF. Using the pointers from the leaves of the trees to the lists in SH, we know the position in these lists where p has to be inserted or deleted. After the

lists in SH are updated, we transport a copy of the new structure SH to secondary memory.

Theorem 2.2: For a range tree with slack parameter m, there exists a shadow administration, with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)=1$, $I_t(n)=O(n)$, and $I_c(n)=O(\log n)$.

(iii) $D_s(n)=1$, $D_t(n)=O(n)$, and $D_c(n)=O(\log n)$.

(iv) $R_s(n)=1$, $R_t(n)=O(n)$, and $R_c(n)=O((n \log n)/m)$.

Proof : The bounds on the space requirement and the update time follow from the above discussion. The bounds on the reconstruction time follow from Theorem 2.1. ❑

Now take for example the slack parameter $m=\lceil \log \log n \rceil$. Then the range tree requires $S(n)=O((n \log n) / \log \log n)$ space. So the space requirement of the additional structures is $o(S(n))$. Also, reconstruction of the range tree requires $O((n \log n) / \log \log n)$ computing time, which is $o(P(n))$ (see the discussion at the end of Section 2.1).

2.3. Order decomposable set problems

In this section, we show that efficient shadow administrations exist for the data structures solving order decomposable set problems. A <u>set problem</u> is a mapping $PR : P(T_2) \rightarrow T_3$ (we can consider this as a searching problem $PR(x,V)$ by using a dummy query object x). An example is the <u>Voronoi diagram problem</u>: Given a set V of n points in the plane, compute the Voronoi diagram of V (cf. Preparata and Shamos [7]). We shall restrict ourselves to set problems, whose answers can be merged efficiently (cf. Overmars [5]).

Definition 2.2: A set problem $PR : P(T_2) \rightarrow T_3$ is called <u>C(n)-order decomposable</u>, if there is an order ORD on T_2, and a function $\square : T_3 \times T_3 \rightarrow T_3$, such that for each set $V=\{p_1 \leq p_2 \leq \ldots \leq p_n\}$, ordered according to ORD, and for each i, $1 \leq i < n$, we have

$$PR(\{p_1,\ldots,p_n\}) = \square(PR(\{p_1,\ldots,p_i\}),PR(\{p_{i+1},\ldots,p_n\})) ,$$

where the function $\square$ takes C(n) time to compute.

As an example, it was shown by Shamos [8], that the Voronoi diagram problem is O(n)-order decomposable, where ORD is the order according to x-coordinates. If we use the divide-and-conquer technique, we see that the answer to a C(n)-order decomposable set problem can be computed in $O(ORD(n) + F(n))$ time, where ORD(n) is the time required to order the n points according to ORD, and $F(n)=O(\Sigma_{i=0,\ldots,\log n} 2^i C(n/2^i))$ is the solution of the recurrence $F(n)=2F(n/2)+C(n)$.

Let PR be a C(n)-order decomposable set problem. We shortly recall a dynamic data structure

DS, solving PR (cf. Overmars [5]). Let V be a set of cardinality n, for which we want to solve PR. We make the following assumptions. The set V can be stored in O(n) space, and the answer PR(V) takes O(C(n)) space to store (in Overmars [5] it is shown that these assumptions are not essential). Let f(n) be a smooth integer function, such that $1 \le f(n) \le n$ (a function f(n) is called smooth if f(O(n))=O(f(n))). First we order the set $V=\{p_1 \le p_2 \le \ldots \le p_n\}$ according to ORD. Next we partition V into subsets $V_1=\{p_1,\ldots,p_{f(n)}\}$, $V_2=\{p_{f(n)+1},\ldots,p_{2f(n)}\}$,... The data structure DS consists of the following.

(i) Each set V_i is stored in a balanced binary search tree T_i. Let r_i be the root of T_i. These roots are ordered according to $r_1<r_2<r_3<\ldots$

(ii) The roots of the trees T_i are stored in an augmented BB[α] leaf search tree T. Let v be a node of T, and suppose the subtree of T with root v has $r_i, r_{i+1}, \ldots, r_j$ as its leaves. Then we store in node v the answer to the set problem PR for the set $V_i \cup V_{i+1} \cup \ldots \cup V_j$. In particular, the root of T contains the answer to PR for the entire set V. Furthermore, each node of T contains information to guide searches.

An insertion of a point p is performed as follows. We walk down tree T to find the appropriate root r_i, and we insert p in the structure T_i. Then we rebuild the answer $PR(V_i)$ and walk back to the root of T. During this walk we copy for each node we encounter, the answers stored in its left and right sons, and we merge these copies using the function □. The deletion procedure is similar. Suppose at the moment we build this structure, the set V contains n_0 elements. Then each subset V_i contains at most $f(n_0)$ elements. As soon as one set V_i contains either $1/2f(n_0)$ or $2f(n_0)$ elements (as a result of insertions and deletions), we rebuild the entire data structure. That is, we partition the set V into subsets of size f(n), where n is the cardinality of V at that moment. So we have to rebuild the data structure at most once every $\Omega(f(n))$ updates. Before we state the results, we introduce some notations.

Notations: Let C(n) and f(n) be functions, $1 \le f(n) \le n$. Then the functions C'(n), C''(n) and F(n) are defined by:

$$C'(n) = \sum_{i=0,\ldots,\log n/f(n)} 2^i\, C(n/2^i),$$
$$C''(n) = \sum_{i=0,\ldots,\log n/f(n)} C(n/2^i),$$
$$F(n) = \sum_{i=0,\ldots,\log n} 2^i\, C(n/2^i).$$

These functions satisfy:

$$C'(n) = \begin{cases} O(\, n / f(n)^{1-\varepsilon}\,) \text{ if } C(n) = O(n^{\varepsilon}) \text{ for some } 0 \le \varepsilon < 1, \\ O(\, C(n)\,) \text{ if } C(n) = \Omega(n^{1+\varepsilon}) \text{ for some } \varepsilon > 0, \\ O(\, C(n) \log n/f(n)\,) \text{ otherwise.} \end{cases}$$

$$C''(n) = \begin{cases} O(\, C(n)\,) \text{ if } C(n) = \Omega(n^{\varepsilon}) \text{ for some } \varepsilon > 0, \\ O(\, C(n) \log n/f(n)\,) \text{ otherwise.} \end{cases}$$

$F(n) = 2\,F(n/2) + C(n)$, and

$$F(n) = \begin{cases} O(n) \text{ if } C(n) = O(n^{\varepsilon}) \text{ for some } 0 \le \varepsilon < 1, \\ O(\ C(n)\ ) \text{ if } C(n) = \Omega(n^{1+\varepsilon}) \text{ for some } \varepsilon > 0, \\ O(\ C(n) \log n\ ) \text{ otherwise.} \end{cases}$$

The proof of the following theorem can be found in [5].

Theorem 2.3: Let f(n) be a smooth integer function, $1 \le f(n) \le n$. For a C(n)-order decomposable set problem, there exists a dynamic data structure DS, with performances:

(i) $S(n) = O(\ n + C'(n)\ )$.

(ii) $P(n) = O(\ n \log n + (n/f(n))\ F(f(n)) + C'(n)\ )$.

(iii) $Q(n)=O(1)$.

(iv) I(n) and D(n) are $O(\ \log n + F(f(n)) + C''(n) + P(n)/f(n)\ )$, on the average.

The shadow administration we take for DS consists of parts of the structure itself. Therefore, we do not need the structures SH and INF. The structure CSH consists of the trees T_i, and the answers $PR(V_i)$ to the set problem PR for the subsets V_i. After an update, we just transport the trees T_i and the answers $PR(V_i)$ to secondary memory. To reconstruct the original data structure, we transport the structure CSH to main memory. We assume that the file CSH is organized in such a way, that we have the roots of the trees T_i in sorted order. The only thing we have to do is to build the tree T: in every node of T, we copy the answers to PR of its sons, and we merge them.

