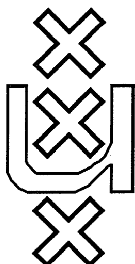
Institute for Language, Logic and Information

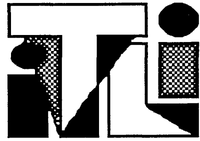
MULTIPLE REPRESENTATIONS OF DYNAMIC DATA STRUCTURES

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

ITLI Prepublication Series
for Computation and Complexity Theory CT-88-08



University of Amsterdam



Institute for Language, Logic and Information
Instituut voor Taal, Logica en Informatie

MULTIPLE REPRESENTATIONS OF DYNAMIC DATA STRUCTURES

Michiel H.M. Smid
Centre for Mathematics and Computer Science, Amsterdam
Mark H. Overmars
Department of Mathematics and Computer Science, University of Utrecht
Leen Torenvliet Peter van Emde Boas
Department of Mathematics and Computer Science, University of Amsterdam

Received October 1988

Correspondence to:

Faculteit der Wiskunde en Informatica
(Department of Mathematics and Computer Science)
Roetersstraat 15
1018WB Amsterdam

or

Faculteit der Wijsbegeerte
(Department of Philosophy)
Grimburgwal 10
1012GA Amsterdam

Multiple Representations of Dynamic Data Structures

Michiel H.M. Smid* Mark H. Overmars† Leen Torenvliet‡
Peter van Emde Boas‡

October 24, 1988

Abstract

We investigate two versions of the problem of maintaining multiple representations of dynamic data structures. In the first version, we maintain copies of the same data structure among a number of processors. In the second version, which is dual to the first one, we maintain a shadow administration in secondary memory, that contains information for the data structure in main memory to be reconstructed after e.g. a system crash. For both versions, we give a general and realistic model to describe solutions. We give general solutions that are especially applicable to data structures in which an update changes only a small part.

1 Introduction

The design of efficient data structures for solving searching problems is an important part of algorithm design. Many types of data structures exist, storing different types of objects and allowing for different types of queries. Data structures and searching problems have been studied in great detail and many properties and general techniques have been found. For example, general techniques exist for turning static data structures that do not allow for insertions and deletions of objects into dynamic structures that do allow for such operations (see [4]).

In most studies it is assumed that the data structure is stored only once in the main memory of a computer and that all operations are performed on this

*Bureau SION, Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. This author was supported by the Netherlands Organization for Scientific Research (NWO).

†Department of Computer Science, University of Utrecht, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands.

‡Departments of Mathematics and Computer Science, University of Amsterdam, Nieuwe Achtergracht 166, 1018 WV Amsterdam, The Netherlands.

one structure. In many situations, however, we store the structure more than once—possibly on different storage media—and have a *multiple representation* of the data structure.

Consider for example the following case. When we have a network of processors, each having its own memory, there are situations in which each processor holds its own copy of a particular data structure. Changes to the data structure have to be made in all copies. Therefore, we dedicate one processor the task of maintaining the data structure and broadcasting the actual changes to the other processors. Here we have one data structure that should allow for updates, and a set of other structures that answer queries. Of course, the query data structures must be structured in such a way that they can perform updates, but they get the update in a 'preprocessed' way that might be easier to handle. The structure that performs the updates will be called the *central structure*. The other structures are the *client structures*. Note that since the central and client structures do not have to support the same operations, they might be structured in a different way.

As another example, consider the following cases. If a data structure is stored only in the main memory of a computer, all information will get lost after system or program errors. Also, after the regular termination of an application program that uses a data structure, the copy of this structure that is stored in main memory can get lost.

In these cases we again represent the data more than once. That is, besides the data structure in main memory, we also store information in secondary memory from which it is possible to reconstruct the original data structure. This shadow administration does not have to support the same operations as the main structure. Only insertions and deletions have to be performed, whereas on the main structure also queries have to be carried out. Furthermore, we only require that the shadow administration contains information to make it possible to reconstruct the main structure. Therefore, this shadow administration might be structured in a different way than the original data structure itself.

