Maintaining multiple representations of dynamic data structures

M.H.M. Smid, M.H. Overmars, L. Torenvliet, P. van Emde Boas

RUU-CS-88-27
August 1988

Rijksuniversiteit Utrecht

Vakgroep Informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Maintaining multiple representations of dynamic data structures

M.H.M. Smid, M.H. Overmars, L. Torenvliet, P. van Emde Boas

RUU-CS-88-27
August 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Maintaining multiple representations of dynamic data structures

M.H.M. Smid, M.H. Overmars, L. Torenvliet, P. van Emde Boas

Technical Report RUU-CS-88-27
August 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

Maintaining Multiple Representations of Dynamic Data Structures

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

June 1988

Abstract

The problem of maintaining multiple representations of dynamic data structures is investigated: Suppose there are a number of processors, each having the same data structure (the so-called client structure). Then the problem arises of how to maintain these structures efficiently under insertions and deletions. We propose a strategy in which we store a central data structure in one processor, to which all other processors are connected. Updates are first 'preprocessed' in the central structure. Then information obtained in this central update is sent to the client structures, where it is used to adapt these structures. By sending appropriate information, each client structure can be updated more efficiently than by just directly performing the update. We present several solutions to this multiple representation problem. For example, we show that a client structure can be updated in time proportional to the size of the changes in this structure.

1 Introduction

The design of efficient data structures for solving different types of 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

*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, Logic and Computation Theory Group, University of Amsterdam, Nieuwe Achtergracht 166, 1018 WV Amsterdam, The Netherlands.

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 e.g. Overmars [9]).

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 one structure. In many situations, however, we store the structure more than once and have a *multiple representation* of the data structure. For example, normally a data structure that is stored in main memory will be stored in secondary memory as well, because system errors or program errors might otherwise destroy the information. Such a *shadow administration* does not have to support the same operations as the main structure. Only insertions and deletions have to be performed. Hence, it might be advantageous to structure it in a different way. (See Smid et al. [13] for examples of such shadow administrations.)

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. When the update time is high this is an unfavourable situation. We'd better dedicate 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 structure. 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 kind of 'preprocessed' way that might be easier to handle.

In this paper we study such multiple representations of data structures. The one structure that performs the updates will be called the *central structure*. The other structures that allow for queries are the *client structures*. We study how to organize the central structure for different types of query problems, how to structure the client structures that sometimes can store less information, and what type of information has to be sent from the central structure to the clients. It will be shown that, after 'preprocessing' an update by the central structure, the clients can often perform the update more efficiently. Also, in some situations the client structures can be smaller than the central structure.

Besides possible practical applications, the results give insight in which parts of data structures are only necessary for performing updates and can, hence, be removed in the client structures. The results also show what portions of data structures are actually changed when performing updates. This might have applications in storing dynamic data structures in write-once memories (e.g. optical disks).

The paper is organized as follows. In Section 2 we describe the general framework we use to describe solutions for the multiple representation problem, and we

introduce complexity measures to express the efficiency of solutions. In Section 3, we study binary trees as our first example where the client structures store less information than the central structure. It is also shown that we can gain a constant factor in the update time of the client structures. In Section 4, we consider order decomposable set problems. We show that in several cases the client structures can be of size asymptotically less than the central structure. In Section 5 we present some simple techniques for decomposable searching problems. These techniques lead to solutions in which the client structures can be updated very fast.

Since in general an update will change only a small part of a data structure, it might be possible to send only those parts of the structure that have been changed. In [10,11] the problem of how to partition a range tree into parts of small size, such that an update changes only a few parts, is studied. The techniques developed there can of course be applied here. In that paper, however, it is assumed that the partition is stored in secondary memory, which is supposed to be divided into blocks of some predetermined size. Furthermore, the only allowed operation in secondary memory is to replace a block by another one. So in secondary memory no computing is possible. Since in our case the client structures are stored in an environment where computing is possible, we can replace much smaller pieces than just parts of some predetermined size. Hence we can send only those pieces that actually have changed. We will consider this idea thoroughly in Section 6, where we will show that this is indeed possible. (Here the main problem is that we want to avoid searching in the client structures—which might take a lot of time—in order to determine the positions where the data structure has to be changed.) The results of Section 6 are illustrated in Section 7 with some examples. We show e.g. that we can maintain a class of range trees, such that the central structure needs $O((\log n)^d)$ time for an update, whereas each client structure can be updated in $O((\log n)^{d-1})$ time. We finish the paper in Section 8 with some concluding remarks.

2 The general approach

In this section we shall give a precise statement of our problem, and we introduce the model in which solutions will be given. We also give complexity measures to express the efficiency of the solutions.

There is a network of processors, the *clients*, each having its own memory. Each of these clients contains the same data structure DS—the so-called *client structure*—solving some searching problem. Now each client uses its structure DS to solve queries. Updates have to be performed in all the client structures. In order to be able to perform these updates fast, we store in one of the processors a data structure DS', the *central structure*. Now an update is performed as follows. We first perform the update in the central structure DS'. During this update

we (hopefully) obtain information that makes it possible to update the client structures more efficiently than by just directly updating them. Then we send information about the update through the network to the clients, and using this information each client adapts its structure DS. We express the complexity of an update of the client structures by the amount of data that is transported to each client, and by the amount of computing time that the client structure needs to perform the update.