Theorem 2.4: Let f(n) be a smooth integer function, $1 \le f(n) \le n$. For the data structure DS of Theorem 2.3, solving a C(n)-order decomposable set problem, there exists a shadow administration, with performances:

(i) $S'(n)=O(\ n + (n/f(n))\ C(f(n))\ )$.

(ii) $I_s(n)=1$ and $I_t(n)=O(\ n + (n/f(n))\ C(f(n))\ )$.

(iii) $D_s(n)=1$ and $D_t(n)=O(\ n + (n/f(n))\ C(f(n))\ )$.

(iv) $R_s(n)=1$, $R_t(n)=O(\ n + (n/f(n))\ C(f(n))\ )$ and $R_c(n)=O(\ C'(n)\ )$.

Proof : Because of our assumptions, the trees T_i together take $O((n/f(n))\ f(n))=O(n)$ space, and the answers $PR(V_i)$ take $O((n/f(n))\ C(f(n)))$ space. The bounds on the update time are trivial. The bound on the reconstruction computing time follows from our assumption that the answer PR(V) for a set of cardinality n takes O(C(n)) space to store (hence it can be copied in O(C(n)) time) and from the definition of a C(n)-order decomposable set problem. Note that the depth of the tree T is bounded by $O(\log n/f(n))$. ❑

Now consider a $\theta(n)$-order decomposable set problem. Let $f(n)=\lceil n/\log n \rceil$. By Theorem 2.3, there exists a dynamic data structure for this problem, with performances $S(n)=O(n \log \log n)$, $P(n)=O(n \log n)$, $Q(n)=O(1)$, $I(n)=O(n)$ and $D(n)=O(n)$ (the latter two on the average). Then Theorem 2.4 leads to the following

Theorem 2.5: For the data structure, solving a $\theta(n)$-order decomposable set problem, there exists a shadow administration with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)=1$ and $I_t(n)=O(n)$.

(iii) $D_s(n)=1$ and $D_t(n)=O(n)$.

(iv) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)= O(n \log \log n)$.

So again we have an example where the storage requirement of the shadow administration is $o(S(n))$, and where the reconstruction computing time is $o(P(n))$. An example of a $\theta(n)$-order decomposable set problem is the Voronoi diagram problem, mentioned in the introduction of this section. Other examples can be found in Overmars [5].

We can use these results for solving the nearest neighbor searching problem (see Section 1.2). It was shown by Kirkpatrick [2] that a data structure, requiring $O(n)$ space, can be built in $O(n)$ time from the Voronoi diagram of a set of n points, such that the nearest neighbor of a query point can be determined in $O(\log n)$ time. Since the Voronoi diagram can be built in $O(n \log n)$ time (see Preparata and Shamos [7]), it follows that there is a static data structure that solves the nearest neighbor searching problem, with performances $S(n)=O(n)$, $P(n)=O(n \log n)$ and $Q(n)=O(\log n)$. Now we put this structure, together with the structure of Theorem 2.3 (applied to the Voronoi diagram problem; take $f(n)=\lceil n/\log n \rceil$), into one data structure. Then we get the following theorem (see Overmars [5]).

Theorem 2.6: For the nearest neighbor searching problem in the plane, there exists a dynamic data structure, with performances:

(i) $S(n)=O(n \log \log n)$.

(ii) $P(n)=O(n \log n)$.

(iii) $Q(n)=O(\log n)$.

(iv) $I(n)$ and $D(n)$ are $O(n)$, on the average.

Proof : The proof follows from the above discussion. ❑

We take as a shadow administration for the data structure of Theorem 2.6 the same structure we used in Theorem 2.5, applied to the Voronoi diagram problem. After a system crash, we first reconstruct the dynamic data structure solving the Voronoi diagram problem. Then we reconstruct from the Voronoi diagram the (static) nearest neighbor searching structure. This leads to the following theorem.

Theorem 2.7: For the data structure of Theorem 2.6, solving the nearest neighbor searching problem in the plane, there exists a shadow administration, with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)=1$ and $I_t(n)=O(n)$.

(iii) $D_s(n)=1$ and $D_t(n)=O(n)$.

(iv) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)=O(n \log \log n)$.

Proof : The bounds follow from Theorem 2.5, and from the above mentioned result of Kirkpatrick. ❑

3. The Indexed Sequential Model

3.1. Description of the model

The Sequential Model has as a disadvantage that we have to replace the entire copy CSH after an update, whereas often only a small part of it actually will have changed. In practice it is possible to write parts of a file to secondary memory by using an indexed sequential file. This gives rise to the Indexed Sequential Model. In this model, the file in secondary memory is divided into blocks of some fixed size. These blocks are linked by pointers: block i contains a pointer to the physical address of block i+1. There is the ability of direct block access: It is possible to access a block directly, provided its physical address is known. To update a file, the following operations are allowed:

(i) We can replace a block by another block, or a number of (physically) consecutive blocks by at most the same number of blocks.

(ii) We can add a new block, or a number of new blocks, at the end of the file.

Let DS be a dynamic data structure, and let SH, CSH and INF be the corresponding additional structures. After an update of the data structure, the structures SH and INF are updated. Of course, the complexity of this update operation is expressed in computing time. Then the copy CSH in secondary memory is adapted, by replacing the actually changed parts by the corresponding updated parts of SH. The complexity of this operation is expressed by the number of seeks that has to be done (for each segment of consecutive blocks we transport, we have to do one seek), and by the transport time (which is the total amount of space that has to be transported). After a system crash, the structure CSH is transported to main memory. This takes one seek and $O(S_{CSH}(n))$ transport time, where $S_{CSH}(n)$ is the amount of space required by CSH. Then the data structure DS and the structures SH and INF are reconstructed. The complexity of this procedure is expressed in computing time.

The main goals we want to achieve in this model are:

(i) We want the additional structures SH, CSH and INF to require less space than DS does (see

Section 2.1).
(ii) We want the reconstruction computing time to be smaller than the preprocessing time of the data structure DS (see Section 2.1).
(iii) We want fast update algorithms for the structures SH and INF.
(iv) We want to partition the shadow administration SH into a number of parts, such that after an update just a few parts have changed. Then the copy CSH is stored in secondary memory, such that each part of the partition occupies a number of consecutive blocks. Hence after an update just a few seeks have to be done. Note that the number of seeks after an update depends on the way parts of the partition are stored in secondary memory.

Just as in Section 2, we illustrate the Indexed Sequential Model with a number of examples. It will turn out that in this model it is possible to obtain efficient shadow administrations for the data structures solving order decomposable set problems and decomposable searching problems.

3.2. Some basic solutions

Let DS be a dynamic data structure, requiring S(n) storage, and P(n) preprocessing time. We can solve the reconstruction problem by keeping in secondary memory a copy of the data structure. In that case we do not need the structures SH and INF, and CSH is a copy of DS. As mentioned earlier, we partition the structure DS into a number of parts. Then the copy CSH is stored in secondary memory by putting each part of the partition in a number of consecutive blocks. In main memory, we record for each part of the partition the address of the beginning of its copy in secondary memory. After an update of the data structure in core, the copy CSH is updated by replacing all parts of the partition that actually have changed. Obviously, the complexity of an update heavily depends on the way the structure DS is partitioned. Reconstruction of the data structure DS takes one seek and O(S(n)) transport time: we only have to transport the structure CSH to main memory. There are, however, some problems with the amount of storage used in secondary memory. When a part of the partition is changed, it has to be rewritten to secondary memory. This new part fits in the old space for it only, if the new size is not greater than the old size. But since in general the sizes of the parts depend on n, these sizes will grow when n grows. Therefore we will reserve larger slots for storing parts than is actually necessary. In this way, the slot will have room enough to store the part, even when it grows. If the part becomes too large to be stored in the slot, the entire file in secondary memory is rebuilt. Also when a part becomes too small, and hence the slot reserved for it becomes too large, we rebuild the entire file. It is shown in Overmars et al. [6], that using this technique, the storage used in secondary memory is bounded by O(S(n)), without increasing the average update costs in order of magnitude. A second problem with storage might be that the physical block size is larger than the slots we need. In that case we can pack a number of slots into one physical record in the usual way for structures in secondary memory.