In this paper, the two problems just mentioned will be studied. We shall give easily analyzable models in which solutions will be described. We give general techniques to design efficient solutions that are widely applicable. For specific solutions we refer the reader to [12,13], where efficient solutions for certain classes of searching problems are given; to [14], where a shadow administration for trie hashing functions is given; to [5,6,8,11], where the problem of maintaining range trees in secondary memory is studied; and to [9], where the union-find problem is investigated.

Clearly, both problems are related to each other. In both cases there is one structure on which the updates are performed. After this update, the other structures that are stored on other media are updated. This is done by transporting data to these other structures. The actual update procedure for the other struc-

tures is somewhat different for both problems. A shadow administration is stored in secondary memory, which is divided into blocks. The only operation allowed here is to replace a block by another one. So in secondary memory no computing is possible. The client structures, however, are stored in a random access machine, on which computing is possible. This makes it possible to replace much smaller pieces of information than just blocks of some predetermined size.

The problems are 'dual' to each other: In the first problem, there is a central structure on which only updates are performed. After this central structure is adapted, we transport data to the client structures that makes it possible to update them. These client structures are also used for answering queries. In the second problem there is a main structure on which queries and updates are performed. After an update has been carried out in the main structure, information is transported to secondary memory, and the shadow structure is updated. In this shadow structure no queries are performed.

Efficient solutions to the problems have applications in the following areas:

- The theory of databases.
- Computational geometry. Since in this area often data structures are used that require more than linear space, it might be possible to improve upon the storage requirements.
- Storing dynamic data structures in write-once memories. The results to the problem of designing efficient client structures give insight in which parts of data structures are actually changed when performing updates.
- Multiprocessing. A system in which several processors execute distinct tasks, and communicate through message passing, might be even more sensitive to crashes than a uni-processor system. To protect a calculation against failure of processors, checkpoints are built in on several places of the calculation. If a checkpoint is reached, the state of all processors, and the interconnection pattern, is transported to secondary memory. If the system crashes, the calculation can be continued from the last reached checkpoint.
- Paging data structures. Techniques to maintain shadow administrations in secondary memory can sometimes be used in cases where the data structure does not fit in main memory and, hence, has to be stored in secondary memory. For an example, see [5,6].

The paper is organized as follows. In Section 2, we introduce a general model in which solutions to the problem of designing efficient client structures will be described. A general technique is given that is particularly useful for data structures in which the size of the changes in an update is much smaller than the time

needed to find these changes. In Section 3 we give the model we use to describe solutions for the reconstruction problem. Again, a general technique is given that is useful in cases where an update changes only a small part of the shadow administration. In Section 4, we give a class of problems for which both versions of the multiple representation problem have efficient solutions. In Section 5 we give some concluding remarks.

2 Maintaining data structures in a network

2.1 The model

We first introduce the general model in which solutions to the problem of designing central and client data structures will be described.

1. There is a network of processors, the *clients*, each having its own memory. Each client contains a copy of a data structure DS, the *client structure*, and uses it to solve queries.

2. One of the processors contains a *central structure* DS'.

We assume that all processors are random access machines. An update is performed as follows. We first update the central structure DS'. During this update we obtain information making it possible to update the client structures efficiently. Then we send this information to the clients. Using the received information each client adapts its structure DS.

The complexity of the client structure DS is expressed by its size, query time and update time. This update time is split in *transport time*, which is proportional to the amount of data that is transported to the client structure, and *computing time*, which is the time the client needs to perform the update, using the received information. Note that this computing time is at least proportional to the transport time, since the client has to write the received information somewhere. (We assume that the central processor does not have direct memory access (DMA).)

The complexity of the central structure DS' is given by the usual measures, i.e. the size of the structure and the update time. (There is no query time, because in the central structure no queries are performed.)

2.2 A general technique

We will sketch a technique that is applicable to an arbitrary structure.

Let DS be the client structure, and let DS' be the corresponding central structure. Denote the size of DS (resp. DS') by $S(n)$ (resp. $S'(n)$). We assume that DS is a *substructure* of DS'. That is, DS is a part of DS', containing enough

information such that queries can be solved fast. Let UI be that part of DS' that is not in DS. (Here UI stands for update information.) For example, if DS' is a balanced binary tree, we can take for DS this tree without the balance information of the nodes, and for UI the balance information. (See the examples below.) Let $C(n)$ denote the amount of data that is changed in the client structure DS in an update.

