The Reconstruction Problem for Dynamic Data Structures, an Overview [1]

Michiel Smid
Fachbereich Informatik, Universität des Saarlandes
D-6600 Saarbrücken, West-Germany

The reconstruction problem is the following: given a searching problem, design an efficient main memory data structure that solves this problem, together with a shadow administration, to be stored in secondary memory, such that the original main memory structure can be reconstructed in case of e.g. a system crash. The problem is illustrated by considering a specific example of a searching problem, the union-find problem. Furthermore, a general technique is given to implement any shadow administration efficiently. Finally, the technique of deferred data structuring is applied for solving the reconstruction problem.

1 Introduction

The theory of data structures and algorithms is concerned with the design and analysis of structures that solve searching problems. In a searching problem, we have to answer a question (also called a *query*) about an object with respect to a given set of objects. A data structure for such a searching problem stores the objects in such a way that queries can be answered efficiently. The design of data structures has received considerable attention.

A large part of the research is focussed on designing structures that are stored in the main memory of a computer, on which all standard computations can be performed, and which is usually modeled as a Random Access Machine (RAM). (See [2].) The memory of a RAM consists of an array, the entries of which can store pieces of information, such as names, integers, pointers, etc. Each such array entry can be accessed at constant cost, provided the address of the entry is known. The main problem is to structure the relations of the basic pieces of information, using a small amount of space, such that queries can be answered fast.

[1] The research that is reported here, was done while the author was with the Department of Mathematics and Computer Science of the University of Amsterdam, The Netherlands. This work was supported by the Netherlands Organization for Scientific Research (NWO), and by the ESPRIT II Basic Research Actions Program under contract No. 3075 (project ALCOM).

Until about 1979, many of the main memory data structures that were designed were static, i.e., it was not possible to insert and delete objects. Exceptions were data structures that can handle dictionary operations. The oldest are the AVL-trees, introduced in 1962 by Adel'son-Vel'skiĭ and Landis [1]. In these trees one can search, insert and delete objects in a number of steps that is logarithmic in the number of objects that are stored in the tree. In 1979, the research on general dynamization techniques was initiated by Bentley [4]. This research consists of designing techniques to transform static data structures into dynamic structures, i.e., structures that do allow insertions and deletions of objects. Many techniques are available nowadays that can be applied to large classes of searching problems. As an example, there exists a general theory to dynamize data structures that solve so-called decomposable searching problems. In a decomposable searching problem, the answer to a query with respect to a set of objects can be obtained by merging the partial answers to the query with respect to a partition of this set. Any static data structure that solves a searching problem satisfying this general constraint, can be turned into a dynamic structure. The reader is referred to Overmars [12] for a detailed account of dynamization techniques.

Another part of the research is concerned with the problem of designing structures that are stored and maintained in secondary memory. This problem often occurs in database applications, where data structures are too large to be stored in main memory, and therefore have to be stored in secondary memory.

Secondary memory is modeled as an array that is divided into blocks. In secondary memory, no computing is possible, and the only allowed operations are to replace a block by another one and to add a new block at the end of the file. All computations take place in main memory, and the blocks that store information that is needed during a computation are transported to main memory. If a block is changed during an operation, it is transported back to secondary memory. For each block we need in a computation, we have to access secondary memory, which takes a considerable amount of time in practice.

Therefore, the main problem is to partition the data structure into parts of a small size, such that each operation needs information from only a few parts. Then, by storing each part of the partition in one block in secondary memory, we can perform operations at the cost of only a few disk accesses and a small amount of data transport.

The best-known example of a data structure for secondary memory applications is the B-tree, introduced in 1972 by Bayer and McCreight [3]. If a B-tree stores n objects, and if it is stored in blocks of size m, then the operations search, insert and delete can be performed at the cost of $O(\log\ n/\log\ m)$ accesses to secondary memory in the worst case.

In most studies that have appeared so far, it is assumed that the objects are represented by only one data structure that is stored either in main memory or in secondary memory, and all operations are performed on this one structure. In many situations, however, we need to represent the data more than once—possibly on different storage media—and have a *multiple representation* of the data.

In the author's Ph.D. Thesis [14], such multiple representation problems are

investigated. In this paper, we consider one such problem: The *reconstruction problem.* After a system crash, or as a result of errors in software, a data structure that is stored in main memory can be destroyed. Another case, in which a main memory structure can be destroyed, is the regular termination of an application program that uses the structure. In case of an application that is executed on a system that is also used by other persons, the copy of a data structure in main memory will be destroyed 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.* So besides the data structure in main memory, we represent the data in a shadow administration that is stored in secondary memory.

This leads to the problem of designing for a given searching problem, a dynamic data structure that solves this searching problem, together with a shadow administration from which the data structure can be reconstructed in case of calamity.

This shadow administration does not have to support the same operations as the main memory data structure. Only insertions and deletions have to be performed, whereas on the main structure itself also queries are carried out. Furthermore, we only require that the shadow administration contains enough information that makes it possible to reconstruct the main structure.

The reconstruction problem was suggested by Ghica van Emde Boas-Lubsen, after she heard a talk by Litwin about trie hashing functions. Leen Torenvliet and Peter van Emde Boas investigated this reconstruction and optimization problem for these functions in [20]. A few years later, Peter van Emde Boas, Leen Torenvliet and Mark Overmars together decided that it ought to be possible to study this reconstruction problem in a more general setting. They wrote a research proposal, that was eventually accepted by the Netherlands Organization for Scientific Research (NWO). The project was carried out by the author at the University of Amsterdam, in close collaboration with the above three persons.

Although the reconstruction problem arises in practice, no other research concerning this problem has been carried out besides [20] and the author's Ph.D. research.

Solutions to the reconstruction problem have applications in the following areas:

- The theory of data bases.
- Computational geometry. Since in this area often data structures are used requiring more than linear space, it is sometimes possible to improve asymptotically upon the storage requirements.
- Multiprocessing. A system in which several processors at distinct times execute distinct tasks, and communicate through message passing, might be even more sensitive to crashes than a uniprocessor system. To protect a computation against failure of processors, checkpoints are built in on several places of the computation. If a checkpoint is reached, the complete state of all processors, and their interconnection pattern, is transported to secondary memory. If the system crashes, the computation can be contin-

ued at the last reached checkpoint. It is clear that much time and space can be saved by efficiently storing the information from each processor.

The rest of this paper is organized as follows. In Section 2, we introduce the models of main and secondary memory. In Section 3, we introduce a realistic general framework that we use to describe solutions to the reconstruction problem, and we introduce the complexity measures to express the efficiency of solutions. In Section 4, we consider a specific example of a problem, the union-find problem. For this problem, we design an efficient main memory data structure—in fact, this structure is optimal in a very general class of data structures—in such a way that a copy of it can efficiently be maintained in secondary memory, thereby leading to a good solution to the reconstruction problem. In Section 5, we give a general technique to implement any shadow administration in such a way that updates and reconstruction can be performed at low costs. In Section 6, we apply the recent idea of deferred data structuring to the reconstruction problem. This leads to another approach in the reconstruction procedure: after a crash, the data structure is reconstructed 'on-the-fly', i.e., we immediately proceed with performing queries and updates, and we reconstruct the data structure during these operations. It should be mentioned that Sections 4 and 6 contain results that are also interesting in other areas besides the reconstruction problem. We finish the paper in Section 7 with some concluding remarks and some directions for future research.