As an example, in Overmars et al. [6], the partitioning of range trees (with slack parameter m=1) is investigated. We mention a result of that paper.

Theorem 3.1: A special version of a two-dimensional range tree, having the same performances as an ordinary range tree, can be maintained in secondary memory, such that an update requires 2 seeks and O(n) transport time, on the average.

A second way to solve the reconstruction problem is the following. After an update of the data structure in core, we just write the information about the update to secondary memory. That is, if element p is inserted or deleted, then we add in secondary memory, at the end of the file, element p together with a bit "inserted" or "deleted". So an update of the shadow administration takes one seek and O(1) transport time. Suppose we have a sequence of N updates. Then this shadow administration requires O(N) space. Reconstruction of the data structure requires one seek, O(N) transport time, and O(N log N + P(n)) computing time. (First we determine the n elements which are present. This takes O(N log N) time. Then we build from these elements the data structure DS in O(P(n)) time.) We can save space and reconstruction time in the above shadow administration, by rebuilding the file in secondary memory, as soon as e.g. one half of the elements have a "deleted" bit. Of course this has as a consequence that the average update time increases.

There is also a mixed solution to the reconstruction problem. Let k be a positive integer, possibly depending on n. After each k-th update, we transport a copy of the data structure DS to secondary memory. After an intermediate update, just the information about the update is transported, as described above. To reconstruct the data structure, we transport the shadow administration to main memory, and we perform the final updates.

3.3. A low storage shadow administration

Again, let DS be a dynamic data structure, representing a set V of n elements. We assume that each element of V contains a primary key from some ordered set. We take as a shadow administration this set V.

Suppose that a block in secondary memory can contain b elements. We partition the set $V=\{p_1 \leq p_2 \leq \ldots \leq p_n\}$ into subsets $V_1=\{p_1,\ldots,p_{b/2}\}$, $V_2=\{p_{b/2+1},\ldots,p_b\}$,... Each (sorted) set V_i is stored in a list L_i. The structure SH consists of a list, containing the lists L_i in sorted order. The copy CSH of this list is stored in secondary memory by putting each list L_i in a separate block. These blocks are linked by two types of pointers. First each block contains a pointer to its successor block in the file. These pointers are maintained by the operating system. Also, each block contains a pointer to its successor block according to the order of the sets V_i, i.e., the block containing the set V_i contains a pointer to the address of the block containing V_{i+1}. The pointers of this second type have to be maintained by ourselves. The structure INF is a balanced leaf search tree, containing in its leaves the elements of V in sorted order. Each leaf of this tree contains a pointer to its copy in the list SH, and the address in secondary memory of the block containing its copy.

Suppose element p is inserted into the set V. First we insert p into the tree INF. This gives us both the position in SH, and the block in secondary memory, where p has to be inserted. Next p is

inserted in the list SH. Let V_i be the subset of V into which p is inserted. There are two possibilities.

(i) After the insertion, the set V_i contains less than b elements. In this case we replace the block in secondary memory, containing the old V_i, by a block containing the updated V_i. Also, we add in INF the information about the positions of p in SH and CSH.

(ii) After the insertion, V_i contains b elements. Then we split V_i in two subsets V_{i1} and V_{i2}, both of cardinality b/2, such that the keys of the elements in V_{i1} are less than those of the elements in V_{i2}. We store the part of the list containing V_{i1} in the block containing the old V_i. The part of the list containing V_{i2} is stored in a new block at the end of the file. Then we insert a pointer in the block containing V_{i1} to the block containing V_{i2}. Also, we add a pointer from V_{i2} to V_{i+1}. Finally, information about the new positions in SH and CSH is inserted into INF.

This insertion procedure takes O(log n + b) computing time (O(log n) time for the insertion of p into INF, and O(b) time for the insertion of the new positions), at most 2 seeks, and at most 2 blocks of data transport. Note that in most cases an insertion takes only one seek and one block of data transport.

The deletion procedure is as follows. Suppose element p is deleted from the set V. First we delete p from the tree INF. This gives us the positions in SH and CSH, where p has to be deleted. Next we delete p from the list SH. Suppose p is deleted from V_i. Again there are two possibilities.

(i) After the deletion, V_i contains more than b/4 elements. Then we proceed in the same way as we did in case (i) of the insertion procedure.

(ii) After the deletion, V_i contains b/4 elements. In this case we merge V_{i-1} and V_i into a new subset V_i, and the old V_{i-1} is discarded (or we merge V_{i+1} and V_i into a new V_i). If the resulting V_i contains at least b elements, then we split it in two equal sets V_{i-1} and V_i. The part of the list containing V_{i-1} resp. V_i is stored in the block containing the old V_{i-1} resp. V_i. If the resulting V_i contains less than b elements, then we store the list containing V_i in the block containing the old V_{i-1}. Also, we insert a pointer in this block to the set V_{i+1}. In order to avoid holes in secondary memory, the block at the end of the file is moved to the block containing the old V_i. Also, we add the new address of this moved block to its predecessor block. Of course, in both cases information about the new positions in SH and CSH is inserted into the tree INF.

This deletion procedure takes O(log n + b) computing time, at most 3 seeks, and at most 3 blocks of data transport. Note that in most cases only one seek and one block of data transport is necessary.

It follows from the above, that the total amount of space required by the additional structures is O(n). After a system crash, we transport the structure CSH to main memory. This takes one seek and O(n) transport time. Then, we put the blocks in the right order (according to the order of the set V) using the pointers, and we build the data structure DS from the ordered set V.

As an example, consider the case where DS is a range tree with slack parameter m (cf. Section 2.2). We keep in secondary memory the set V, once ordered according to their x-coordinates, and once ordered according to their y-coordinates, as described above. Then we have the following theorem.

Theorem 3.2: For a range tree with slack parameter m, there exists a shadow administration with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)\leq 4$, $I_t(n)=O(b)$ and $I_c(n)=O(\log n + b)$, where b is the number of elements a block can contain.

(iii) $D_s(n)\leq 6$, $D_t(n)=O(b)$ and $D_c(n)=O(\log n + b)$.

(iv) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)=O((n \log n) / m)$.

Proof : This follows from Theorem 2.2 and the above discussion. Note that we keep in secondary memory two shadow administrations as described above. Therefore the number of seeks after an insertion resp. a deletion is at most 4 resp. 6. ❑

3.4. Order decomposable set problems

Just as in the Sequential Model, it is possible in the Indexed Sequential Model to obtain an efficient shadow administration for the data structures solving order decomposable set problems. For the definition of this type of problems, for a dynamic data structure solving them, and for the notations, see Section 2.3.

Let PR be a C(n)-order decomposable set problem, and let V be a set of cardinality n, for which we want to solve PR. Let f(n) be a smooth integer function, such that $1\leq f(n)\leq n$. Let DS be the dynamic data structure of Section 2.3, solving PR.