Note that we have introduced a multiple representation of the data. We have a number of copies of the same data structure DS, solving some searching problem. Furthermore, there is a data structure DS', that is used to 'preprocess' updates, such that the client structures can be updated more efficiently. On the client structures, queries and preprocessed updates are performed, whereas on the central structure only updates are carried out. Later on we shall see that it is not necessary for the client structure and the central structure to be identical. Therefore we use different notations for these structures.

The complexity of the client structure DS is expressed by the following functions (n is the number of objects represented by the structure):

- $S(n)$: the amount of space needed to store the structure DS,
- $Q(n)$: the time required to answer a query using DS,
- $F(n)$: the amount of data that is transported to DS in an update,
- $G(n)$: the amount of computing time needed to update DS, using the information received from the central structure.

Note that $G(n) = \Omega(F(n))$, since a client receives an amount of $F(n)$ data, and it has to store it somewhere. Also the total time needed to update the structure DS is $O(F(n) + G(n)) = O(G(n))$.

The complexity of the central structure DS' is given by the usual measures, and they are denoted by:

- $S'(n)$: the amount of space used by DS',
- $I'(n)$: the time needed to insert an object into DS',
- $D'(n)$: the time needed to delete an object from DS',
- if the insertion and deletion times are equal, we denote this common update time by $U'(n)$.

(There is no query time here, because on the central structure no queries are performed.)

The problem to be investigated in this paper is the following. We are given a searching problem. The main goal is to design a client structure DS for this

searching problem, such that when an update is given in some preprocessed form, this update can be performed efficiently. Ideally, the size of this preprocessed form and the time to perform the update using this information—i.e. the values of $F(n)$ and $G(n)$—are much smaller than the time needed to perform the update directly in the structure. A second goal is to design a central structure DS' in which the updates can be preprocessed efficiently. In this paper, however, we shall emphasize the design of the client structure DS.

3 Binary trees

Suppose that the client structures have to solve the member searching problem. A well known dynamic data structure for this problem is a balanced binary search tree, e.g. an AVL-tree, or a BB[α]-tree. Such a tree allows member queries and insertions/deletions to be performed in $O(\log n)$ time, if n is the number of objects stored in the tree. Internal nodes of these trees contain balance information. E.g. in an AVL-tree each internal node contains the difference of the longest path in its left subtree and the longest path in its right subtree (which is -1, 0 or 1). If an object is inserted in or deleted from the tree, all nodes are determined that do not satisfy the balance property anymore, and then by a local restructuring technique (mostly single and double rotations), balance is restored in these nodes. Clearly, this balance information is only used to update the tree; in case member queries are performed, this information is superfluous.

So take a class of balanced binary search trees, that can be maintained by means of single and double rotations. We consider these trees as leaf search trees, i.e. the objects are stored in the leaves. Let T' be a tree in this class, and let T be a copy of T' without the balance information in its nodes. This tree T' will be the central structure, in which updates are preprocessed, and the tree T will be the client structure. Clearly, the tree T contains enough information to allow member queries to be carried out in logarithmic time.

Suppose object p is to be inserted or deleted in the structures. Then we first insert or delete p in the central structure T'. This gives us a path in T', from the root to an appropriate leaf, along which (possibly) rotations have been carried out. We encode this path by a string $s = (r_1, b_1, r_2, b_2, \ldots, r_k, b_k)$, where k is the length of the path. Starting at the root of the tree, r_1 indicates what kind of rotation, if any, has to be performed there. That is, r_1 contains information whether a left single rotation, a right single rotation, etc. has to be carried out, or that no restructuring operation is necessary. Next, b_1 tells us whether the next node on the path lies to the left or to the right of the root. Then r_2 tells us what kind of rotation has to be carried out in the second node of the path, and b_2 says in which direction the path proceeds, and so on. Note that $O(k) = O(\log n)$ bits are sufficient to represent the string s. Now we send to each client structure the

object p together with information whether it has to be inserted or deleted, and the string s. Using p and s, the client structures T are updated. Note that we know exactly which path in T we have to walk down, and where on this path restructuring operations have to be carried out. So we do not have to decide in each node—by means of a comparison of p with the value stored in one of the two sons of the current node—in which direction to proceed. Hence this will save for each client structure $O(\log n)$ comparisons in the update procedure.

The complexity of this solution is as follows. The central structure has size $O(n)$, and an update takes $O(\log n)$ time. Each client structure has also size $O(n)$. In this last bound, however, the constant factor will be smaller. Member queries can be solved in the client structures in $O(\log n)$ time. To perform an update, an object p and a bitstring s of length $O(\log n)$ have to be sent to the client structures, and for each of these structures $O(\log n)$ computing time is needed to adapt it. Again the constant factor is smaller than in the update time of the central structure, since for the client structures we save $O(\log n)$ comparisons. So at the cost of a slight increase in the amount of data that is transported to the client structures—by sending an additional string of $O(\log n)$ bits—we have decreased the constant factors in the complexity bounds for the client structures, compared to the constants in the bounds of the central structure.

Remark that the client structures can be used for solving other searching problems, as long as the balance information of the nodes is not necessary. Examples are the one-dimensional *range searching problem*, where we are given a range $[a : b]$, and where we have to report all points lying in this range. Such a range query can be answered, without needing balance information of the nodes, in $O(\log n + t)$ time, where t is the number of points in the range. Another example is the one-dimensional *nearest neighbor searching problem*. Here we are given a point p, and we have to report the point in the tree that is closest—in absolute value—to p. Clearly, such a query can be answered, again without using balance information, in $O(\log n)$ time.