2 Storage and computation models

The medium in which all computations take place, and in which also data can be stored, is called *main memory*. We model main memory as a Random Access Machine (RAM). The memory of a RAM consists of an array, the entries of which have unique indices. The contents of such an array entry can be obtained at constant cost, provided its address, i.e., its index, is known. We express the complexity of a computation in main memory in *computing time*, which is the usual measure—in terms of words—to express the length of a computation. (In the theory of algorithms and data structures it is customary to express complexities in terms of words, not in terms of bits.)

Our second storage medium is *secondary memory*. Just as for the RAM, secondary memory consists of an array. Now, this array is divided into *blocks* of a fixed size. This block-size can be chosen arbitrary. Each such block has a unique address, and it is possible to access a block directly, provided its address is known.

A data structure is stored in secondary memory by distributing it over a number of blocks of a predetermined size. In secondary memory no computing is possible. Therefore, to perform an operation—a query or an update—on a data structure, we send information from secondary memory to main memory—where computing is possible—and vice versa. The following update operations are possible in secondary memory:

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

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

Hence, we can only update complete blocks. It is also possible to transport (complete) blocks from secondary memory to main memory. To transport a block to secondary memory, we have to know the address where the block will be stored. Similarly, a block can be transported to main memory only if its address in secondary memory is known.

We express the complexity of an operation in secondary memory by two quantities. In practice, these two quantities dominate the time for the operation. The first one—which is in general the most time consuming—is the number of *disk accesses*—also called *seeks*—that has to be done: For each segment of consecutive blocks we transport, we have to do one disk access. Hence, we can transport the entire data structure in one disk access to secondary memory, provided we store the structure in consecutive blocks. Also, it takes one disk access to transport a structure that is stored in secondary memory in consecutive blocks, to main memory. In this latter case, it is sufficient to know the address—in secondary memory—of the first block of the segment that stores the structure: We transport all blocks 'to the right' of this first block, in which some information is stored. (Here we assume that blocks that do not contain information of the structure, are empty.)

The second quantity is the *transport time*: We assume that an amount of n data can be transported in $O(n)$ transport time from main memory to secondary memory, and vice versa. In general the constants in this estimate for the transport time are incomparable to the constants in computing time.

We already said that in practice the time for one disk access is high. In order to get an impression, for a typical standard computer, one disk access takes about 15 milliseconds, whereas data transport between main and secondary memory is performed at a rate of 3 Mbyte per second. Therefore it is essential to limit the number of disk accesses as much as possible.

3 A MODEL FOR THE RECONSTRUCTION PROBLEM

To study and analyze solutions to the reconstruction problem, we use the following conceptual model. We remark here that this is not the best way of implementing the techniques. Our approach is easy to analyze and does not increase the complexity in order of magnitude. We store the following information:

- DS is a dynamic data structure, stored in main memory.
- SH is a *shadow administration* from which the data structure DS can be reconstructed. This shadow administration is also stored in main memory.
- In secondary memory, we store a copy CSH of the shadow administration SH.
- Finally, there is extra information INF, that is used to update the shadow administration SH and its copy CSH. This extra information is not needed to reconstruct the data structure, and, hence, it may be destroyed in a system crash. Therefore, it is only stored in main memory.

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

Let DS be a dynamic data structure, and let SH, CSH and INF be the corresponding additional structures. To perform an update we carry out the following steps:

1. The data structure DS is updated.
2. The structures SH and INF are updated.
3. The copy CSH in secondary memory is updated.

Steps 1 and 2 take place in main memory. Therefore, all standard operations are allowed for these two steps of the update procedure. The complexity of these steps is expressed in computing time.

In step 3, data in secondary memory has to be updated. The structure CSH is distributed over a number of blocks in secondary memory. After the update of SH, we know which parts of CSH have to be updated. We update CSH by replacing all blocks in which some information has to be changed by the corresponding updated parts of SH. The complexity of this operation is given by the number of disk accesses that has to be done; the amount of transport time which is proportional to the amount of data that is transported; and the amount of computing time needed to collect the information that is transported.

After a system crash, or as a result of program errors, the contents of main memory (i.e., DS, SH and INF) will be destroyed. To reconstruct the structures, we transport the copy CSH of the shadow administration to main memory. This copy takes over the role of the destroyed shadow administration SH. Then we reconstruct from SH the structures DS and INF. After the reconstruction, we proceed with query answering and performing updates.

The reconstruction procedure takes a number of disk accesses, $O(S_{CSH}(n))$ transport time, where $S_{CSH}(n)$ is the size of CSH, and an amount of computing time.

In most cases, the copy CSH of the shadow administration is stored in secondary memory in consecutive blocks, always starting at the same block. This block is called block 0. We assume that the system knows the address of block 0; it is not destroyed in a system crash. Then, the number of disk accesses in the reconstruction procedure is equal to one.

An important issue in the reconstruction procedure is how we store the copy CSH in main memory. Note that data structures contain pointers, which we consider to be indices of memory locations. In order to guarantee that these pointers 'point' to the correct objects, each indivisible piece of information of CSH should be stored in exactly the same location in main memory as its corresponding piece of SH was, before the information was destroyed. In general, this is not possible, because the crash may also have destroyed physical parts of main memory where the information was stored. In this case, we can of course store the information in another part of main memory, in such a way that all addresses are shifted by the same amount.

We assume for simplicity, however, that a crash only destroys the pieces of information; the memory locations themselves are not destroyed. Hence, these locations can be used after the crash to store information again.

We store in secondary memory with each piece of information of CSH, the address of its corresponding piece in main memory. In this way, the size of the structure CSH is at most twice as large as the size of SH. Note that now the structure CSH is not an exact copy, since it contains more information. To reconstruct the structures, we transport CSH to main memory, and we store the information in the same positions as SH was, using the addresses. Then all pointers indeed have the correct meaning, and we can reconstruct DS and INF. It follows that the computing time needed to reconstruct the structures is $\Omega(S_{CSH}(n))$, since in main memory an amount of $S_{CSH}(n)$ information has to be written in the correct positions.

4 An example: the union-find problem

The *union-find problem* is one of the basic problems in the theory of algorithms and data structures. In this problem we are given a collection of n disjoint sets $V_1, V_2, \ldots, V_n$, each containing one single element, and we have to carry out a sequence of operations of the following two types:

1. $UNION(A, B, C)$: combine the two disjoint sets A and B into a new set named C.
2. $FIND(x)$: compute the name of the (unique) set that contains x.

The union-find problem has many applications, and many algorithms use the problem in some way as a subroutine. Examples are algorithms for computing minimum spanning trees, solving an off-line minimum problem, computing depths in trees and determining the equivalence of finite automata. (See [2].) The problem has received considerable attention. See Tarjan [19] and La Poutré [18] for some important results.