The shadow administration we take for DS consists of parts of the structure itself. Therefore, we do not need the structures SH and INF. The structure CSH consists of the trees T_i, and the answers $PR(V_i)$ to the set problem PR of the subsets V_i. We divide secondary memory in parts of consecutive blocks, such that each part can contain an answer $PR(V_i)$ and a tree T_i, for a set V_i of cardinality at most 2f(n). Then we store in each such part an answer $PR(V_i)$ and the corresponding tree T_i. Also, in each leaf of the tree T (which is stored in main memory; the leaves contain the roots of the trees T_i), we store the address of the corresponding structures T_i and $PR(V_i)$ in secondary memory. If after an update the data structure is not rebuilt, only one tree T_i and one answer $PR(V_i)$ will have changed and, hence, have to be transported to secondary memory. Remark that we know from the update of the data structure, which T_i and which $PR(V_i)$ are changed. Also we know the position in secondary memory where these changed structures have to be written. This leads to the following theorem.

Theorem 3.3: Let f(n) be a smooth integer function, $1 \le f(n) \le n$. For the data structure DS, solving a C(n)-order decomposable set problem, there exists a shadow administration, with performances:

(i) $S'(n) = O(\, n + (n/f(n))\, C(f(n))\,)$.

(ii) $I_s(n)=1$ and $I_t(n) = O(\, f(n) + C(f(n)) + n/f(n) + (n/(f(n))^2)\, C(f(n))\,)$, on the average.

(iii) $D_s(n)=1$ and $D_t(n) = O(\, f(n) + C(f(n)) + n/f(n) + (n/(f(n))^2)\, C(f(n))\,)$, on the average.

(iv) $R_s(n)=1$, $R_t(n)=O(\, n + (n/f(n))\, C(f(n))\,)$ and $R_c(n)=O(\, C'(n)\,)$.

Proof : The bounds on the storage and the reconstruction time follow in the same way as in Theorem 2.4. If the data structure is not rebuilt after an update, then just one tree T_i and one answer $PR(V_i)$ will have to be transported to secondary memory. So in that case the adaptation of the shadow administration in secondary memory takes one seek and $O(\, f(n) + C(f(n))\,)$ transport time. If the data structure is rebuilt, then the entire file in secondary memory has to be rebuilt. This will take one seek (see Section 1.3) and $O(n + (n/f(n))\, C(f(n)))$ transport time. However, this happens at most once every $\Omega(f(n))$ updates. This leads to the bounds on the update time. ❑

Now we consider a $\theta(n)$-order decomposable set problem (e.g. the Voronoi diagram problem; other examples can be found in Overmars [5]). Let $f(n)=\lceil n/\log n \rceil$. Then, by Theorem 2.3, there exists a dynamic data structure for this problem, with performances $S(n)=O(n \log \log n)$, $P(n)=O(n \log n)$, $Q(n)=O(1)$, $I(n)=O(n)$ and $D(n)=O(n)$ (the latter two on the average). Theorem 3.3 leads to the following theorem.

Theorem 3.4: For the data structure, solving a $\theta(n)$-order decomposable set problem, there exists a shadow administration with performances:

(i) $S'(n) = O(n)$.

(ii) $I_s(n)=1$ and $I_t(n)=O(n/\log n)$, on the average.

(iii) $D_s(n)=1$ and $D_t(n)=O(n/\log n)$, on the average.

(iii) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)=O(n \log \log n)$.

In the same way as in Theorem 2.7, these bounds also apply to e.g. the nearest neighbor searching problem in the plane.

4. Decomposable searching problems

In this section, we show how dynamization techniques can be applied to get efficient shadow administrations (in the Indexed Sequential Model) for the data structures solving decomposable searching problems. Decomposable searching problems were introduced by Bentley [1].

Definition 4.1: A searching problem $PR : T_1 \times P(T_2) \rightarrow T_3$ is called <u>decomposable</u>, if there is a function $\square: T_3 \times T_3 \rightarrow T_3$ which can be computed in constant time, such that for each partition $A \cup B$ of any subset V of T_2, and for each query object x in T_1, we have $PR(x,V) = \square(PR(x,A),PR(x,B))$.

E.g., the member searching problem is decomposable with $\square$ = <u>or</u>. Another example is the orthogonal range searching problem (see Section 1.3). This problem is decomposable with $\square = \cup$. Note that, since we require the sets A and B to be disjoint, we can take the union of PR(x,A) and PR(x,B) in constant time.

In Overmars [5], several methods are given to obtain efficient dynamic data structures for decomposable searching problems. The main idea is to partition the set of elements into a number of subsets, and then to store each subset in a static data structure. In the resulting dynamic data structure, queries are answered by querying the static structures separately, and combining the answers using the function $\square$. Insertions are performed by rebuilding some small static structures together with the inserted element.

4.1. The equal block method

One way of turning a static data structure for a decomposable searching problem into a dynamic structure is the so-called "equal block method". We will briefly describe this method here. See Overmars [5] for details.

Let DS be a (static or dynamic) data structure, solving the decomposable searching problem PR. Let SH, CSH and INF form a shadow administration for DS. Let f(n) be a smooth, non-decreasing, integer function, such that $1 \le f(n) \le n$. Let V be a set of n elements for which we want to solve the problem PR. We partition the set V into sets $V_1, V_2, ..., V_i$, such that

(1) $f(n/2) \le i \le f(2n)$, and
(2) $|V_j| \le 2n/i$, $j=1,2,...,i$.

Now we make a dynamic data structure DS' for the searching problem PR as follows. Each set V_j is stored in a structure DS. There is a dictionary DICT, which contains for each element the subset V_j it is in. Finally, the sizes of the subsets V_j are stored in a search tree. An insertion is performed by inserting the new element into the structure DS, representing the smallest subset V_j (if

the structure DS is static, a completely new structure is built out of V_j and the new element). A deletion is performed by deleting the element from the structure DS it belongs to (also this in general means that we build a new structure DS). If, after an update, constraints (1) or (2) are violated, we rebuild the entire data structure DS' with $i=f(m)$ and $|V_j| \leq \lceil m/i \rceil$, $j=1,2,\ldots,i$, where m is the cardinality of the set V at that moment. It was shown by Overmars [5], that if this rebuilding has to be done again, at least m/2 updates must have taken place. The following theorem gives the complexity of the structure DS' (the proof can be found in [5]). For the notations, the reader is referred to Section 1.3 (these notations indicate the performances of the structure DS).

Theorem 4.1: The performances of the data structure DS' are given by:

(i) The storage is bounded by $O(f(n) \cdot S(n/f(n)))$.

(ii) The query time is $O(f(n) \cdot Q(n/f(n)))$.

(iii) The average insertion time is $O(\log n + P(n)/n + I(n/f(n)))$.

(iv) The average deletion time is $O(\log n + P(n)/n + D(n/f(n)))$.

For this dynamic data structure DS', we can construct the following shadow administration: For each $j=1,2,\ldots,i$, we make shadow structures SH_j, CSH_j and INF_j, representing the set V_j. We divide secondary memory in parts of consecutive blocks, such that each part is large enough to contain a structure CSH_j for a set V_j of cardinality at most 2n/i. In each such part, we store a structure CSH_j (note that this structure itself may be partitioned into a number of parts). Also, we store in main memory for each structure SH_j the address of its copy CSH_j in secondary memory. Of course, if the structure SH_j is partitioned into a number of parts, we store for each such part the address of its copy in secondary memory. We use the notations introduced in Section 1.3, to indicate the performances of the structures SH, CSH and INF.

Theorem 4.2: For the data structure DS', there exists a shadow administration, with performances:

(i) The storage is bounded by $O(f(n) \cdot S'(n/f(n)))$.