4 Order decomposable set problems

In [9] a class of so-called order decomposable set problems has been defined. In a *set problem* we are given a set of objects, and we are asked some question about this set. To be 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 in the d-dimensional euclidean space, and we are asked to compute the convex hull of S. Here T_1 is the set of all points in d-dimensional space, and T_2 is the set of all convex polytopes.

In this section we want to solve the problem of maintaining the answer to a set problem under insertions and deletions of objects. We restrict ourselves to

set problems, the answers of which can be merged efficiently. That is, once the answers for two—in some way 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 can 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 $C(n)$-order decomposable, if there is an order ORD on T_1, and a function $\Box : T_2 \times T_2 \rightarrow T_2$, 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*

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

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

For example, as was shown by Preparata and Hong [12], the three-dimensional convex hull problem is $O(n)$-order decomposable, where ORD is the order according to x-coordinate.

Let PR be a $C(n)$-order decomposable set problem. We briefly recall a dynamic data structure solving PR (for details, see [9]). Let S be a set of cardinality n, for which we want to maintain the answer to PR. We store the objects of S, ordered according to ORD, in the leaves of a BB[α]-tree. Internal nodes of this tree contain information to guide searches. Also, each internal node v of this binary tree contains the answer to PR of the objects of S that are in the subtree of v. That is, if S_v is the set of objects in the subtree of v, then node v contains a representation of $PR(S_v)$. Hence, the root of the tree contains $PR(S)$, the answer to PR of the entire set S. The following theorem gives the complexity of this dynamic data structure. For a proof, see [9] (there it is also shown how the amount of space can be reduced without affecting the update complexity, if the sizes of the answers $PR(S_v)$ are large, and for certain values of $C(n)$).

Theorem 1 *For a $C(n)$-order decomposable set problem, there exists a dynamic data structure of size $S(n)$ and update time $U(n)$, given by:*

1. $$S(n) = \begin{cases} O(n) & \text{if } C(n) = O(n^\varepsilon) \text{ for some } 0 < \varepsilon < 1, \\ O(C(n)) & \text{if } C(n) = \Omega(n^{1+\varepsilon}) \text{ for some } \varepsilon > 0, \\ O(C(n) \log\log n) & \text{if } C(n) \text{ is at least linear,} \\ O(n + C(n) \log n) & \text{otherwise.} \end{cases}$$

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

Note that the data structures presented here have the property that just a small part of the structure is used for query answering—the answer to the problem is

stored in the root of the tree—whereas the rest of the structure is only used to update this answer efficiently.

Therefore, we take for the client structures the answer $PR(S)$ to the set problem for the entire set S, and we take for the central structure the full dynamic data structure. Updates are first performed on the central structure. Then we replace each old client structure by the new answer to the set problem. The result is given in the following theorem. (We use the notations introduced in Section 2.)

Theorem 2 *For a $C(n)$-order decomposable set problem, there exists a client structure, that maintains the answer to the set problem, with complexity*

1. $S(n) = O(PR(n))$,
2. $F(n) = O(PR(n))$,
3. $G(n) = O(PR(n))$,

where $PR(n)$ is the size of the answer to the set problem for a set of n objects.

Proof. The proof follows from the above discussion. □

It follows from these two theorems, that for many values of $C(n)$, the client structures have asymptotically lower complexity than the central structure has. For example, in the three-dimensional convex hull problem—which is $\Theta(n)$-order decomposable—the central structure has size $O(n \log\log n)$, whereas the client structures have size only $O(n)$.

5 Decomposable searching problems

A *searching problem* can be seen as a mapping $PR : T_1 \times P(T_2) \rightarrow T_3$, where T_1, T_2 and T_3 are sets of objects. For example, in the member searching problem, $T_1 = T_2$, $T_3 = \{\text{true}, \text{false}\}$, and $PR(x, S) = (x \in S)$.

In this section we consider so-called decomposable searching problems, as introduced by Bentley [1]. For this class of searching problems a query for a set can be answered efficiently by merging the answers for a partition of the set.

Definition 2 *A searching problem $PR : T_1 \times P(T_2) \rightarrow T_3$ is called* decomposable, *if there is a function $\Box : T_3 \times T_3 \rightarrow T_3$, such that for each partition $S = A \cup B$ of any subset S of T_2, and for each query object x in T_1, we have*

$$PR(x, S) = \Box(PR(x, A), PR(x, B)),$$

where the function $\Box$ can be computed in constant time.

For example, the member searching problem is decomposable with $\square = \vee$. Another example is the *orthogonal range searching problem.* Here we are given a set S of points in d-dimensional space, and an axis-parallel hyperrectangle $([a_1 : b_1], [a_2 : b_2], \ldots, [a_d : b_d])$, and we have to report all points $p = (p_1, \ldots, p_d)$ in S, such that $a_1 \leq p_1 \leq b_1, \ldots, a_d \leq p_d \leq b_d$. This problem is decomposable with $\square = \cup$. Note that since we require the sets A and B to be disjoint, we can take the union of $PR(x, A)$ and $PR(x, B)$ in constant time.

A number of techniques have been developed to design dynamic data structures for decomposable searching problems. It turns out that especially in the case where only insertions are performed efficient structures can be designed. See Bentley [1], Bentley and Saxe [2] or Overmars [9].