We shall implement this multiple representation, such that an update of the client structure requires $O(C(n))$ transport and computing time. Also the other performances of the structures remain asymptotically the same.

Our processors are random access machines, the memories of which are modeled as arrays. Each entry in the array has a unique index. Data structures are composed of 'indivisible pieces of information' of constant size, such as pointers, integers, etc. Each such indivisible piece will be stored in one array entry.

In the central computer, we store the structure DS' such that DS and UI are in non-interfering parts of the memory. The client structure DS is stored in the client's memory, such that each 'indivisible piece of information' is located in the same position as its corresponding piece in the central memory. Note that data structures contain pointers, which we consider to be indices of array entries. By storing pieces of DS in each memory in the same positions, these pointers indeed 'point' to the correct object.

Suppose we want to perform an update. Then we first update the central structure. Next we send to each client the indices of all entries that are changed, together with the new contents of these entries. Using this information, each client structure is adapted. Since the indices of the entries that have to be changed in the client structure are known, it can be updated in time proportional to the number of changed entries. So the update of each client structure takes $O(C(n))$ transport and computing time. Note that the memory management of the client computers is arranged by the central structure. Clearly, at each moment the client structure is up to date and, hence, can be used to answer queries.

Let us give an example of this technique. Suppose we want to solve the member searching problem. An efficient data structure for this problem is the balanced binary search tree. Since balance information is only used for performing updates, the client versions of the tree do not need this balance information: Updates are first performed in the central structure. Then the nodes where rotations—and what kind of rotations—have to be performed are known.

Now take the binary tree from the class of αBB-trees; see Olivié [3]. These trees can be maintained by performing at most three rotations in the worst case. Hence in the client structure, an update changes only a constant amount of data. (In the central structure the balance information in the nodes has to adapted after an update, hence in this structure $O(\log n)$ data will change.)

Our technique gives client structures, in which updates can be performed

in constant transport and computing time, whereas the central structure needs $O(\log n)$ time. Using this result we can define a class of *range trees* [1], such that the corresponding client trees can be updated in $O((\log n)^{d-1})$ transport and computing time, whereas the central tree needs $O((\log n)^d)$ time. (d is the dimension of the stored points.)

3 The reconstruction problem

3.1 The model

For the reconstruction problem, i.e. the problem of designing a shadow administration for a given data structure, we use the following conceptual model.

1. *DS* is a dynamic data structure, stored in main memory.
2. *SH* is a *shadow administration*—stored in main memory—from which *DS* can be reconstructed.
3. In secondary memory, we store a copy *CSH* of *SH*.
4. There might be some extra information *INF* that is used to update *SH* and *CSH* efficiently. Since this information is not needed to reconstruct the structures, 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.

An update is performed as follows. First the structures *DS*, *SH* and *INF* are updated. Then *CSH* in secondary memory is updated.

The updates of *DS*, *SH* and *INF* take place in main memory, which is a random access machine. The complexity is expressed in *computing time.*

To update *CSH*, data in secondary memory has to be updated. We assume that the file in secondary memory is divided into blocks of some fixed size. Each block has a unique address. We can access a block directly, provided its address is known. To update a file, we can replace a number of (physically) consecutive blocks by at most the same number of blocks. Also, we can add a number of new blocks, at the end of the file.

The copy *CSH* is updated by transporting data from main memory to secondary memory. More precisely, *CSH* is updated, by replacing all blocks that have to be changed by the corresponding updated parts of *SH*. The complexity of this transport process is given by the number of *disk accesses* that has to be done—for each segment of consecutive blocks we transport, we have to do one disk access—and by the *transport time*, which is proportional to the total amount of

data that is transported. Remark that the number of disk accesses depends on the way *CSH* is stored in secondary memory.

After a system crash, the contents of main memory will get lost. Therefore, we transport *CSH* to main memory, where it takes over the role of the lost *SH*. Then we reconstruct from *SH* the structures *DS* and *INF*. The entire reconstruction procedure takes one disk access, $O(S_{CSH}(n))$ transport time, where $S_{CSH}(n)$ is the size of *CSH*, and an amount of computing time. Note that this computing time is $\Omega(S_{CSH}(n))$, since the shadow administration has to be written in main memory. (We assume that secondary memory does not have DMA.)