(ii) An insertion takes $O(I_s(n/f(n)))$ seeks, $O(I_t(n/f(n)) + (f(n)/n)\, S'(n/f(n)))$ transport time and $O(I_c(n/f(n)) + (f(n)/n)\, P_c(n/f(n)))$ computing time, on the average.

(iii) A deletion takes $O(D_s(n/f(n)))$ seeks, $O(D_t(n/f(n)) + (f(n)/n)\, S'(n/f(n)))$ transport time and $O(D_c(n/f(n)) + (f(n)/n)\, P_c(n/f(n)))$ computing time, on the average.

(iv) Reconstruction takes one seek, $O(f(n) \cdot S'(n/f(n)))$ transport time and $O(f(n) \cdot R_c(n/f(n)))$ computing time.

Proof : Clearly, the total amount of space required by the additional structures is $O(i \cdot S'(2n/i))$. Since $i=\theta(f(n))$, the bound on the storage follows. An insertion of an element p is performed by inserting it into the smallest set V_j (we know this smallest set from the update of the data structure DS'). If constraints (1) and (2) are not violated, we insert p into the structures SH_j and INF_j, representing V_j, and we transport copies of the changed parts of SH_j to secondary memory, where

they replace the old ones (remark that we know the positions where these copies have to be written). This will take $O(I_s(2n/i))$ seeks, $O(I_t(2n/i))$ transport time, and $O(I_c(2n/i))$ computing time. If constraint (1) or (2) is violated (which happens at most once every $\Omega(n)$ updates), the additional structures are completely rebuilt. This rebuilding procedure takes one seek (see Section 1.3), $O(i \cdot S'(n/i))$ transport time, and $O(i \cdot P_c(n/i))$ computing time. The deletion procedure is similar, and the bounds on the performance follow in the same way. To reconstruct the data structure DS', we transport the structures CSH_j to main memory in one seek and $O(i \cdot S'(2n/i))$ transport time, and we reconstruct in $O(i \cdot R_c(2n/i))$ computing time. This completes the proof. ❑

We shall illustrate the above results with an example. Let PR be the nearest neighbor searching problem in the plane (note that this problem is indeed decomposable). As stated in Theorem 2.6, there exists a dynamic data structure DS for this problem, with performances $P(n)=O(n \log n)$, $S(n)=O(n \log \log n)$, $Q(n)=O(\log n)$, $I(n)=O(n)$, and $D(n)=O(n)$. If we apply Theorem 4.1, with $f(n) = \lceil \sqrt{n} / \sqrt{\log n} \rceil$, we get a dynamic data structure DS', with performances $S(n)=O(n \log \log n)$, $Q(n)=O(\sqrt{n} \sqrt{\log n})$, $I(n)=O(\sqrt{n} \sqrt{\log n})$, on the average, and $D(n)=O(\sqrt{n} \sqrt{\log n})$, on the average (see [5]). We remarked after Theorem 3.4, that there is a shadow administration for the structure DS with performances $S'(n)=O(n)$, $P_s(n)=1$, $P_t(n)=O(n)$, $I_s(n)=1$, $I_t(n)=O(n/\log n)$ (on the average), $D_s(n) =1$, $D_t(n)=O(n/\log n)$ (also on the average), $R_s(n)=1$, $R_t(n)=O(n)$, and $R_c(n)=O(n \log \log n)$. Applying Theorem 4.2, with the same function f(n), leads to

Theorem 4.3: For the dynamic data structure DS', solving the nearest neighbor searching problem in the plane, there exists a shadow administration, with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)=1$ and $I_t(n)=O(\sqrt{n} / \sqrt{\log n})$, on the average.

(iii) $D_s(n)=1$ and $D_t(n)=O(\sqrt{n} / \sqrt{\log n})$, on the average.

(iv) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)=O(n \log \log n)$.

As a final remark, it should be noted that the equal block method of this section also has the following important application. Consider a dynamic data structure DS, solving the decomposable searching problem PR. Suppose that the structure DS is too large to be stored in main memory. Then we split the set of points for which we want to solve the problem PR into a number of subsets, and store each such subset in a structure DS. This splitting is done in such a way that each structure DS can be stored entirely in main memory. In the resulting dynamic data structure DS', queries are answered by subsequently transporting a data structure DS to core, querying it, and transporting it back to secondary memory. The final answer to the query is obtained by combining all partial answers, using the function □. The update algorithms for DS' are similar to those of this section. That is, an element is inserted into the smallest structure DS, and an element is deleted from the structure DS it belongs to. Also in this application, if constraints (1) or (2) are violated, the entire structure is built anew.

4.2. The logarithmic method

The logarithmic method, due to Bentley [1], transforms a static data structure into a semi-dynamic structure, i.e., a data structure in which only insertions can be performed. We briefly recall the method.

Let DS be a static data structure for the decomposable searching problem PR. Let SH, CSH and INF form a shadow administration for DS. We assume that both S(n) and S'(n) are $\Omega(n)$. Also we assume that the structure SH is not partitioned into parts. The reason for this assumption will be clear later. Let V be a set of n points, for which we want to solve the problem PR. We write n in the binary number system, i.e., $n = \Sigma_{i\geq 0} a_i 2^i$, where $a_i \in \{0,1\}$. Then we partition the set V into subsets $V_0, V_1, V_2, \ldots$, such that either V_i is empty or $|V_i|=2^i$ (so $|V_i|=a_i 2^i$, $i\geq 0$). Now our semi-dynamic data structure DS' is obtained by storing each non-empty set V_i in a static structure DS. An insertion of a point p is performed as follows. Let i be the smallest index for which $a_i=0$. First we discard the structures DS containing the sets $V_0, V_1, \ldots, V_{i-1}$. Then we build a new structure DS out of $V_i := V_0 \cup V_1 \cup \ldots \cup V_{i-1} \cup \{p\}$. The complexity of this structure DS' is given in the following theorem (for a proof, the reader is referred to [1,5]). As before, S(n), Q(n) and P(n) denote the performances of the structure DS.

Theorem 4.4: The performances of the data structure DS' are given by:

(i) The storage is bounded by O(S(n)).

(ii) The query time is O(Q(n)) if $Q(n)=\Omega(n^{\varepsilon})$ for some $\varepsilon>0$, and O(Q(n) log n) otherwise.

(iii) The average insertion time is O(P(n)/n) if $P(n)=\Omega(n^{1+\varepsilon})$ for some $\varepsilon>0$, and O((P(n)/n) log n) otherwise.

The shadow administration we make for this data structure DS' looks as follows. For each non-empty set V_i we make additional structures SH_i, CSH_i and INF_i representing this set. The copies CSH_i of the structures SH_i are stored in secondary memory as one sequential file in decreasing size. That is, the file in secondary memory contains (in this order) the copy CSH_i representing the largest non-empty V_i, then the copy CSH_i representing the second largest non-empty V_i, etc. At the end of the file, the copy CSH_i representing the smallest non-empty V_i is stored. Also we store in main memory for each structure SH_i the address of its copy CSH_i in secondary memory.

Suppose we insert point p into the set V. Let i be the minimal index for which $a_i=0$. We discard the structures SH_j and INF_j for the sets $V_0, V_1, \ldots, V_{i-1}$. Let $V_i := V_0 \cup V_1 \cup \ldots \cup V_{i-1} \cup \{p\}$; $V_0 := V_1 := \ldots := V_{i-1} := \emptyset$. We build additional structures SH_i and INF_i for this new set V_i. Then we transport a copy CSH_i of the resulting structure SH_i to secondary memory. This copy is stored in the blocks containing the copies for the old sets $V_0, V_1, \ldots, V_{i-1}$ (which are stored at the end of the file), and in some extra blocks at the end of the file. This insertion procedure takes one seek, $O(S'(2^i))$ transport time, and $O(P_c(2^i))$ computing time. Since we transport a copy of the entire structure SH_i to secondary memory, there is no reason to partition it into parts. This explains why

we assumed that the shadow administration SH is not partitioned. The following theorem gives the complexity of the shadow administration for DS' (again $S'(n)$, $P_c(n)$ and $R_c(n)$ denote the complexity of the structures SH, CSH and INF).