Let PR be a decomposable searching problem, and let DS be a dynamic data structure solving PR. We consider the case in which only insertions are performed. Let $S(n)$ be the size of the structure DS, and let $Q(n)$ be the query time of DS. We assume that $S(n)/n$ and $Q(n)$ are non-decreasing, and that $S(n)$ and $Q(n)$ are smooth.

To maintain a multiple representation for PR we proceed in the following way: Let the client structure consist of a copy of the structure DS, together with a list of objects. The central structure consists of the structure DS. Initially, the list of objects in the client structure is empty, and all structures DS are up to date. Let n be the initial number of objects. Consider an insertion of an object p. First we insert p in the central structure. If p is already present, then nothing has to be done (note that in this case the client structures do not have to know that anything happened). If p is a new object, we add it to the list of each client structure. After $Q(n)$ objects are inserted in this way—hence each client structure contains a list of $Q(n)$ objects—a copy of the central structure—which is up to date—is sent to the clients. Each old client structure is then replaced by this new structure, and the list of objects is initialized again as an empty list. If m is the number of objects that are present after these $Q(n)$ insertions, we repeat this procedure, now with a sequence of $Q(m)$ insertions.

Queries are solved in a client structure as follows. First we query the data structure DS. Next we query the at most $Q(n)$ objects in the list of most recently inserted objects, by considering each of them separately. Then all answers obtained are merged using the function $\square$. (Note that all objects in the list are different, and are not present in the data structure DS.)

Theorem 3 *Let DS be a data structure for a decomposable searching problem PR, of size $S(n)$ and query time $Q(n)$. There exists a client structure solving PR, with performances:*

1. *The size of the client structure is bounded by $O(S(n))$.*
2. *$F(n) = O(S(n)/Q(n))$ on the average, for an insertion.*

3. *$G(n) = O(S(n)/Q(n))$ on the average, for an insertion.*

4. *The query time of the client structure is bounded by $O(Q(n))$.*

Proof. The client structure consists of a copy of the data structure DS as it is at the beginning of a sequence of insertions, together with a list containing the—at most $Q(n)$—insertions performed so far. Insertions and queries are carried out as described above. The size of the client structure is bounded by the size of DS and by the number of objects in the list. Let N be the number of objects that are currently present, and let n be the number of objects that were present at the beginning of the sequence of insertions. Then the size of the client structure is bounded by $O(S(n) + Q(n)) = O(S(N))$: Because for a decomposable searching problem obviously $Q(n) = O(n)$, and since $S(n)/n$ is non-decreasing, we have $Q(n) = O(S(n))$. Finally, since $n \leq N \leq n + Q(n) = O(n)$, and since $S(n)$ is smooth, the bound on the space complexity follows. In a sequence of $Q(n)$ insertions, the total amount of data that is transported to a client structure, is bounded by $O(Q(n)+S(n)) = O(S(n))$. Hence the average amount of data that is transported for an insertion is $O(S(n)/Q(n))$. The total computing time for $Q(n)$ insertions into a client structure, is also bounded by $O(Q(n) + S(n))$, since a new object can be inserted to the list in constant time, and since it takes $O(S(n))$ time to receive and write a data structure of size $S(n)$. Hence $G(n) = O(S(n)/Q(n))$ on the average for an insertion. Finally, the query time of the client structure is bounded by $O(Q(n))$, because the structure DS can be queried in $Q(n)$ time, and by the definition of a decomposable searching problem, the objects in the list can be queried in $O(Q(n))$ time. □

In the above theorem, the insert complexity for the client structures is an average case complexity. We shall show now how these bounds can be turned into worst case bounds. The idea is to spread out the transport of the large data structure over a number of insertions. In the sequel we assume that if object p is to be inserted, it is not present yet. As we saw already, if the object is present, the client structures do not have to know that anything happened. We denote by DS_n the data structure DS representing a set of n objects.

The client structure consists of a data structure DS, and two lists of objects. The central structure consists of a structure DS and one list of objects. Let k be the initial number of objects. Then both the client structure and the central structure contain an up to date data structure DS_k and all lists are empty. During the first $Q(k)$ insertions, we add the new objects to one of the lists of the client structures (each time we add it to the same list). Furthermore, all these insertions are performed in the central structure DS.

Hence after the first $Q(k)$ insertions, the client structures consist of a data structure DS_k, representing the objects that were initially present, a list of the $Q(k)$ most recently inserted objects, and an empty list. The central structure

consists of an up to date structure $DS_{k+Q(k)}$, and an empty list.

Let $n = k + Q(k)$, i.e. n is the number of objects that are currently present. Consider a sequence of $Q(n)$ insertions. During the first $Q(n)/2$ insertions, we add the new objects to the initially empty lists of the client structures, and we send the central structure $DS_{k+Q(k)} = DS_n$ to the clients: Each update we send a part of DS_n of size $O(S(n)/Q(n))$. Then, after these $Q(n)/2$ insertions, each client structure contains a data structure DS_n, and a list of the $Q(n)/2$ most recently inserted objects. Now we throw away the old client structures DS_k and we set the old list of $Q(k)$ inserted objects to the empty list. In the central structure we add the $Q(n)/2$ new objects to the list. Note that the central structure DS_n cannot be affected during these insertions.

The final $Q(n)/2$ insertions are performed as follows. The new objects are added to the non-empty list of the client structure. In the central structure, we perform in each update the current one, and one update from the list of updates. (Note that the order in which we perform the updates in the central structure does not matter, since all updates are insertions. If, however, also deletions were possible, the updates had to be carried out in chronological order. See Section 6.4.) Afterwards the list of the central structure is set to the empty list.