In this section we are interested in the single-operation time complexity of the union-find problem. Blum [7] has given a data structure of size $O(n)$, in which each $UNION$ operation can be performed in $O(k+\log_k n)$ time, and each $FIND$ operation in $O(\log_k n)$ time. Here k is a parameter, possibly depending on n. He also gives a very general class $\mathcal{B}$ of data structures, that contains many implementations of known algorithms for the union-find problem:

The class $\mathcal{B}$. Data structures in class $\mathcal{B}$ are linked structures that are considered as directed graphs. The algorithms that use these data structures for solving the union-find problem should satisfy the following constraints:

1. For each set and for each element, there is exactly one node in the data structure that contains the name of this set or element.
2. The data structure can be partitioned into subgraphs, such that each subgraph corresponds to a current set. There are no edges between two such subgraphs.
3. To perform an operation $FIND(x)$, the algorithm gets the node v that contains x. The algorithm follows paths in the graph, until it reaches the node that contains the node of the corresponding set.
4. To perform a $UNION$ or a $FIND$ operation, the algorithm may insert or delete any edges, as long as Condition 2 is satisfied.

For structures in class $\mathcal{B}$, the following theorem holds.

THEOREM 1 (BLUM [7]). *Let DS be any data structure in class $\mathcal{B}$. Suppose that each $UNION$ operation can be performed in $O(k)$ time. Then there is a $FIND$ operation that needs time*

$$\Omega\left(\frac{\log\ n}{\log\ k + \log\ \log\ n}\right).$$

In this section, we give a variant of Blum's structure that gives a better trade-off between the times for $UNION$ and $FIND$ operations. This structure depends on a parameter, and for many values of this parameter it is optimal in the class $\mathcal{B}$.

The data structure consists of a number of trees, and has the property that for a $UNION$ operation we only have to visit the roots of two trees, together with their direct descendants. Furthermore, a $FIND$ operation does not change the structure. This property implies that we can efficiently maintain a copy of the data structure in secondary memory, leading to a good solution to the reconstruction problem.

4.1 The union-find data structure

Let V be a set of n elements for which we want to solve the union-find problem. That is, we want to maintain a partition of V under a sequence of $UNION$ and $FIND$ operations, where initially each set in the partition contains exactly one element. We store sets in $UF(k)$-trees, that are defined as follows.

DEFINITION 1. Let k be an integer, $2 \leq k \leq n$. A tree T is called a $UF(k)$-tree, if

1. the root of T has at most k sons,
2. each node in T has either 0 or more than k grandsons. (Here, a grandson of a node v is a son of a son of v.)

Each set A in the partition of V is stored in a separate $UF(k)$-tree, as follows: The elements of A are stored in the leaves of the tree. In the root, we store the name of the set, the height of the tree, and the number of its sons. Each non-root node contains a pointer to its father, and the root contains pointers to all its sons. Note that the root contains at most k pointers. A $UF(k)$-tree storing a set of cardinality one, consists of two nodes, a root and one leaf.

THE FIND-ALGORITHM. To perform an operation $FIND(x)$, we get at constant cost the leaf that contains element x. Then we follow father-pointers, until we reach the root of the tree, where we read the name of the set that contains x.

THE UNION-ALGORITHM. To perform the operation $UNION(A, B, C)$, we get at constant cost the root r resp. s of the tree that contains the set A resp. B. We distinguish three cases.

CASE 1. The trees containing A and B have equal height, and the total number of sons of r and s is $\leq k$.

Assume without loss of generality that the number of sons of s is less than or equal to the number of sons of r. We change the father-pointers from all sons of s into pointers to r, and we store in r pointers to its new sons. Next, we discard the root s, together with all its information. Finally, we adapt in r the number of its sons and the name of the set. It is clear that the resulting tree is again a $UF(k)$-tree.

CASE 2. The trees containing A and B have equal height, and the total number of sons of r and s is $> k$.

In this case, we create a new root t. In this new root, we store pointers to r and s; the name of the new set C; the height of the new tree, which is one more than the corresponding value stored in r; and the number of sons, which is 2. In the old roots r and s, we discard all information, and we add pointers to their new father t. Again, the resulting tree is a $UF(k)$-tree.

CASE 3. The trees containing A and B have unequal height.

Assume without loss of generality that the tree of B has smaller height than the tree of A. Let v be an arbitrary son of r. Then we change the father-pointers from all sons of s into pointers to v. The root s, together with all its information, is discarded. Also, we adapt the name of the set stored in r. Note that the height of the tree and the number of sons of r does not change. Again, it is not difficult to see that the resulting tree is a $UF(k)$-tree.

THEOREM 2. *Let k and n be integers, such that $2 \leq k \leq n$. Using $UF(k)$-trees, the union-find problem on n elements can be solved, such that*

1. *each $UNION$ takes $O(k)$ time,*
2. *each $FIND$ takes $O(\log_k n)$ time,*
3. *the data structure has size $O(n)$.*

PROOF. We saw already that the $UNION$-algorithm correctly maintains $UF(k)$-trees. Note that we can determine in constant time in which of the three cases we are, since all information for deciding this is stored in the roots. Cases 1 and 3 of the $UNION$-algorithm take $O(k)$ time in the worst case. Case 2 can be handled in $O(1)$ time.

The size of a $UF(k)$-tree is linear in the number of its leaves, which shows that the entire data structure has size $O(n)$.

The time needed for a $FIND$ operation is bounded above by the height of a $UF(k)$-tree. The problem is that Definition 1 does not imply that the height of a $UF(k)$-tree is bounded above by $O(\log_k n)$. In fact, the reader is encouraged to construct a $UF(k)$-tree having a height that is proportional to n/k. Of course, we only have to give an upper bound on the heights of the trees that are made by the $UNION$-algorithm. It can be shown, that the trees that are made by the $UNION$-algorithm, have height at most $1 + 2\lceil \log_k n \rceil$. (For a proof of this fact, see [14,15].) Therefore, each $FIND$ operation takes $O(\log_k n)$ time in the worst case. □

It is clear that the given data structure is contained in Blum's class $\mathcal{B}$. Therefore, Theorems 1 and 2 yield:

COROLLARY 1. *The data structure of Theorem 2 is optimal in Blum's class $\mathcal{B}$ of structures for the union-find problem, for all values of k satisfying $k = \Omega((\log n)^\epsilon)$ for some $\epsilon > 0$.*

4.2 An efficient shadow administration

In this section we show that by using the data structure of Theorem 2, we can obtain an efficient solution to the reconstruction problem.

We store a copy of the data structure in secondary memory as follows. We reserve a number of consecutive blocks of some predetermined size (see below), and we distribute the structure over these blocks. Together with each indivisible piece of information, we store in secondary memory the address of the corresponding piece in main memory.

Since the root of a $UF(k)$-tree has at most k sons, the total size of this root, together with all its sons and all the information stored in these nodes (i.e., pointers, name of the set, height of the tree and number of sons), and all their addresses in main memory, is bounded above by ck for some constant c. Also, there is a constant c' such that the size of the entire data structure, together with all the addresses, is at most $c'n$.

We reserve in secondary memory $\lceil (c'n)/(ck) \rceil$ consecutive blocks of size $2ck$, starting at block 0. The copy of the data structure will be stored in these blocks. We call a block *free* if at least half of the block is empty. The following lemma can easily be proved.