3.2 A general technique

For the reconstruction problem, a general technique exists, that is similar to the one in Subsection 2.2. The technique consists of sending to secondary memory all changes to the shadow administration *SH*. These changes are not carried out, they are only stored in secondary memory. To reconstruct the data structure *DS*, we transport the file from secondary memory to main memory. Then we perform the updates in the shadow administration, and finally the structures *DS* and *INF* are reconstructed.

Let *DS* be a data structure and let *SH* and *INF* be the corresponding shadow administration. Let $S(n)$, $U(n)$, $C(n)$ and $R(n)$ denote respectively the size of *SH*, the total update computing time of *SH* and *INF*, the amount of data that is changed in *SH* after an update, and the computing time needed to reconstruct the structures *DS* and *INF* from *SH*. Note that $R(n) = \Omega(S(n))$.

We shall implement this multiple representation, such that the entire shadow structure can be updated in $O(U(n))$ computing time, one disk access and $O(C(n))$ transport time. Also, the other performances remain asymptotically the same. In this new representation the shadow administration in secondary memory is not an exact copy of the one in main memory.

Again, main memory is modeled as an array, the entries of which have unique indices. In each array entry we store an 'indivisible piece of information'.

Let m be the initial number of objects. Initially, *SH* is stored in the first $S(m)$ entries of main memory. Further, main memory contains *DS* and *INF*. In secondary memory we store—in contiguous blocks of size $O(C(m))$—the copy *CSH*. As in Subsection 2.2, initially *CSH* is an exact copy of *SH*, i.e. each indivisible piece is located in the same positions in both memories. Finally, we store in secondary memory an initially empty list *UF*. (*UF* stands for update file.)

Consider a sequence of $S(m)/C(m)$ updates. Note that $S(m)/C(m) \leq m$. Each update is performed in the structures *DS*, *SH* and *INF*, as usual. After each update of the structure *SH*, we send the indices of all changed entries of *SH*, together with the new contents of these entries, to secondary memory. These

changes—of total size $O(C(m))$—are stored in contiguous blocks at the end of the list *UF*. The structure *CSH* is not affected during the updates. So the update of the shadow administration takes $O(U(m))$ computing time, one disk access and $O(C(m))$ transport time.

Suppose we want to reconstruct the structures during these updates. Then we transport *CSH* and *UF* to main memory, where we store *CSH* in the first $S(m)$ positions. Hence pointers in *CSH* indeed 'point' to the correct objects. Next we carry out the at most $S(m)/C(m)$ updates using the list *UF*. Since we know the indices of the entries in *CSH* that have to be changed, each update takes $O(C(m))$ computing time. Hence all updates together take $O(C(m) \times S(m)/C(m)) = O(S(m))$ computing time. After these updates, the resulting structure *CSH* contains the up to date shadow administration. Hence it can take over the role of *SH*. Finally, we reconstruct from *SH* the structures *DS* and *INF* in $R(n)$ computing time, where n is the current number of objects.

Hence the entire reconstruction algorithm takes one disk access, $O(S(m)) = O(S(n))$ transport time and $O(S(m) + R(n)) = O(R(n))$ computing time.

After these $S(m)/C(m)$ updates, we build all structures anew. Then we continue in the same way with a sequence of $S(m')/C(m')$ updates, where m' is the number of objects at that moment.

Clearly, the space requirements in main and secondary memory do not increase asymptotically. Also, it can be shown that the average update complexity of the shadow administration is bounded by $O(U(n))$ computing time, one disk access and $O(C(n))$ transport time, where n is the current number of objects. These average case bounds can be turned into worst case bounds, see [10]. (Then, the number of disk accesses per update increases to two.)

We apply this technique to range trees with a slack parameter; see Mehlhorn [2]. In such a range tree—with slack parameter m—for a set of n points, range queries can be solved in time $O((\log n)^2 2^m/m + t)$, if t is the number of reported answers. The structure has size $O((n \log n)/m)$, and can be built in $O(n \log n + (n \log n)/m)$ time. Here the first term is the time needed to sort the n points to both coordinates, whereas the second term is the actual building time. Therefore, let the shadow administration *SH* consist of two lists, one containing the points ordered to x-coordinate, and the other containing the points ordered to y-coordinate. The structure *INF* consists of two balanced binary trees, having the points in the lists in sorted order in their leaves. Each leaf of such a tree contains a pointer to the corresponding point in the list. Clearly, *SH* and *INF* can be updated in $O(\log n)$ time. In *SH*, an update changes only a constant amount of data.