So after the entire sequence of $Q(n)$ updates, the client structure contains a data structure DS_n, a list of the $Q(n)$ most recently inserted objects, and an empty list. The central structure consists of an up to date structure $DS_{n+Q(n)}$ and an empty list. Hence we are in the same situation as $Q(n)$ updates ago, and we can continue in the same way.

Queries in a client structure are solved, by querying the data structure DS, and by walking along the two lists of objects. Then using the function $\square$, the answers are merged to get the final answer to the query.

The result is given in the following theorem.

Theorem 4 *Let DS be a data structure for a decomposable searching problem PR with worst case complexity $S(n)$, $I(n)$ and $Q(n)$. There exists a client structure solving PR, with performances:*

1. *The size of the client structure is bounded by $O(S(n))$.*
2. *$F(n) = O(S(n)/Q(n))$ in the worst case, for an insertion,*
3. *$G(n) = O(S(n)/Q(n))$ in the worst case, for an insertion,*
4. *The query time of the client structure is bounded by $O(Q(n))$.*

Furthermore, the size and the insertion time of the central structure are bounded by $O(S(n))$ and $O(I(n))$.

Proof. It follows from the above that in each insertion we send an amount of $O(S(n)/Q(n)) + O(1) = O(S(n)/Q(n))$ data, and for each client structure we have

to spend $O(S(n)/Q(n)) + O(1) = O(S(n)/Q(n))$ time to receive and write this data. Hence both $F(n)$ and $G(n)$ are bounded by $O(S(n)/Q(n))$ in the worst case. Also, the query time of the client structure is $O(Q(n))$. Clearly, the performances for the central structure are increased by at most a constant factor. □

There are other techniques to obtain efficient solutions to the multiple representation problem. We can e.g. consider sequences of more than $Q(n)$ insertions. Then the most recently inserted objects are stored in a small data structure, to ensure that the query time remains $O(Q(n))$. In this way the values of $F(n)$ and $G(n)$ can be decreased. This idea will be worked out below.

Let PR be a decomposable searching problem, and let DS be a dynamic data structure solving PR. The size and the query time of DS are denoted by $S(n)$ and $Q(n)$. As before, we assume that $S(n)/n$ and $Q(n)$ are non-decreasing, and that $S(n)$ and $Q(n)$ are smooth.

Let $f(n)$ be an integer function, such that $Q(n) < f(n) < n$. The client structure consists of two data structures DS_1 and DS_2, and a list of objects. The central structure contains the structures DS_1 and DS_2.

Initially, all structures DS_1, and the lists in the client structures, are empty. Each structure DS_2 stores the n objects that are present at this moment.

Consider a sequence of $f(n) - 1$ insertions. We insert the new objects in the central data structure DS_1. In the client structure we add the new objects to the list. Every $Q(n)$-th insertion, the central structure DS_1 as it is that moment is sent to the client structure (where it replaces the old DS_1), and the list of objects is set to the empty list. Hence during these $f(n) - 1$ insertions, the client structure consists of a list of at most $Q(n)$ objects, and of two data structures DS_1 and DS_2. The structure DS_1 represents at most $f(n)$ objects. At each moment, the objects represented by these three structures form a partition of all the objects that are present at that moment.

In the $f(n)$-th insertion, we build a new structure DS_2 storing all objects that are present at this moment, and send it to the clients. Also, all structures DS_1 and all lists are made empty. If m is the number of present objects at this moment we repeat this procedure, now with a sequence of $f(m)$ insertions.

Clearly, the size and the query time of the client structure remain $O(S(n))$ and $O(Q(n))$. Furthermore, the average values of $F(n)$ and $G(n)$ are both bounded by $O(S(f(n))/Q(n) + S(n)/f(n))$.

We can generalize this solution as follows. Let k be a positive integer, and let $f_i(n)$ be integer functions, $i = 0, 1, \ldots, k$, such that $Q(n) = f_0(n) < f_1(n) < f_2(n) < \ldots < f_{k-1}(n) < f_k(n) = n$. Then the client structure contains a collection of data structures DS_i, $i = 1, 2, \ldots, k$, and a list of at most $Q(n)$ objects. The central structure contains the structures DS_i, $i = 1, 2, \ldots, k$. Each DS_i will represent at most $f_i(n)$ objects. Initially, all structures $DS_1, \ldots, DS_{k-1}$, and all lists, are empty. Each structure DS_k stores the n objects that are present at this

moment.

Consider a sequence of $f_{k-1}(n)$ insertions. In the j-th insertion, we do the following. If there is an i, $0 \leq i \leq k-1$, such that $j \equiv 0 \bmod f_i(n)$, determine the maximal such i. Then build a new structure DS_{i+1}, storing all objects that were present in the old central structures $DS_1, \ldots, DS_{i+1}$, and add it to the central structure. Also, the old central structures $DS_1, \ldots, DS_i$ are made empty. Next, send this new structure DS_{i+1} to the clients, where it replaces the old DS_{i+1}. Finally, all structures $DS_1, \ldots, DS_i$ and the lists are made empty. If there is no i such that $j \equiv 0 \bmod f_i(n)$, add the new object to the list of the client structures, and insert the new object in the central structure DS_1.