LEMMA 1. *Among the reserved blocks, there is always at least one free block.*

Initially we have n trees, each of them having one root and one leaf. We store these trees in main memory. Copies of the trees are distributed over the reserved blocks. For each tree, the root and its son, together of course with all their information and their positions in main memory, are stored in the same block. We store in main memory in the root of each tree, the address of the block in secondary memory that contains the copy of this root. Finally, we maintain in main memory a stack containing the addresses of the free blocks. By Lemma 1, this stack is never empty. The stack will only be used for updating the structure in secondary memory; it is not used for reconstructing the data structure. Therefore it may be destroyed in a crash. Note that the amount of space in main memory remains bounded by $O(n)$.

Since a $FIND$ operation does not change the data structure, such an operation does not affect the shadow administration.

A $UNION$ operation is first performed on the structure in main memory according to the algorithm of Section 4.1. Then the shadow administration in secondary memory is updated. We take care that at each moment the following holds:

INVARIANT. For each $UF(k)$-tree, the root and all its sons, together with all the information stored in these nodes, and all their positions in main memory, are stored in the same block in secondary memory.

Clearly, this invariant holds initially. (In the sequel we shall not state each time explicitly that if we put information in a block, we also store with it its position in main memory. It is clear how this can be done.)

THE UNION-ALGORITHM. The operation $UNION(A, B, C)$ is performed as follows. Let r resp. s be the root of the tree containing the set A resp. B.

CASE 1. The trees containing A and B have equal height, and the total number of sons of r and s is $\leq k$.

Assume without loss of generality that the number of sons of s is less than or equal to the number of sons of r. In the block containing r we remove this root and all its sons. (Note that we can read the address of this block in the root r that is stored in main memory.) If this block becomes free, we put its address on the stack. In the block containing s we do the same. Next we take the address of a free block from the stack, and in that block we add the root, together with its sons, of the new tree. If this block remains free we put its address back on the stack. In main memory, we store in the root of the new tree, the address of the block containing its copy.

CASE 2. The trees containing A and B have equal height, and the total number of sons of r and s is $> k$.

In the block containing r we remove this root, together with all the information stored in it. If the block becomes free, we put its address on the stack. In the block containing s we do the same. Then we add the new root, together with its sons r and s and all the information that these three nodes contain, to a free block, the address of which we take from the stack. If this block remains free its address is put back on the stack. In main memory we store in the new root the address of the block containing its copy.

CASE 3. The trees containing A and B have unequal height.

Assume without loss of generality that the tree of B has smaller height than the tree of A. In the block containing r we change the name of the set from A to C. In the block containing s, we change the pointers of the sons of s, and we remove the root s together with all its information. If this block becomes free we put its address on the stack.

THE RECONSTRUCTION ALGORITHM. To reconstruct the data structure, we transport the entire file to main memory, and we rebuild the stack of free blocks. Then each indivisible piece of information of the data structure is stored in the array location where it was before the information was destroyed. This guarantees that each pointer 'points' to the correct position in main memory. Now we can proceed performing $UNION$ and $FIND$ operations.

The following theorem summarizes the result.

THEOREM 3. *Let k and n be integers, such that $2 \leq k \leq n$. For the data structure of Theorem 2, solving the union-find problem on n elements, there exists a shadow administration*

1. *of size $O(n)$,*
2. *that can be maintained after a $UNION$ operation at the cost of at most three disk accesses, $O(k)$ computing time and $O(k)$ transport time.*

The data structure can be reconstructed at the cost of one disk access, $O(n)$ transport time and $O(n)$ computing time.

The data structure of this section nicely illustrates how shadow administrations can be implemented. It also shows that the copy in secondary memory is stored in such a way that all pieces of information are mixed up together. For example, the pointers of the main memory structure do not have any meaning in secondary memory. This does not matter, since we only require that the shadow administration contains information from which the original main memory structure can be reconstructed.

The structure of this section cannot be applied in a scenario where the data structure is too large to be stored in main memory. In that case, the structure is maintained in secondary memory. The reason that the structure cannot be applied in this situation is that then the pointers must have a meaning in secondary memory. The maintenance of the correct meaning of these pointers will take a lot of disk accesses.

5 A GENERAL TECHNIQUE

In this section, we give a technique to implement *any* shadow administration. Let DS be a dynamic data structure and let SH and INF be a corresponding shadow administration. (See Section 3 for the notation.) We denote the size of SH by $S_{SH}(n)$, the size of INF by $S_{INF}(n)$, the total update computing time of SH and INF by $U_c(n)$, and the computing time needed to reconstruct the structures DS and INF from SH by $R_c(n)$. Let $C(n)$ be the amount of data that is changed in an update in SH. We assume that all these complexity measures are smooth and non-decreasing. (A function f is called *smooth* if $f(O(n)) = O(f(n))$.)

We show how to implement these structures, such that the entire shadow administration can be updated in one disk access, $O(U_c(n))$ computing time and $O(C(n))$ transport time. These bounds are amortized bounds. Also, the total size of the additional structures is bounded by $O(S_{SH}(n)+S_{INF}(n))$, and reconstruction takes one disk access, $O(S_{SH}(n))$ transport time, and $O(R_c(n))$ computing time. Hence, this technique is especially interesting if $C(n) = o(U_c(n))$, i.e., if the amount of data that is changed in the shadow administration is much smaller than the time needed to find these changes. We will need the following lemma.

LEMMA 2. *The complexity measures introduced above satisfy*

1. $C(n) \leq U_c(n)$.
2. $S_{SH}(n) \leq n \times C(n)$.

PROOF. To update SH, we spend at most $U_c(n)$ time. In this update, the amount of data that is changed can never be greater than $U_c(n)$. Therefore, $C(n) \leq U_c(n)$. We can build the structure SH, by performing n insertions into an initially empty structure. In this way, the total size of the changes is at most $C(1) + C(2) + \cdots + C(n) \leq n \times C(n)$. During these insertions, we have built a structure of size $S_{SH}(n)$, and hence an amount of at least $S_{SH}(n)$ data is changed. This proves that $S_{SH}(n) \leq n \times C(n)$. □

5.1 Implementing the shadow administration

THE STRUCTURES. Let m be the initial number of objects represented by the data structure DS. We store DS and the corresponding additional structures SH and INF in main memory. In secondary memory we store—in consecutive blocks, starting at block 0—the copy CSH of SH. This copy CSH contains with each piece of information the address of the corresponding piece in main memory. We also store in secondary memory an initially empty list UF. (UF stands for update file.) This list is positioned in the block 'next to' CSH.

THE UPDATE ALGORITHM. Consider a sequence of $S_{SH}(m)/(2C(m))$ updates. Each update in this sequence is performed, in main memory, on the structures DS, SH and INF. After an update of the structure SH, we send the addresses of all changed entries of SH, together with the new contents of these entries, to secondary memory. (Note that main memory consists of an array. The structure SH is stored in the entries of this array. Each such entry has a unique address.) These changes—of total size $O(C(n))$, where n is the current number of objects—are stored in a new block at the end of the list UF. The structure CSH is not affected during the updates.