Hence our technique gives a shadow administration of optimal size $O(n)$, that can be maintained at the cost of $O(\log n)$ computing time, two disk accesses and $O(1)$ transport time in the worst case. Reconstruction takes one disk access, $O(n)$ transport time and $O((n \log n)/m)$ computing time. Now take $m = \log \log n$. Then

the reconstruction computing time is bounded by $O((n \log n)/\log\log n)$, which is asymptotically smaller than the time needed to build the range tree from scratch. Also, the size of the shadow administration is asymptotically smaller than that of the data structure itself!

4 Order decomposable set problems

It turns out that for specific classes of problems efficient solutions for both versions of the multiple representation problem can be designed. In this subsection we consider one such class, the order decomposable set problems.

In a *set problem* we are given a set of objects, and we are asked some question about this set. More precisely, if T_1 and T_2 are sets, then a set problem is a mapping $PR : P(T_1) \rightarrow T_2$. Here $P(T_1)$ denotes the power set of T_1. For example, in the *convex hull problem*, we are given a set S of points, and we are asked to compute their convex hull. Here T_1 is the set of points in euclidean space, and T_2 is the set of all convex polytopes.

We restrict ourselves to set problems, the answers of which can be merged efficiently. That is, once the answers for two separated halves of a set are known, the answer for the entire set can be obtained fast. For such a class of set problems, we maintain the answer for the entire set, by decomposing the set into subsets, and by maintaining the answers for these subsets.

Definition 1 *A set problem $PR : P(T_1) \rightarrow T_2$ is called $M(n)$-*order decomposable*, if there is an order ORD on T_1, and a function $\square : T_2 \times T_2 \rightarrow T_2$, such that for each set $S = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$, 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 $M(n)$ time to compute.

For example, as was shown by Preparata and Hong [7], the three-dimensional convex hull problem is $O(n)$-order decomposable, where ORD is the order according to x-coordinate.

Let PR be an $M(n)$-order decomposable set problem. We sketch a dynamic data structure solving PR (for details, see [4]). Let $S = \{p_1 \leq p_2 \leq \ldots \leq p_n\}$ be a set, ordered according to ORD, for which we want to maintain the answer to PR. Let $f(n)$ be an integer function, such that $1 \leq f(n) \leq n$. Partition S into subsets $S_1 = \{p_1, \ldots, p_{f(n)}\}$, $S_2 = \{p_{f(n)+1}, \ldots, p_{2f(n)}\}$, ...

The data structure DS consists of the following. Each set S_i is stored in a balanced binary tree T_i. Let r_i be the root of T_i. These roots are ordered according to $r_1 < r_2 < r_3 < \ldots$, and we store them in the leaves of a balanced binary tree T. Each node v of T contains the following additional information. Suppose the

subtree of T with root v has $r_i, r_{i+1}, \ldots, r_j$ as its leaves. Then node v contains the answer to the set problem PR for the set $S_i \cup S_{i+1} \cup \ldots \cup S_j$.

It will be clear how this structure can be built using the divide-and-conquer principle.

An insertion of an object p is performed by walking down tree T to find the appropriate root r_i, and by inserting p in the tree T_i. Then we rebuild the answer $PR(S_i)$ and walk back to the root of T. During this walk we rebuild for each node we encounter its additional information by merging the answers stored in its left and right sons. The deletion procedure is similar. By rebuilding the entire structure after $f(n)$ updates, it will remain balanced.

An efficient client structure. Clearly, DS has the property that a very small part of it is used for query answering—the answer to the problem is stored in the root of T—whereas the rest of the structure is used only to update this answer efficiently.

Therefore, we take for the client structure the answer $PR(S)$ to the set problem for the entire set S, and we take for the central structure the full dynamic structure DS. Updates are first performed in the central structure. Then the new answer to the set problem is sent to the clients, where it replaces the old answer.