It is not difficult to see, that indeed each DS_i represents at most $f_i(n)$ objects, and that the list in the client structure contains at most $Q(n)$ objects. Also, each DS_i is sent to the clients at most once every $f_{i-1}(n)$ insertions.

After these $f_{k-1}(n)$ insertions, all structures $DS_1, \ldots, DS_{k-1}$, and all lists, are empty again, and each structure DS_k stores the objects that are present at this moment. (Note that in the $f_{k-1}(n)$-th insertion, the maximal value of i in the above update procedure is $k-1$.) So we can proceed in the same way, now with a sequence of $f_{k-1}(m)$ insertions, where m is the current number of objects.

In this way the average values of $F(n)$ and $G(n)$ are bounded by

$$\frac{S(f_1(n))}{Q(n)} + \frac{S(f_2(n))}{f_1(n)} + \cdots + \frac{S(f_{k-1}(n))}{f_{k-2}(n)} + \frac{S(n)}{f_{k-1}(n)}.$$

Since we assumed that $S(n)/n$ is non-decreasing, it follows that this sum is bounded above by

$$\frac{S(n)}{n}\left(\frac{f_1(n)}{Q(n)} + \frac{f_2(n)}{f_1(n)} + \cdots + \frac{f_{k-1}(n)}{f_{k-2}(n)} + \frac{n}{f_{k-1}(n)}\right).$$

Now take $f_i(n) = \left\lceil n^{i/k}(Q(n))^{1-i/k}\right\rceil$. Then the average values of $F(n)$ and $G(n)$ are bounded above by

$$k\,\frac{S(n)}{n}\left(\frac{n}{Q(n)}\right)^{1/k}.$$

In the same way as before these average case bounds can be turned into worst case bounds. The result is expressed in the following theorem, the proof of which is left to the reader.

Theorem 5 *Let DS be a data structure for a decomposable searching problem PR with complexity $S(n)$ and $Q(n)$. Then for each positive integer k there exists a client structure solving PR, with performances:*

1. *The size of the client structure is bounded by $O(S(n))$.*
2. *$F(n) = O(k\,\frac{S(n)}{n}(\frac{n}{Q(n)})^{1/k})$ in the worst case, for an insertion,*

3. $G(n) = O(k \frac{S(n)}{n}(\frac{n}{Q(n)})^{1/k})$ *in the worst case, for an insertion,*

4. *The query time of the client structure is bounded by* $O(k \times Q(n))$.

We shall illustrate this result with an example. In the *nearest neighbor searching problem*, we are given a set S of n points in the plane, and a query point p, and we are asked to find the point in S that is closed to p with respect to the euclidean distance. Clearly, this problem is decomposable. There exists a data structure for this problem of size $O(n)$ such that queries can be solved in $O(\log n)$ time, see e.g. Kirkpatrick [4]. Applying Theorem 5, we obtain

Theorem 6 *Let k be a positive integer. For the nearest neighbor searching problem in the plane, there exists a client structure, with performances:*

1. *The size of the client structure is bounded by* $O(n)$.

2. $F(n) = O(k\,(\frac{n}{\log n})^{1/k})$ *in the worst case, for an insertion,*

3. $G(n) = O(k\,(\frac{n}{\log n})^{1/k})$ *in the worst case, for an insertion,*

4. *The query time of the client structure is bounded by* $O(k \log n)$.

It is clear that the technique presented in this section only allows insertions to be carried out. In some cases, however, deletions can also be performed. For example, we can restrict ourselves to a subclass of the decomposable searching problems, the so-called *decomposable counting problems*. Roughly speaking, a decomposable counting problem is a decomposable searching problem where the function $\square$ has an inverse, that can also be computed in constant time (see [2,9]). An example is the *orthogonal range counting problem*. Here we are given a set S of points in the plane, and an axis-parallel query rectangle, and we are asked how many points of S are in the rectangle.

For decomposable counting problems we can design a full dynamic data structure by maintaining two structures. In one structure new objects are inserted, whereas a deletion is performed by inserting it into the other structure. A query is solved by querying the two structures, and by 'subtracting' the two obtained answers from each other, using the inverse of the function $\square$.

For decomposable counting problems, the following analogue of Theorem 5 can be proved.

Theorem 7 *Let DS be a data structure for a decomposable counting problem PR with complexity $S(n)$ and $Q(n)$. Then for each positive integer k there exists a full dynamic client structure solving PR, with performances:*

1. *The size of the client structure is bounded by* $O(S(n))$.

2. $F(n) = O(k \frac{S(n)}{n}(\frac{n}{Q(n)})^{1/k})$ *in the worst case,*

3. $G(n) = O(k \frac{S(n)}{n}(\frac{n}{Q(n)})^{1/k})$ *in the worst case,*

4. *The query time of the client structure is bounded by* $O(k \times Q(n))$.

6 A general technique

6.1 Introduction

Consider again our strategy with respect to the member searching problem of Section 3. In this solution, we send in each update a string of $O(\log n)$ bits to the client structures, where the string contains an encoding of the path to the node where the update is carried out, together with information about what kind of rotations have to be performed. In order to update the client structure, we follow the path, insert or delete the object, and perform the rotations. Clearly, this procedure takes $O(\log n)$ time. If we consider, however, how many nodes in the tree are changed in this update, we see that $O(1)$ of them are changed due to the insertion or deletion, and the rest of them are changed due to rotations. Therefore, if $O(1)$ rotations are carried out, only $O(1)$ nodes of the tree are changed. (Note that a client structure does not contain balance information. Otherwise, $O(\log n)$ nodes were changed.) So if we could avoid to walk down the path, it could be possible to update the client structure in only $O(1)$ time.