After $S_{SH}(m)/(2C(m))$ updates have been performed in this way, we transport a copy—that is called CSH again—of the up-to-date structure SH, together with the addresses in main memory, to secondary memory. This copy CSH is stored in consecutive blocks, starting at block 0, and it replaces the old structures CSH and UF. If the size of the new copy CSH is less than the total size of the old CSH and UF, we make the blocks at the end of the file, that contain the old information, empty. We also initialize in secondary memory an empty list UF in the block 'next to' the new CSH. Then we continue in the same way, now with a sequence of $S_{SH}(m')/(2C(m'))$ updates, where m' is the number of objects at this moment.

THE RECONSTRUCTION ALGORITHM. To reconstruct the structures, we transport CSH and UF to main memory, where we store CSH in the correct locations using the addresses. Then pointers in CSH indeed 'point' to the correct objects. Next we carry out the at most $S_{SH}(m)/(2C(m))$ updates using the list UF. (This list gives us the addresses of the entries in CSH that have to be changed, and the new contents of these entries.) 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. Then all information is reconstructed, and we can proceed with answering queries and performing updates.

THEOREM 4. *Let SH and INF be a shadow administration for the dynamic data structure DS, with complexity $S_{SH}(n)$, $S_{INF}(n)$, $U_c(n)$, $R_c(n)$ and $C(n)$. We can implement these structures such that the resulting shadow administration*

1. *has size $O(S_{SH}(n) + S_{INF}(n))$,*
2. *can be updated in one disk access, an amortized computing time of $O(U_c(n))$, and an amortized transport time of $O(C(n))$.*

The structures DS, SH and INF can be reconstructed in one disk access, $O(S_{SH}(n))$ transport time and $O(S_{SH}(n) + R_c(n))$ computing time.

PROOF. First note that all information in secondary memory is stored in consecutive blocks, always starting at block 0. In particular, there are no gaps. Therefore, the amount of space used in secondary memory is proportional to the total size of the structures CSH and UF. The size of CSH, together with the corresponding addresses in main memory, is equal to $O(S_{SH}(m))$, where m is the number of objects at the beginning of the sequence of updates. During this sequence, n—the current number of objects—satisfies $n \leq m + S_{SH}(m)/(2C(m))$. It follows from Lemma 2, that $n \leq 3m/2$. Similarly, $n \geq m/2$, and hence $n = \Theta(m)$. Since our complexity measures are assumed to be smooth, we have $C(n) = \Theta(C(m))$. Hence in each update we add $O(C(n)) = O(C(m))$ data to the list UF. It follows that the size of UF is bounded by $(S_{SH}(m)/(2C(m))) \times O(C(m)) = O(S_{SH}(m))$. Therefore, the total amount of space used in secondary memory is bounded by $O(S_{SH}(m)) = O(S_{SH}(n))$. The amount of space used in main memory by the shadow administration is bounded by $S_{SH}(n) + S_{INF}(n)$. This proves the bound on the space complexity.

It follows from the given algorithm that the number of disk accesses in an update is equal to one. The amortized transport time for an update is bounded by

$$O\left(C(n) + \frac{O(S_{SH}(m'))}{S_{SH}(m)/(2C(m))}\right) = O(C(n)),$$

where m' is the number of objects at the end of the sequence of updates. (Note that $n = \Theta(m) = \Theta(m')$.) Similarly, the amortized computing time for an update is bounded by

$$O\left(U_c(n) + \frac{O(S_{SH}(m'))}{S_{SH}(m)/(2C(m))}\right) = O(U_c(n) + C(m)) = O(U_c(n)).$$

Here we have used Lemma 2.

In the reconstruction algorithm, it takes one disk access and $O(S_{SH}(n))$ transport time and computing time to transport CSH and UF to main memory and to store CSH in the correct positions. Each update from the list UF takes $O(C(m))$ computing time. It follows that all updates from UF together take an amount of computing time that is bounded by $O((S_{SH}(m)/C(m)) \times C(m))$

$= O(S_{SH}(m))$. Finally, it takes $R_c(n)$ computing time to reconstruct the structures DS and INF from the up-to-date structure CSH. Hence, the entire reconstruction algorithm takes one disk access, $O(S_{SH}(n))$ transport time and $O(S_{SH}(n) + R_c(n))$ computing time. □

REMARK. The amortized bounds in Theorem 4 can be made worst-case, by spreading out the transport of the copy of SH over a number of updates. Then, the number of disk accesses for an update resp. reconstruction increases to two resp. three. See [14]. So this result gives an efficient implementation of *any* shadow administration. Especially the update algorithm is interesting: it takes only two disk accesses and a small amount of transport time, even in the *worst case*.

We illustrate Theorem 4 with an example. Consider the union-find data structure of Section 4.1. In the notation of the present section, DS is the union-find structure, SH is a copy of DS, and INF is the stack that contains the addresses of the free blocks. (Here we have an example where strictly speaking the structure SH is not needed because it is an exact copy of DS.)

It is clear that the size of the shadow administration is bounded by $O(n)$. It follows from Theorem 2 that this shadow administration can be updated after a $UNION$ operation in $O(k)$ computing time. Hence, $U_c(n) = O(k)$ and $C(n) = O(k)$. (Note that a $FIND$ operation does not change the structure.) Finally, $R_c(n) = O(n)$.

Applying the worst-case version of Theorem 4—see the above remark—gives an implementation of this shadow administration of size $O(n)$, that can be updated after a $UNION$ operation in two disk accesses, and $O(k)$ computing and transport time in the worst case. Reconstruction takes three disk accesses, and an amount of $O(n)$ transport and computing time.

This result is similar to that of Theorem 3; only the number of disk accesses differs. Note that the implementation of Section 4.2 will be easier in practice than the one of the present section. (Clearly, considering a specific shadow administration will in general lead to a better result than by applying a general technique that works for any shadow administration.)

In [14], other applications of Theorem 4 can be found.

6 ANOTHER APPROACH: DEFERRED DATA STRUCTURING

In the solutions we have seen so far for the reconstruction problem, we first completely rebuild the data structure DS and the corresponding structures SH and INF, after a crash. Then we proceed with query answering and performing updates. Hence, if the reconstruction time is high, it takes a lot of time before we can proceed again. To avoid this problem, we introduce another approach to the reconstruction problem. The idea is to maintain in secondary memory the objects that are represented by the data structure DS. If we want to reconstruct this data structure, we transport the objects to main memory. Then we immediately continue with answering queries and performing updates. The data structure is built 'on-the-fly' during these operations. With each operation, those parts of the data structure that do not exist at that moment, but that are

needed in the operation, are built. These parts can then be used for future operations.

This technique of building a data structure is due to Karp, Motwani and Raghavan [9], who call it *deferred data structuring*, although they do not apply this technique to the reconstruction problem. Their motivation to design deferred data structures is to solve a sequence of queries, where the length of the sequence is not known. They only give static deferred data structures. The design of deferred data structures for dynamic data sets in which insertions and deletions are allowed concurrently with queries, is left as an open problem.

In this section, we give one technique to dynamize a static deferred data structure. The idea is illustrated by considering dynamic deferred structures for the member searching problem. We show that deferred binary search trees—if properly chosen—can be maintained by this generalized technique.

6.1 The static deferred binary search tree

We first recall the static solution of [9] for the member searching problem.