Theorem 4.5: For the data structure DS', there exists a shadow administration, with performances:

(i) The storage is bounded by $O(S'(n))$.

(ii) An insertion takes one seek; an average transport time of $O(S'(n)/n)$ if $S'(n)=\Omega(n^{1+\varepsilon})$ for some $\varepsilon>0$, and $O((S'(n)/n) \log n)$ otherwise; and an average computing time of $O(P_c(n)/n)$ if $P_c(n)=\Omega(n^{1+\varepsilon})$ for some $\varepsilon>0$, $O(1)$ if $P_c(n)=O(n^{\varepsilon})$ for some $0<\varepsilon<1$, and $O((P_c(n)/n) \log n)$ otherwise.

(iii) Reconstruction takes one seek, $O(S'(n))$ transport time, and an average computing time of $O(R_c(n))$ if $R_c(n)=\Omega(n^{\varepsilon})$ for some $\varepsilon>0$, and $O(R_c(n) \log n)$ otherwise.

Proof : The storage required by the additional structures described above, is $O(\Sigma_{i\geq 0} a_i S'(2^i)) = O(\Sigma_{i\geq 0} a_i 2^i S'(n)/n) = O(S'(n))$, since we assumed that $S'(n)=\Omega(n)$. Clearly, reconstruction takes one seek, $O(S'(n))$ transport time, and $O(\Sigma_{i=0,\ldots,\log n} R_c(2^i))$ computing time, which proves the bounds on the reconstruction computing time. So we are left with the insertion time. Clearly, the number of seeks after an update is one. Suppose we start with an empty set V. Now consider a sequence of n insertions. Let n_i be the number of times additional structures for a set of cardinality 2^i are built. Each point is built at most once in a structure representing 2^i points (note that with an insertion, a point in V_j, $0\leq j<i$, moves to V_i, which has a higher index). Hence $2^i \cdot n_i \leq n$. So the total transport time required for these n insertions is $O(\Sigma_{i\geq 0} n_i S'(2^i)) = O(n \Sigma_{i=0,\ldots,\log n} S'(2^i)/2^i)$. From this it is easy to prove the bound for the average transport time required to perform an insertion. In the same way, the average computing time for an insertion is bounded by $O(\Sigma_{i=0,\ldots,\log n} P_c(2^i)/2^i)$. This completes the proof. ❑

As an illustration of the logarithmic method, let PR be the orthogonal range searching problem in 2-dimensional space. In Theorem 2.1, we saw that there is a data structure DS (a range tree with slack parameter m) for this problem with performances $P(n)=O(n \log n)$, $S(n)=O((n \log n) / m)$, and $Q(n)=O(\log^2 n \cdot 2^m/m)$. Now apply Theorem 4.4. Then we get a semi-dynamic data structure DS', with performances $S(n)=O((n \log n) / m)$, $Q(n)=O(\log^3 n \cdot 2^m/m)$, and $I(n)=O(\log^2 n)$ on the average. The shadow administration SH for the data structure DS consists of two arrays: one array contains the points represented by DS ordered according to their x-coordinates, the other contains these points ordered according to their y-coordinates. The copy CSH of the structure SH is stored in secondary memory as one part. We do not use a structure INF. The complexity of these additional structures is given by: $S'(n)=O(n)$, $P_s(n)=1$, $P_t(n)=O(n)$, $R_s(n)=1$, $R_t(n)=O(n)$, $R_c(n)=O((n \log n) / m)$. Note that if the data structure DS is built, we first sort the points to both coordinates, and store them in two arrays. Therefore there is no preprocessing computing time for the additional structures. Furthermore, the bound for $R_c(n)$ follows from Theorem 2.1 (see also

Theorem 2.2). Now Theorem 4.5 leads to

Theorem 4.6: For the semi-dynamic data structure DS', solving the range searching problem in the plane, there exists a shadow administration, with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)=1$ and $I_t(n)=O(\log n)$ on the average.

(iii) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)=O((n \log n) / m)$.

Note that in this example, the data structure DS' is less efficient than the dynamic version of it (cf. Theorem 2.1): the query time increases by a factor log n. Also, no deletions are possible. However, the gain of the logarithmic method here is in the complexity of the additional structures: an insertion requires only one seek, O(log n) transport time on the average, and no computing time.

4.3. A general, optimal insertion method

We will now describe a shadow administration for an optimal dynamization technique for decomposable searching problems. The dynamization technique is due to Mehlhorn and Overmars [4] (see also [5]). The technique is optimal in the sense that it gives optimal trade-offs between query time and insertion time.

Again, let DS be a static data structure for the decomposable searching problem PR. Let SH, CSH and INF form a shadow administration for DS. For the same reason as in Section 4.2, we assume that the structure SH is not partitioned into parts. Also, we assume that both S(n) and S'(n) are $\Omega(n)$. We shall give two methods of dynamization, Method A and Method B.

Method A.

Let g(n) be a smooth, non-decreasing, positive function, such that $g(n)=O(\log n)$. Let V be a set of n points, for which we want to solve PR. Let $b = \lceil n^{1/g(n)} \rceil$. Since $g(n)=O(\log n)$, we have $b \geq 2$. Although b depends on n, we fix its value over the range of a number of insertions. First we write n in the b-ary number system, i.e., we write $n = \Sigma_{i \geq 0} a_i b^i$, where $a_i \in \{0,1,\ldots,b-1\}$. Then we partition the set V into sets $V_0, V_1, V_2, \ldots$, such that $|V_i| = a_i b^i$. Each non-empty set V_i is stored in a static structure DS. The resulting data structure DS' is semi-dynamic, and its performances are given by the following theorem (the proof can be found in [5]).

Theorem 4.7: The data structure DS' has performances:

(i) The storage is bounded by O(S(n)).

(ii) The query time is $O(g(n) \cdot Q(n))$.

(iii) The average insertion time is $O(g(n) \cdot n^{1/g(n)} \cdot P(n)/n)$.

We make the following shadow administration for the data structure DS'. For each non-empty set V_i, we build additional structures SH_i and INF_i. Copies CSH_i of the structures SH_i are stored in secondary memory in decreasing size, just as in the preceding section. In main memory, we store for each structure SH_i the address of its copy in secondary memory.

Now suppose point p is inserted into the set V. Then we determine the minimal index i, such that $a_i<b-1$. Next, the additional structures SH_j and INF_j for the sets $V_0, V_1,..., V_i$ are discarded. Let $V_i := V_0 \cup V_1 \cup ... \cup V_i \cup \{p\}$; $V_0:=V_1:=...:=V_{i-1}:=\emptyset$. We build additional structures SH_i and INF_i for the new set V_i. Finally, a copy CSH_i of this new structure SH_i is stored in secondary memory in the blocks containing the copies for the old sets $V_0, V_1,..., V_i$ (which are stored at the end of the file), and in some new blocks at the end of the file. This procedure takes one seek, $O(S'((a_i+1)b^i))$ transport time and $O(P_c((a_i+1)b^i))$ computing time.