We apply the above to an $O(n)$-order decomposable set problem, e.g. the three-dimensional convex hull problem; for other examples see [4]. Let $f(n) = \lceil n/\log n \rceil$. Then the central structure has size $O(n \log\log n)$ and an update takes $O(n)$ time on the average. The client structure, however, has size only $O(n)$, and an update takes $O(n)$ transport and computing time on the average.

An efficient shadow administration. Consider DS again. In general, the total size of all trees T_i and all answers $PR(S_i)$ is bounded by $O(n)$. As soon as we have these parts, the rest of the structure can be built very fast: We only have to merge the answers to obtain the tree T with the partial answers in its nodes.

This leads to the following shadow administration. The structure CSH consists of the trees T_i and the answers $PR(S_i)$. (Since the shadow structure consists only of parts of DS itself, we do not need the structures SH and INF.) We divide secondary memory into parts of consecutive blocks, such that each part can contain an answer $PR(S_i)$ and a tree T_i, for a set S_i of cardinality at most $2f(n)$. Then we store in each such part an answer $PR(S_i)$ and its corresponding tree T_i. In each leaf r_i of T, we store the address of the corresponding structures T_i and $PR(S_i)$ in secondary memory.

After an update, only one tree T_i and one answer $PR(S_i)$ will have changed and, hence, have to be transported to secondary memory. Remark that we know from the update of the structure DS the position in secondary memory where these changed structures have to be written. After $f(n)$ updates, all structures are rebuilt.

We again apply the above to an $O(n)$-order decomposable set problem. Let $f(n) = \lceil n/\log n \rceil$. Then we get a shadow administration of size $O(n)$. An update takes one disk access and an average amount of $O(n/\log n)$ transport time. Reconstruction of the structure DS takes $O(n \log\log n)$ computing time, since the tree T has depth $O(\log\log n)$, and on each level we spend $O(n)$ time to merge the answers stored one level below it.

Again we have a shadow administration of size that is asymptotically smaller than that of the data structure itself (which has size $O(n \log\log n)$), and the original data structure can be reconstructed in asymptotically less time than by building it from scratch (which would take $O(n \log n)$ time).

5 Concluding remarks

We have studied two dual versions of the problem of maintaining multiple representations of data structures, and we have given basic models to describe solutions. For both versions we have given a general technique to obtain solutions, in which the transport time in an update is bounded by $O(C(n))$, which is the size of the changes in the client structure or in the shadow administration. So this technique is particularly useful for structures where $C(n)$ is small compared to the time needed to find the changes. We have given some examples of such structures. It would be interesting, however, to have more of them.

We have given techniques that are applicable to order decomposable set problems. There are also several techniques that can be applied to the related class of decomposable searching problems. See [12,13].

There remain some problems and directions for further research:

In this paper, we perform single updates in the data structures. Is it possible to carry out sets of updates more efficiently, than by just performing them one after another?

Finally, one could investigate other multiple representation problems. For example, what to do if the client structures do not necessarily have to represent the same set of objects?

References

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

[2] K. Mehlhorn. *Data structures and algorithms, volume 3: multi-dimensional searching and computational geometry.* Springer Verlag, 1984.

[3] H.J. Olivié. *A new class of balanced trees: half balanced binary search trees.* RAIRO Informatique Théorique **16** (1982), pp. 51-71.

[4] M.H. Overmars. *The design of dynamic data structures.* Springer Lecture Notes in Computer Science, Vol. 156, Springer Verlag, 1983.

[5] M.H. Overmars and M.H.M. Smid. *Maintaining range trees in secondary memory.* Proc. 5-th Annual STACS, Springer Lecture Notes in Computer Science, Vol. 294, Springer Verlag, 1988, pp. 38-51.

[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.* Report FVI-87-14, University of Amsterdam, 1987.

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

[8] M.H.M. Smid. *General lower bounds for the partitioning of range trees.* ITLI Prepublication Series CT-88-02, University of Amsterdam, 1988.

[9] M.H.M. Smid. *A data structure for the union-find problem having good single-operation complexity.* ITLI Prepublication Series CT-88-06, University of Amsterdam, 1988.

[10] M.H.M. Smid. *General techniques for maintaining shadow administrations.* In preparation.

[11] M.H.M. Smid and M.H. Overmars. *Maintaining range trees in secondary memory, part II: lower bounds.* Report FVI-87-15, University of Amsterdam, 1987.