Let V be a set of n objects drawn from some totally ordered universe U. We are asked to perform—on-line—a sequence of member queries. In each such query we get an object q of U, and we have to decide whether or not $q \in V$.

The algorithm that answers these queries builds a binary search tree as follows. Initially there is only the root, containing the set V. Consider the first query q. We compute the median m of V, and store it in the root. Then we make two new nodes u and v. Node u will be the left son of the root, and we store in it all objects of V that are smaller than m. Similarly, v will be the right son of the root, and we store in it the objects of V that are larger than m. Then we compare the query object q with m. If $q = m$ we know that $q \in V$, and we stop. Suppose $q < m$. Then we proceed in the same way with node u. That is, we find the median of all objects stored in u, we store this median in u, we give u two sons with the appropriate objects, and we compare q with the new median. This procedure is repeated until we either find a node in which the 'local' median is equal to q, in which case we are finished, or end in a node storing only one object not equal to q, in which case we know that $q \notin V$.

The first query takes $O(n + n/2 + n/4 + \cdots) = O(n)$ time, since in each node we have to find a median, which can be done in linear time [6]. During this first query, however, we have built some structure that can be used for future queries: In the second query, we have to perform only one comparison in the root to decide whether we have to proceed to the left or right son. In fact, in any node we visit that is visited already before, we spend only one comparison.

This is the general principle in deferred data structuring: If we do a lot of work to answer one query, we do it in such a way that we can take advantage from it in future queries.

We now describe the algorithm in more detail. Each node v in the structure contains a list $L(v)$ of objects, two variables $N(v)$ and $key(v)$, and two pointers. Some of these values may be undefined. The value of $N(v)$ is equal to the number of objects that are stored in the subtree with root v. The meaning of the other variables will be clear from the algorithms below. (Strictly speaking, the variable $N(v)$ is not needed in the static case.)

INITIALIZATION. At the start of the algorithm there is one node, the root r. The list $L(r)$ stores all objects of V. (This list is *not* sorted.) The value of $N(r)$ is equal to n, which is the cardinality of V, and the value of $key(r)$ is undefined.

EXPAND. Let v be a node having an undefined variable $key(v)$. In this case, the list $L(v)$ will contain at least 2 objects, and the value of $N(v)$ will be equal to $\mid L(v) \mid$. The operation *expand* is performed as follows:

First we compute the median m of $L(v)$, and we determine the sets $V_1 = \{x \in L(v) \mid x < m\}$ and $V_2 = \{x \in L(v) \mid x > m\}$. Then we set $key(v) := m$ and $L(v) := \emptyset$. Next we make two new nodes v_1 and v_2. Node v_1 will be the left son of v, so we store in v a pointer to v_1. If $\mid V_1 \mid > 1$, we set $L(v_1) := V_1$, $N(v_1) :=\mid V_1 \mid$ and $key(v_1) :=$ undefined. If $\mid V_1 \mid = 1$, we set $L(v_1) := \emptyset$, $N(v_1) := 1$ and $key(v_1) := s$, where s is the (only) object of V_1. (Of course, if $V_1 = \emptyset$, we do not create the node v_1.) Similarly for v_2.

ANSWERING ONE QUERY. Let q be a query object, i.e., we want to know whether or not $q \in V$. Then we start at the root, and we follow the appropriate path in the deferred tree, by comparing q with the values of key in the nodes we encounter. If one of these key values is equal to q we know that $q \in V$ and we are finished.

If we encounter a node v having an undefined variable $key(v)$, we expand node v, as described above. Then we proceed our query by comparing q with the value of $key(v)$. If $q = key(v)$, we know that $q \in V$, and we can stop. Otherwise, if $q < key(v)$, we expand the left son of v, and we continue in the same way. If this left son does not exist, we know that $q \notin V$. Similarly, if $q > key(v)$.

The following theorem gives the complexity of the algorithm. For a proof, see [9] or Section 6.2. (The proof in Section 6.2 is a generalization of the proof in [9] to the dynamic case.)

THEOREM 5. *A sequence of k member queries in a set of n objects can be solved in total time $O(n \log k)$ if $k \leq n$, and $O((n+k) \log n)$ if $k > n$.*

In [9], it is shown that this theorem gives an optimal result: The number of comparisons needed to perform k member queries in a set of size n is $\Omega((n + k) \times \log min(n, k))$. In fact, this lower bound even holds in the off-line case, i.e., in case the queries are known in advance.

6.2 A dynamic solution

Consider the deferred tree of the preceding section. At some point in the sequence of queries, the structure consists of a number of nodes. Take such a node v.

Suppose $key(v)$ is defined. Then the list $L(v)$ is empty, the value of $N(v)$ is equal to the number of objects that are stored in the subtree with root v, and the value of $key(v)$ is equal to the median of the objects stored in this subtree.

If $key(v)$ is undefined, node v contains a list $L(v)$ storing a subset of V—those

objects that 'belong' in the subtree of v—and the variable $N(v)$ has the value $\mid L(v) \mid$, which is at least two.

AN UPDATE ALGORITHM. We only give the insert algorithm. Deletions can be performed similarly. See [13,14]. Suppose we have to insert an object x. Then we start searching for x in the deferred tree, using the *key* values stored in the encountered nodes. In each node v we encounter, we increase the value of $N(v)$ by one, since the object x has to be inserted in the subtree of v.

If we end in a leaf, we insert x in the standard way, by creating a new node for it, and we set the variables L, N and *key* to their correct values. (A node v in the deferred tree is called a leaf if $N(v) = 1$. So a node that is not expanded—such a node does not have any sons—is not a leaf.) Note that if x is already present in the deferred tree, we will have encountered it. In that case, we have to decrease the values of the increased $N(v)$'s by one.

If we do not end in a leaf, we reach a node w with an undefined *key* value. Since we have to check whether x is already present in the structure, we have to walk along the list $L(w)$. (The list $L(w)$ is not sorted!) If x is present, we decrease the increased $N(v)$'s. Otherwise, if x is a new object, we add it to the list, and increase $N(w)$ by one. Note that the walk along $L(w)$ takes $O(\mid L(w) \mid)$ time. Hence a number of such insertions would take a lot of time. Then, our general principle—if we do a lot of work, we do it in such a way that it saves work in future operations—is violated. Therefore, after we have checked whether x is a new object, and—in case it is—after we have added x to the list $L(w)$, we expand node w. So if we again have to insert an object in the subtree of w, the time for this insertion will be halved.

Of course, we have to take care that the deferred tree remains balanced. We will consider this problem below.

We are left with the problem of keeping the deferred tree balanced. There are various types of balanced binary search trees that can be maintained after insertions and deletions. The oldest are the AVL-trees, see [1]. The balance condition for these trees depends on the exact heights of subtrees. Since in our deferred tree several subtrees are not complete during the sequence of operations, their exact heights will not be known. So AVL-trees do not seem appropriate for deferred trees.

There exists, however, a class of balanced binary search trees, for which the balance criterion depends only on the size of its subtrees:

DEFINITION 2 (NIEVERGELT AND REINGOLD [11]). Let α be a real number, $0 < \alpha < 1/2$. A binary tree is called a BB[α]-*tree*, if for each internal node v, the number of leaves in the left subtree of v divided by the number of leaves in the entire subtree of v, lies in between α and $1 - \alpha$.