Suppose at the moment the data structure DS' is built, the set V contains m points, so the value of $b = \lceil m^{1/g(m)} \rceil$. In order to keep the complexity of the structures balanced, we completely rebuild them after m insertions. That is, we take $b = \lceil n^{1/g(n)} \rceil$, where n is the cardinality of V at that moment (so n=2m), and we partition the set V again, as described before. This rebuilding operation of the additional structures takes one seek, $O(\Sigma_{i\geq 0} S'(a_i b^i)) = O(S'(n))$ transport time (since we assumed that $S'(n)=\Omega(n)$), and $O(\Sigma_{i\geq 0} P_c(a_i b^i))$ computing time, where $n = \Sigma_{i\geq 0} a_i b^i$, $a_i \in \{0,1,...,b-1\}$, $b = \lceil n^{1/g(n)} \rceil$.

For the sake of simplicity, we assume that $P_c(n)=0$, i.e., in order to build the additional structures, no computing time is required.

Theorem 4.8: Let $n = \Sigma_{i\geq 0} a_i b^i$, $a_i \in \{0,1,...,b-1\}$, $b = \lceil n^{1/g(n)} \rceil$. For the data structure DS', there exists a shadow administration, with performances:

(i) The storage is bounded by $O(S'(n))$.

(ii) An insertion takes one seek and $O((S'(n)/n) \cdot g(n) \cdot n^{1/g(n)})$ transport time, on the average.

(iii) Reconstruction takes one seek, $O(S'(n))$ transport time, and $O(\Sigma_{i\geq 0} R_c(a_i b^i))$ computing time.

Proof : The storage required by the additional structures described above is $O(\Sigma_{i\geq 0} S'(a_i b^i))= O(S'(n))$, since $S'(n)=\Omega(n)$. The bounds for the reconstruction time are trivial. So it remains to show the bounds for the insertion time. Clearly, the number of seeks after an update is one. Suppose we start with an empty set V. Consider a sequence of n insertions. The rebalancing operation is performed after insertion $1,2,2^2,2^3$, etc. Let m_k be the size of the set V_i which is made after the k-th insertion. Then, the average transport time after an insertion is

$$O(1/n [\Sigma_{i=0,...,\log n} S'(2^i) + \Sigma_{k=1,...,n} S'(m_k)]) =$$
$$O(1/n [(S'(n)/n) \Sigma_{i=0,...,\log n} 2^i + (S'(n)/n) \Sigma_{k=1,...,n} m_k]) =$$
$$O(S'(n)/n + (S'(n)/n^2) \Sigma_{k=1,...,n} m_k).$$

We shall derive an upper bound for $\Sigma_{k=1,...,n} m_k$. Therefore we look what happens between insertion h and 2h. Then the value of b is $\lceil h^{1/g(h)} \rceil$. According to [5], the number of non-empty

sets V_i is at most $c \cdot g(h)$, for some constant c, independent of h. Observe that each point is inserted at most b times in a (fixed) set V_i. Also, each set V_i contains at most 2h points. Hence:

$$\Sigma_{k=1,\ldots,n}\, m_k \leq$$
$$\Sigma_{i=0,\ldots,\log n}\, c \cdot g(\exp(i)) \cdot \lceil (\exp(i))^{1/g(\exp(i))} \rceil \cdot 2 \cdot \exp(i) = \quad \{\text{here } \exp(i) = 2^i\}$$
$$O(\Sigma_{i=0,\ldots,\log n}\, g(n) \cdot 2^{(\log n)/g(n)} \cdot 2^i) =$$
$$O(n \cdot g(n) \cdot n^{1/g(n)}).$$

Here we have used the fact that the function $(\log n)/g(n)$ is non-decreasing, since $g(n)=O(\log n)$. It follows that the average transport time after an insertion is

$$O(S'(n)/n + (S'(n)/n) \cdot g(n) \cdot n^{1/g(n)}) = O((S'(n)/n) \cdot g(n) \cdot n^{1/g(n)}). \quad \square$$

As an illustration, let PR be the orthogonal range searching problem in the plane. Let DS be a range tree with slack parameter m for solving PR (cf. Section 2.2). This data structure has performances $P(n)=O(n \log n)$, $S(n)=O((n \log n) / m)$, $Q(n)=O(\log^2 n \cdot 2^m/m)$. Let g(n) be a smooth, non-decreasing, positive function, such that $g(n)=O(\log n)$. Now apply Theorem 4.7. Then we get a semi-dynamic data structure DS', with performances $S(n)=O((n \log n) / m)$, $Q(n)= O(g(n) \cdot \log^2 n \cdot 2^m/m)$, and $I(n)=O(g(n) \cdot n^{1/g(n)} \cdot \log n)$ on the average. The shadow structures we take for the data structure DS are the same as in Section 4.2. That is, we keep in secondary memory the points represented by DS, ordered according to both coordinates. The complexity of this shadow administration is given by $S'(n)=O(n)$, $P_s(n)=1$, $P_t(n)=O(n)$, $R_s(n)=1$, $R_t(n)=O(n)$, $R_c(n)=O((n \log n) / m)$. Note that here $P_c(n)=0$. Then applying Theorem 4.8 leads to

Theorem 4.9: For the data structure DS', solving the range searching problem in the plane, there exists a shadow administration, with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)=1$ and $I_t(n)=O(g(n) \cdot n^{1/g(n)})$, on the average.

(iii) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)=O((n \log n) / m)$.

Method B.

Again, let g(n) be a smooth, non-decreasing, positive function, such that $g(n)=O(\log n)$. Let V be a set of n points, for which we want to solve the problem PR. Let $b = \lceil n^{1/g(n)} \rceil$. Also in this case, we fix the value of b over the range of a number of insertions. Write n in the b-ary number system, $n = \Sigma_{i \geq 0}\, a_i b^i$, $a_i \in \{0,1,\ldots,b-1\}$. Then partition the set V into subsets V_{ij}, $i \geq 0$, $1 \leq j \leq a_i$, such that $|V_{ij}|=b^i$. We store each set V_{ij} in a static data structure DS. The performances of the resulting semi-dynamic data structure DS' are given by the following theorem (cf. [5]).

Theorem 4.10: The data structure DS' has performances:

(i) The storage is bounded by O(S(n)).

(ii) The query time is $O(g(n) \cdot n^{1/g(n)} \cdot Q(n))$.

(iii) The average insertion time is $O(g(n) \cdot P(n)/n)$.

The shadow administration for the data structure DS' will be clear. For each set V_{ij}, we build additional structures SH_{ij} and INF_{ij}. Copies CSH_{ij} of the structures SH_{ij} are stored in secondary memory in decreasing size. Equal sized copies (i.e., copies representing sets V_{ij} with the same value for i) are stored in increasing order with respect to the second index. For example, suppose $b= 4$, $n = 59 = 3\cdot b^2 + 2\cdot b^1 + 3\cdot b^0$. Then the file in secondary memory contains the copies CSH_{ij} in the order $CSH_{21}, CSH_{22}, CSH_{23}, CSH_{11}, CSH_{12}, CSH_{01}, CSH_{02}, CSH_{03}$. Also, we store in core for each structure SH_{ij}, the address of its copy in secondary memory.

An insertion of point p is performed as follows. Let i be the minimal index, for which $a_i<b-1$. Let $j = a_i+1$. We discard the additional structures SH_{hk} and INF_{hk} for the sets V_{hk}, $0\le h\le i-1$, $1\le k\le b-1$. Let $V_{ij} := \cup_{h=0,\dots,i-1} \cup_{k=1,\dots,b-1} V_{hk} \cup \{p\}$; $V_{hk} := \emptyset$ for $0\le h\le i-1$, $1\le k\le b-1$. Note that $|V_{ij}| = 1 + \Sigma_{h=0,\dots,i-1} \Sigma_{k=1,\dots,b-1} b^h = b^i$. Then we build additional structures SH_{ij} and INF_{ij} for the new set V_{ij}. A copy CSH_{ij} of this new structure SH_{ij} is stored in secondary memory in the blocks containing the copies for the old sets V_{hk}, $0\le h\le i-1$, $1\le k\le b-1$ (which are stored at the end of the file), and in some new blocks at the end of the file. This procedure takes one seek, $O(S'(b^i))$ transport time, and $O(P_c(b^i))$ computing time.