[12] M.H.M. Smid, M.H. Overmars, L. Torenvliet and P. van Emde Boas. *Maintaining multiple representations of dynamic data structures.* ITLI Prepublication Series CT-88-03, University of Amsterdam, 1988.

[13] M.H.M. Smid, L. Torenvliet, P. van Emde Boas and M.H. Overmars. *Two models for the reconstruction problem for dynamic data structures.* Report FVI-87-13, University of Amsterdam, 1987.

[14] L. Torenvliet and P. van Emde Boas. *The reconstruction and optimization of trie hashing functions.* Proc. 9-th International Conf. on Very Large Databases, 1983, pp. 142-156.

The ITLI Prepublication Series

1986

86-01	The Institute of Language, Logic and Information
86-02 Peter van Emde Boas	A Semantical Model for Integration and Modularization of Rules
86-03 Johan van Benthem	Categorial Grammar and Lambda Calculus
86-04 Reinhard Muskens	A Relational Formulation of the Theory of Types
86-05 Kenneth A. Bowen, Dick de Jongh	Some Complete Logics for Branched Time, Part I Well-founded Time, Forward looking Operators
86-06 Johan van Benthem	Logical Syntax

1987

87-01 Jeroen Groenendijk, Martin Stokhof	Type shifting Rules and the Semantics of Interrogatives
87-02 Renate Bartsch	Frame Representations and Discourse Representations
87-03 Jan Willem Klop, Roel de Vrijer	Unique Normal Forms for Lambda Calculus with Surjective Pairing
87-04 Johan van Benthem	Polyadic quantifiers
87-05 Víctor Sánchez Valencia	Traditional Logicians and de Morgan's Example
87-06 Eleonore Oversteegen	Temporal Adverbials in the Two Track Theory of Time
87-07 Johan van Benthem	Categorial Grammar and Type Theory
87-08 Renate Bartsch	The Construction of Properties under Perspectives
87-09 Herman Hendriks	Type Change in Semantics: The Scope of Quantification and Coordination

1988

Logic, Semantics and Philosophy of Language:

LP-88-01 Michiel van Lambalgen	Algorithmic Information Theory
LP-88-02 Yde Venema	Expressiveness and Completeness of an Interval Tense Logic
LP-88-03	Year Report 1987
LP-88-04 Reinhard Muskens	Going partial in Montague Grammar
LP-88-05 Johan van Benthem	Logical Constants across Varying Types
LP-88-06 Johan van Benthem	Semantic Parallels in Natural Language and Computation
LP-88-07 Renate Bartsch	Tenses, Aspects, and their Scopes in Discourse
LP-88-08 Jeroen Groenendijk, Martin Stokhof	Context and Information in Dynamic Semantics
LP-88-09 Theo M.V. Janssen	A mathematical model for the CAT framework of Eurotra

Mathematical Logic and Foundations:

ML-88-01 Jaap van Oosten	Lifschitz' Realizabiility
ML-88-02 M.D.G. Swaen	The Arithmetical Fragment of Martin Löf's Type Theories with weak Σ-elimination
ML-88-03 Dick de Jongh, Frank Veltman	Provability Logics for Relative Interpretability
ML-88-04 A.S. Troelstra	On the Early History of Intuitionistic Logic

Computation and Complexity Theory:

CT-88-01 Ming Li, Paul M.B.Vitanyi	Two Decades of Applied Kolmogorov Complexity
CT-88-02 Michiel H.M. Smid	General Lower Bounds for the Partitioning of Range Trees
CT-88-03 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas	Maintaining Multiple Representations of Dynamic Data Structures
CT-88-04 Dick de Jongh, Lex Hendriks Gerard R. Renardel de Lavalette	Computations in Fragments of Intuitionistic Propositional Logic
CT-88-05 Peter van Emde Boas	Machine Models and Simulations (revised version)
CT-88-06 Michiel H.M. Smid	A Data Structure for the Union-find Problem having good Single-Operation Complexity
CT-88-07 Johan van Benthem	Time, Logic and Computation
CT-88-08 Michiel H.M. Smid, Mark H. Overmars Leen Torenvliet, Peter van Emde Boas	Multiple Representations of Dynamic Data Structures