The solution is to send to the client structures the inserted or deleted object, together with the positions in the tree where changes—and what kind of changes—have to be carried out. Since there are binary trees that can be maintained in logarithmic time with only $O(1)$ rotations in the worst case (see [7,8]), this will give us a solution where the client structures can be maintained in constant time.

This is the main idea of the general technique that will be worked out in this section. We will achieve our result in a number of steps. First we give a solution in case the data structures do not exceed some given size. Next we extend this solution to a general one having a low average case complexity. Then we turn these average case bounds into worst case bounds.

Let PR be a searching problem, and let DS (resp. DS') be the corresponding client structure (resp. central structure). The performances of DS are denoted by $S(n)$ and $Q(n)$, and those of DS' by $S'(n)$ and $U'(n)$ (see Section 2 for these notations). 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. For example, if DS' is a balanced binary tree, then we can take for DS this tree without the balance information of the nodes. Updates are performed as before. That is, first the central structure DS' is adapted, then information is sent to the client structures, and finally the client structures DS are adapted. Let $C(n)$ denote the amount of data that is changed in the client structure DS in an update. We assume that all these complexity measures and $S(n)/n$ are non-decreasing and smooth.

We shall transform this multiple representation into another one, such that each transformed client structure has size $O(S(n))$, update complexity $F(n) = O(C(n))$

and $G(n) = O(C(n))$, and in which queries can be solved in $O(Q(n))$ time. The idea is to send in each update only the changes of the client structure DS. In order to avoid searching for the positions in the client structure where the changes have to be carried out, we also send these positions. Therefore, we implement the data structures as arrays. (We assume that our processors are random access machines, the memories of which are modelled as an array. Hence we can indeed implement the data structures as an array.) We take care that each part of DS is stored in the same position in all computers. If such a part has to be changed, we only send the index in the array where this part is stored. Then, in each client structure, we can find in constant time the position where the change has to be carried out. This implementation will be described more precisely in the next section.

We finish this section with the following lemma.

Lemma 1 *The complexity measures introduced above satisfy*

1. $S(n) \leq S'(n)$,
2. $S'(n)/n = O(U'(n))$,
3. $S(n)/n = O(C(n))$.

Proof. Since DS is a substructure of DS', we have $S(n) \leq S'(n)$. We can build the structure DS' by performing n insertions into an initially empty structure, which takes at most $U'(1) + U'(2) + \cdots + U'(n) \leq n \times U'(n)$ time. During these n insertions we have built a structure of size $S'(n)$, and hence we have spent at least $S'(n)$ time. This proves that $S'(n) = O(n \times U'(n))$. In the same way we can build the structure DS. The total amount of data that has changed during n insertions, is at most $C(1) + C(2) + \cdots + C(n) \leq n \times C(n)$. Since at the end there is a structure of size $S(n)$, it follows that $S(n) = O(n \times C(n))$. □

6.2 A fixed size solution

Let N be an integer, the maximal number of objects that can be represented by our data structures. We use in this section—and in the following ones—the notations introduced in Section 6.1.

We have a client structure DS and a central structure DS', and we want to implement these structures as arrays. These 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 location. Since the data structures represent at most N objects, we take a *client array* A of $S(N)$ entries, containing DS, and a *central array* A' of $S'(N)$ entries, containing DS'. If n is the current number of objects, $S(n)$ entries of the client array and $S'(n)$ entries of the central array are occupied. We assume that the first $S(N)$ entries of the central array are identical to those of the client array. Clearly, this can always be achieved.

Finally, we introduce two stacks FE and FE' of free entries. In FE we store those indices of the first $S(N)$ entries of the client array A, that are unoccupied. So the client structure is stored in the client array locations $\{1, \ldots, S(N)\} \setminus \{i | i \in FE\}$. Similarly, the stack FE' contains those indices of the last $S'(N) - S(N)$ entries of the central array A' that are unoccupied. The purpose of these stacks is to perform our own memory management.

Now the transformed client structure consists of the array A, that contains information to answer queries. The transformed central structure consists of the array A', containing in its first $S(N)$ entries the array A, and in the other locations information that is used to preprocess updates. Also, the central structure contains the stacks FE and FE' of free entries.

Suppose we want to insert or delete object p. We assume that there is room in the arrays for a new object. Then we first perform this update in the central structure. If we need new entries, we take them from the appropriate stack FE or FE', and if entries become unoccupied, we put them on the stack where they belong. Clearly, this update procedure takes $O(U'(n))$ time. Next we send all changes on the client structure, i.e. the indices of the entries in the array A that are changed together with the changes themselves. Using this information, each client structure is adapted. Since the indices of the entries that have to be changed in the client array are known, this array can be updated in time proportional to the number of changed entries. So in our notation we have $F(n) = O(C(n))$ and $G(n) = O(C(n))$. Note that the client structures do not need to contain the stack FE of free array indices: the entire memory management is arranged by the central structure. Clearly, at each moment the client structure is up to date and, hence, it can be used to answer queries.