In order to keep the complexity of the structures balanced, we rebalance in exactly the same way as in Method A. Such a rebalancing operation takes one seek, $O(\Sigma_{i\ge0} a_i S'(b^i)) = O(S'(n))$ transport time and $O(\Sigma_{i\ge0} a_i P_c(b^i))$ computing time, where $n = \Sigma_{i\ge0} a_i b^i$, $a_i\in \{0,1,\dots,b-1\}$, $b = \lceil n^{1/g(n)}\rceil$.

Just as in Method A, we assume for simplicity, that $P_c(n)=0$.

Theorem 4.11: Let $n = \Sigma_{i\ge0} a_i b^i$, $a_i\in \{0,1,\dots,b-1\}$, $b = \lceil n^{1/g(n)}\rceil$. For the data structure DS', there exists a shadow administration, with performances:

(i) The storage is bounded by $O(S'(n))$.

(ii) An insertion takes one seek and $O(g(n)\cdot S'(n)/n)$ transport time, on the average.

(iii) Reconstruction takes one seek, $O(S'(n))$ transport time, and $O(\Sigma_{i\ge0} a_i R_c(b^i))$ computing time.

Proof : The storage required by the additional structures described above is $O(S'(n))$, since we assumed $S'(n)$ to be at least linear. The bounds for the reconstruction time are trivial. So we are left with the insertion time. It is obvious that the number of seeks after an update is one. Suppose we start with an empty set V. Consider a sequence of n insertions. The entire file in secondary memory is rebalanced after insertion $1,2,2^2,2^3$, etc. Let m_k be the size of the set V_{ij} which is made after the k-th insertion. Then the average transport time after an insertion is

$$O(\ 1/n\ [\ \Sigma_{i=0,\dots,\log n} S'(2^i) + \Sigma_{k=1,\dots,n} S'(m_k)\ ]\ ) =$$
$$O(\ S'(n)/n + (S'(n)/n^2)\ \Sigma_{k=1,\dots,n} m_k\ ).$$

Again, we derive an upper bound for $\Sigma_{k=1,\dots,n} m_k$. Look what happens between update h and 2h. Then $b = \lceil h^{1/g(h)}\rceil$. According to Overmars [5], the number of indices i, for which a non-empty set V_{ij} exists, is at most $c\cdot g(h)$, for some constant c, independent of h. Furthermore, each point is inserted at most once into a set of cardinality b^j. Finally, each set V_{ij} contains at most 2h points.

Therefore $\Sigma_{k=1,\ldots,n} m_k \leq \Sigma_{i=0,\ldots,\log n} c \cdot g(2^i) \cdot 2 \cdot 2^i = O(g(n) \Sigma_{i=0,\ldots,\log n} 2^i) = O(n \cdot g(n))$. It follows that the average transport time after an insertion is $O(S'(n)/n + (S'(n)/n^2) \cdot n \cdot g(n)) = O(g(n) \cdot S'(n)/n)$. ❑

As an example, let DS be a range tree with slack parameter m, solving the orthogonal range searching problem in the plane. Let g(n) be a smooth, non-decreasing, positive function, such that $g(n)=O(\log n)$. Then Theorem 4.10 gives us a semi-dynamic data structure DS', with performances $S(n)=O((n \log n) / m)$, $Q(n)=O(g(n) \cdot n^{1/g(n)} \cdot \log^2 n \cdot 2^m/m)$, and $I(n)=O(g(n) \cdot \log n)$ on the average. Again we take the same additional structures for the data structure DS as in Section 4.2. Then Theorem 4.11 leads to

Theorem 4.12: For the data structure DS', solving the range searching problem in the plane, there exists a shadow administration, with performances:

(i) $S'(n)=O(n)$.

(ii) $I_s(n)=1$ and $I_t(n)=O(g(n))$, on the average.

(iii) $R_s(n)=1$, $R_t(n)=O(n)$ and $R_c(n)=O((n \log n) / m)$.

5. Concluding remarks

We have studied the reconstruction problem for dynamic data structures: Given a dynamic data structure solving some searching problem, design a shadow administration, to be stored in secondary memory, from which the original data structure can be reconstructed in case of a system crash. We have given two realistic models of secondary memory in which the maintenance of shadow administrations is described, and we have introduced complexity measures to express the complexity of the shadow structures. In the Sequential Model, the information in secondary memory is stored as one sequential file. This file can be updated by replacing the entire file by the updated one. In the second model, the Indexed Sequential Model, the file in secondary memory is divided into blocks. The shadow administration is partitioned into parts, and each part is stored in a number of consecutive blocks. After an update of the data structure in main memory, the shadow administration is updated by adapting the parts that actually have changed. The models are illustrated with a number of examples, including some general techniques that can be used for large classes of searching problems. It turns out that in both models efficient shadow administrations can be obtained for the data structures solving order decomposable set problems. Also, in the Indexed Sequential Model, we can design efficient shadow administrations for the structures solving decomposable searching problems.

There remain several interesting directions for future research. First it would be interesting to have other examples of data structures for which efficient shadow administrations exist. We noted already at the end of Section 2.1, for which data structures efficient shadow structures might exist

in the Sequential Model. Of course, also in the Indexed Sequential Model, other examples are wanted. In this model, the efficiency also depends on the way the shadow structure is partitioned and stored in secondary memory. So it would be interesting to search for efficient partitions of data structures and shadow structures (see Overmars et al. [6], where the partitioning of range trees is investigated), and for efficient ways to store parts of the partition in secondary memory (as we did in Section 4).

Another interesting direction is the investigation of other models of secondary memory. An example of such a model is the Random Access Model: In this model, also secondary memory is assumed to be a random access machine. Hence, in secondary memory, the same operations as in main memory are possible, although the running times will be slower. With the rise of the RAM-disks, this model will be realistic in the near future. Another example is the Write Once Model, where each position in secondary memory can be used only once. Also this model will be important with the rise of the optical disk. Both models are currently being investigated.

References

1. J.L. Bentley. Decomposable Searching Problems. Inform. Proc. Lett. **8** (1979), pp. 244-251.
2. D.G. Kirkpatrick. Optimal Search in Planar Subdivisions. SIAM J. Computing **12** (1983), pp. 28-35.
3. K. Mehlhorn. Data Structures and Algorithms, Volume 3: Multi-dimensional Searching and Computational Geometry. Springer Verlag, 1984.
4. K. Mehlhorn and M.H. Overmars. Optimal Dynamization of Decomposable Searching Problems. Inform. Proc. Lett. **12** (1981), pp. 93-98.
5. M.H. Overmars. The Design of Dynamic Data Structures. Springer Lecture Notes in Computer Science, Volume 156, Springer Verlag, 1983.
6. M.H. Overmars, M.H.M. Smid, M.T. de Berg and M.J. van Kreveld. Maintaining Range Trees in Secondary Memory Part I: Partitions. To appear.
7. F.P. Preparata and M.I. Shamos. Computational Geometry, an Introduction. Springer Verlag, 1985.
8. M.I. Shamos. Computational Geometry. Ph.D.Thesis, Dept.of Computer Science, Yale University, 1978.
9. L. Torenvliet and P. van Emde Boas. The Reconstruction and Optimization of Trie Hashing Functions. Proc.9th International Conf. on Very Large Databases (1983), pp. 142-156.