Note that for our deferred trees, the size of each subtree—whether it has been completely built already or not—is known at each moment: It is stored in the variable $N(v)$.

THE PARTIAL DISMANTLING TECHNIQUE. This technique enables us to keep the deferred binary trees balanced. It is a generalization of Lueker's partial re-

building technique. (See [10].) This generalized technique can also be applied to dynamize other deferred data structures.

Our data structure is a deferred BB[α]-tree. Updates are performed as described above. Rebalancing is carried out as follows. After the insertion or deletion, we walk back to the root of the deferred tree to find the highest node v that is out of balance. Then we *dismantle* the subtree with root v. That is, we collect all objects that are stored in this subtree, and put them in the list $L(v)$. Furthermore, we set $key(v) :=$ undefined. Note that the value of $N(v)$ is already equal to $\mid L(v) \mid$. Finally, we discard all nodes in the subtree of v (except for v itself).

Such a dismantling operation takes $O(N(v))$ time. Hence, if v is high in the tree, this will take a lot of time. It turns out, however, that this does not occur too often.

THEOREM 6. *A sequence of $k \leq n$ member queries, insertions and deletions in a set of initially n objects can be performed in total time $O(n \log k)$.*

PROOF. Let $f(n,k)$ denote the total time to perform a sequence of k member queries and updates in a set of initially n objects, with the above algorithms. It is shown on page 53 of [12], that there is a constant c, such that during a sequence of $\leq cn$ updates, the root of the deferred tree always satisfies the balance condition of Definition 2. So in a sequence of $k \leq cn$ queries and updates, the root of the tree is expanded exactly once. The total time we spend in the root in such a sequence is therefore bounded by $O(n+k) = O(n)$. If k_1 operations are performed in the left subtree, we spend an amount of time there bounded by $f(n/2,\ k_1)$, since the left subtree initially contains $n/2$ objects. Similarly, we spend an amount of $f(n/2,\ k-k_1)$ time in the right subtree. It follows that

$$f(n,k) \leq \max_{0 \leq k_1 \leq k} \{f(n/2,\ k_1) + f(n/2,\ k-k_1)\} + c_1 n \quad \text{if } k \leq cn,$$

for some constant c_1.

Each query or update takes $O(m)$ time if m is the number of objects. Therefore, a sequence of k operations takes $O(k(n+k))$ time, since the number of objects is always $\leq n+k$. It follows that

$$f(n,k) \leq c_2 k^2 \text{ if } k \geq cn,$$

for some constant c_2.

It can easily be shown by induction that $f(n,k) = O(n \log k + k^2)$. So a sequence of $k \leq \sqrt{n}$ queries and updates takes $O(n \log k)$ time.

After $\sqrt{n}$ operations, we have spent already $\Omega(n \log n)$ time. Therefore, we build in the $\sqrt{n}$-th operation a binary tree for the objects that are present at this moment. So the $\sqrt{n}$-th operation takes $O(n \log n)$ time. The future operations are performed in this complete structure in the standard non-deferred way. This proves the theorem. □

There are other techniques to dynamize static deferred data structures. These techniques are generalizations of known methods for 'ordinary', i.e., non-deferred, data structures. See [8,13,14].

6.3 *Applications to the reconstruction problem*

We now apply the technique of deferred data structuring to the reconstruction problem. Let DS be a dynamic data structure representing a set V of n objects. Suppose that the structure DS can be built in a deferred way. We take for DS a shadow administration that stores the objects of V in sorted order.

So let SH be a sorted list that stores the objects of the set V. Let INF be a balanced binary search tree that contains the objects of V in sorted order in its leaves. Each leaf—storing say object p—contains a pointer to object p in the list SH.

This structures SH and INF can be updated in $O(\log\ n)$ time. Clearly, an update changes only a constant amount of data in the list SH. So in the notation of Section 5, we have $U_c(n) = O(\log\ n)$ and $C(n) = O(1)$. Therefore, applying the worst-case version of Theorem 4, this shadow administration can be implemented in $O(n)$ space, such that an update takes $O(\log\ n)$ computing time, two disk accesses, and $O(1)$ transport time.

Suppose all information in main memory is destroyed. Then we transport the structures from secondary memory to main memory, we make the sorted list SH up-to-date as described in Section 5.1. Then we build the binary tree INF, using the list SH. By Theorem 4 and the remark made after it, this takes three disk accesses, $O(n)$ transport time and $O(n)$ computing time. (Note that SH is a sorted list. Therefore the tree INF can be built in linear time.)

At this moment, main memory contains the objects in sorted order. We immediately proceed with answering queries and performing updates in the structure DS, in a deferred way. Therefore, the first operations take a lot of time, but the operations will be executed faster and faster the more operations are performed. The data structure DS will be reconstructed gradually *during* the operations. Note that we now start with the objects in sorted order; in Sections 6.1 and 6.2, we started with an unsorted set of objects.

As an illustration, consider the *range counting problem.* Here, we are given a set V of n points in the d-dimensional real vector space. For a given query hyperrectangle $([x_1 : y_1], \ldots, [x_d : y_d])$, we have to report the number of points in V that are in this rectangle. That is, we want the number of points $p = (p_1, \ldots, p_d)$ in V, such that $x_1 \leq p_1 \leq y_1, \ldots, x_d \leq p_d \leq y_d$.

It was shown by Bentley [5], that for this problem a (static) data structure exists of size $O(n(\log\ n)^{d-1})$, that can be built in $O(n(\log\ n)^{d-1})$ time, and in which range counting queries can be solved in $O((\log\ n)^d)$ time. This structure consists of a 'binary tree of binary trees': There is a binary tree, in which each node contains a pointer to another binary tree, each node of which contains a pointer to another binary tree, etc.

Using a similar technique as in Subsection 6.1, it can be shown that a static deferred version of this structure exists, such that a sequence of $k \leq n$ range counting queries can be solved in $O(n(\log\ k)^{d-1} + k(\log\ n)^d)$ time, if the points are ordered according to one of their coordinates. (See also [9].)

Using the partial dismantling technique of the preceding section, a dynamic deferred solution for the range counting problem can be obtained. In fact, then the update algorithm for the dynamic deferred structure is almost the same as Lueker's algorithm that dynamizes a range tree. (See [10].) The result is ex-

pressed in the following theorem.

THEOREM 7. *A sequence of $k \leq n$ range counting queries, insertions and deletions in a set of initially n points in d-dimensional space, initially ordered according to one of their coordinates, can be performed in total time $O(n(\log k)^{d-1} + k(\log n)^d)$.*

If we apply the technique from Section 6.2, then we build after $\sqrt{n}$ operations a complete data structure—in $O(n(\log n)^{d-1})$ time—and we proceed in the non-deferred way. Since we have spent already an amount of $\Omega(n(\log n)^{d-1})$ time after these $\sqrt{n}$ operations, this does not increase the total time for the entire sequence of operations. The $\sqrt{n}$-th operation, however, takes a lot of time. We can get rid of this expensive operation, by building the complete data structure during the first $\sqrt{n}$ operations. With each operation, we count the number of steps we spend in the deferred data structure. Then we spend the same number of steps in building the complete structure. It follows that after these $\sqrt{n}$ operations, the non-deferred structure is completely built. Then we use this structure for future operations; the deferred structure is discarded.

So we have a dynamic deferred data structure for the range counting problem. Now take as a shadow administration the points represented by the structure, ordered according to one of their coordinates. Then after a crash, we reconstruct the ordered list SH of points and the binary tree INF, as described before, in three disk accesses, $O(n)$ transport time and $O(n)$ computing time. Then we immediately proceed with performing operations in the deferred way. Of course, with each update, we also maintain the shadow administration. In this new approach, the first operation takes $O(n)$ time. The data structure will become, however, more complete, and the operations will be executed faster and faster the more operations are performed. In fact, by Theorem 7, we can perform $\Theta(n/\log n)$ operations in $O(n(\log n)^{d-1})$ time.

Using the old approach, in which we completely reconstruct the data structure before we proceed with query answering and performing updates, it takes $O(n(\log n)^{d-1})$ computing time before we can proceed, since the data structure has size $O(n(\log n)^{d-1})$. Then the first $n/\log n$ operations also take $O(n(\log n)^{d-1})$ time, because each operation takes, amortized, $O((\log n)^d)$ time.

Hence, in the approach of the current section, the first $n/\log n$ operations take the same amount of time as we would have needed in the old approach. In this new approach, however, we do not have to wait $O(n(\log n)^{d-1})$ time before we can start with the operations. (Also in the $\sqrt{n}$-th operation, we do not have to wait $O(n(\log n)^{d-1})$ time until the non-deferred structure is built.)

7 CONCLUDING REMARKS AND DIRECTIONS FOR FUTURE RESEARCH

In this paper, we have given an overview of one multiple representation problem: the reconstruction problem. In the author's Ph.D. Thesis, more techniques are given for designing shadow administrations. For example, there are general techniques to design shadow administrations for the data structures solving large classes of searching problems, such as decomposable searching problems and order decomposable set problems. (See also [17].) In fact, many techniques that

were designed for main memory data structures, can be generalized to shadow administrations.

In the present paper, we have considered only one multiple representation problem. Another case where data is represented more than once is investigated in [14,16]: 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. Updates have to be made in all copies. When the time for an update is high, this is an unfavorable situation. In this situation, we are better off dedicating one processor the task of maintaining the data structure and broadcasting the actual changes to the other processors. Again we have a situation in which there is a multiple representation of the data. One data structure should allow for updates, and a set of other structures 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 kind of 'preprocessed' form that is easier to handle.

This multiple representation problem is related to the reconstruction problem. In both cases, there is one structure on which 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 structures is somewhat different for both problems. A shadow administration is stored in secondary memory, where it is only possible to replace complete blocks by other ones. The client structures, however, are stored in processors on which computing is possible. This makes it possible to replace much smaller pieces of information than just blocks of some predetermined size.

Most of the techniques for designing shadow administrations can be generalized to this second multiple representation problem. For details, the reader is referred to [14,16].

We finish this paper with some directions for future research. A first direction is to search for other general solutions. By restricting ourselves to special classes of data structures, techniques might exist to design efficient shadow administrations. Also, it would be interesting to have more examples of shadow administrations for specific data structures. For example, in order to apply the general technique of Section 5, shadow administrations are needed for which $C(n)$—the amount of data that is changed in an update—is small.

Another direction is to perform sets of updates, instead of performing each update separately. Again one can study special classes of data structures, or design general techniques.

A very important problem, that we have not considered at all, is the following *optimization problem*: In the reconstruction problem, we often reconstruct the data structure exactly as it was before the information was destroyed. The optimization problem is to reconstruct the structure in such a way that it is 'more balanced' than the destroyed structure was. For example, in case of a binary tree, we can maintain in secondary memory the points represented by the tree. The data structure is reconstructed by building it from these points. Of course, this tree is rebuilt as a perfectly balanced tree. So after reconstruction, the data structure is—in general—more balanced than it was before the information

was destroyed. An interesting research direction is to study this optimization problem. Again, general techniques might exist, and special classes of data structures might admit efficient solutions. An example of a solution to this problem for trie hashing functions is given in [20].

References

1. G.M. Adel'son-Vel'skiĭ, E.M. Landis (1962). An algorithm for the organization of information. *Soviet Math. Dokl. 3*, 1259-1262.
2. A.V. Aho, J.E. Hopcroft, J.D. Ullman (1974). *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.
3. R. Bayer, E.M. McCreight (1972). Organisation and maintenance of large ordered indexes. *Acta Informatica 1*, 173-189.
4. J.L. Bentley (1979). Decomposable searching problems. *Inform. Proc. Lett. 8*, 244-251.
5. J.L. Bentley (1980). Multidimensional divide and conquer. *Comm. of the ACM 23*, 214-229.
6. M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, R.E. Tarjan (1973). Time bounds for selection. *J. Comput. System Sci. 7*, 448-461.
7. N. Blum (1986). On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM J. Comput. 15*, 1021-1024.
8. Y.T. Ching, K. Mehlhorn, M. Smid. *Dynamic Deferred Data Structuring.* To appear in: Inform. Proc. Lett.
9. R.M. Karp, R. Motwani, P. Raghavan (1988). Deferred data structuring. *SIAM J. Comput. 17*, 883-902.
10. G.S. Lueker (1978). A data structure for orthogonal range queries. *Proc. 19-th Annual IEEE Symp. on Foundations of Computer Science*, 28-34.
11. J. Nievergelt, E.M. Reingold (1973). Binary search trees of bounded balance. *SIAM J. Comput. 2*, 33-43.
12. M.H. Overmars (1983). *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin.
13. M.H.M. Smid (1989). *Dynamic Deferred Data Structures*, ITLI Prepublication Series CT-89-01, Department of Mathematics and Computer Science, University of Amsterdam.
14. M.H.M. Smid (1989). *Dynamic Data Structures on Multiple Storage Media*, Ph.D. Thesis, University of Amsterdam.
15. M.H.M. Smid (1990). A data structure for the union-find problem having good single-operation complexity. *Algorithms Review* (Newsletter of the ESPRIT II ALCOM Project) *1*, 1-11.
16. M.H.M. Smid, M.H. Overmars, L. Torenvliet, P. van Emde Boas (1989). Maintaining multiple representations of dynamic data structures. *Information and Computation 83*, 206-233.
17. M.H.M. Smid, L. Torenvliet, P. van Emde Boas M.H. Overmars (1989). Two models for the reconstruction problem for dynamic data structures. *J. Inform. Process. Cybernet. EIK 25*, 131-155.
18. J.A. La Poutré (1989). *Lower Bounds for the Union-Find Problem and the Split-Find Problem on Pointer Machines*, Report RUU-CS-89-21, Department of Computer Science, University of Utrecht.

19. R.E. TARJAN (1975). Efficiency of a good but not linear set union algorithm. *J. of the ACM 22*, 215-225.
20. L. TORENVLIET, P. VAN EMDE BOAS (1983). The reconstruction and optimization of trie hashing functions. *Proc. 9-th International Conf. on Very Large Databases*, 142-156.