Theorem 8 *Let DS be a client structure solving some searching problem, with complexity $S(n)$, $Q(n)$ and $C(n)$. Let DS' be the corresponding central structure, with complexity $S'(n)$ and $U'(n)$. We can transform these structures into a multiple representation, such that each client structure*

1. *has size $O(S(N))$,*
2. *has a query time bounded by $O(Q(n))$,*
3. *has $F(n) = O(C(n))$,*
4. *has $G(n) = O(C(n))$,*

where N is the maximal number of objects that can be represented by the structures, and n is the current number of objects. Furthermore, the central structure has size $O(S'(N))$, and its update time is bounded by $O(U'(n))$.

Proof. The size of the central structure is bounded by $O(S'(N))$ for the array A', and by $O(|FE| + |FE'|) = O(S'(N))$ for the stacks. Hence the total size of

the central structure is bounded by $O(S'(N))$. The other bounds follow from the above discussion. □

If we know in advance, in some way, that the number of objects does not vary too much, this will be an efficient solution. If, however, the number of objects becomes too large, after a number of insertions, our arrays will become too small. Similarly, after a number of deletions, a large part of the arrays will become empty, and so the amount of space will become too large. In these cases the solution, of course, is to rebuild the structures.

6.3 An efficient average case solution

Suppose that the data structures initially represent n objects. We store each structure in an array that can store a data structure of $\frac{3}{2}n$ objects. In this way there is room in the structures for $n/2$ insertions. So in the notation of the preceding section, we take $N = \frac{3}{2}n$. The client structure consists of the array A of length $S(N)$. The central structure contains the array A' of length $S'(N)$, and the stacks FE and FE', of size at most $S'(N)$. The information is stored in these data structures as in the previous section, and updates are performed in exactly the same way. As soon as the number of objects becomes either $\frac{1}{2}n$ or $\frac{3}{2}n$, we rebuild our data structures. That is, if m is the number of objects at that moment, we build a new array A' and new stacks FE and FE', that are large enough to contain a data structure for $\frac{3}{2}m$ objects, and we send the subarray containing the first $S(\frac{3}{2}m)$ entries of A'—this subarray will be the new client structure A—to the clients, where this new array replaces the old one. Then we proceed in the same way. Note that this rebuilding can be done in $O(S'(n))$ time: we have to put the array A' into a new one, and we have to make new stacks for the new free array locations. This can be done by just walking along the $O(S'(n))$ array entries. (Remark that $n = \Theta(N) = \Theta(m)$.)

Theorem 9 *Let DS be a client structure solving some searching problem, with complexity $S(n)$, $Q(n)$ and $C(n)$. Let DS' be the corresponding central structure, with complexity $S'(n)$ and $U'(n)$. We can transform these structures into a multiple representation, such that each client structure*

1. *has size $O(S(n))$,*
2. *has a query time bounded by $O(Q(n))$,*
3. *has $F(n) = O(C(n))$, on the average,*
4. *has $G(n) = O(C(n))$, on the average.*

The central structure has size $O(S'(n))$, and its average update time is bounded by $O(U'(n))$.

Proof. The bounds on the amount of space used by the structures follow from Theorem 8, and from the fact that N—the maximal number of objects that can be represented—and n—the current number of objects—satisfy $n = \Theta(N)$. Clearly, the query time for a client structure remains $O(Q(n))$. Since the structures are rebuilt at most once every $n/2$ updates, the average values of both $F(n)$ and $G(n)$ are bounded by $O(C(n) + S(n)/n)$, which is $O(C(n))$ by Lemma 1. In the same way it follows from Theorem 8 and Lemma 1, that the average update time of the central structure is bounded by $O(U'(n) + S'(n)/n) = O(U'(n))$. □

6.4 An efficient worst case solution

In this section we assume that the update time $U'(n)$ of the central structure and the amount of data $C(n)$ that an update changes in the client structure are worst case bounds. We shall show how the average case bounds of the preceding section can be made into worst case bounds. The idea is to spread out the construction of the new structures over a number of updates. The technique is related to the global rebuilding technique in [9]

Let m be the number of objects initially represented by the data structures. Let l be an integer, such that $\frac{3}{2}m \leq l \leq 3m$. We first describe the update algorithm for the client structure; later we shall consider the central structure. The client structure consists of the array A of length $S(l)$, as before.

Consider a sequence of $m/2$ updates. (Note that the array A has room for at least $m/2$ new objects.) We split this sequence into 3 phases.

First phase: The first phase consists of the first $m/4$ updates. These are performed as before. That is, the changes of the client structures, together with the positions in the array A where the changes have to be carried out, are sent to them, and using the received information, each client structure is adapted. So after the first phase, the client structures are up to date.

Let m_0 be the number of objects that are present after the first phase, and let $l_0 = 2m_0$. (We use l_0 to estimate the number of objects that are present after the third phase.)

Second phase: The second phase consists of the next $m/8$ updates. These updates are performed as in the first phase. Also, a new client array A_0 is built in the central computer during the first $m/16$ updates of this second phase. This array has length $S(l_0)$, and it stores the client data structure as it was after the first phase. (Later we shall describe how the central computer builds this new array; we now just assume that it is there.) This new array is sent to the clients during the last $m/16$ updates of the second phase. In each update we send an amount of $O(S(l_0)/m) = O(S(m)/m)$, which is bounded by $O(C(m))$ by Lemma 1.

After the second phase, the client structure consists of an up to date array A and an array A_0, containing the client structure as it was after the first phase. We also assume that the central structure contains a list of the updates in the

second phase, i.e. a list containing the $m/8$ objects, and for each object information whether it has to be inserted or deleted.

Third phase: This phase consists of the final $m/8$ updates. These updates are carried out on the up to date client array A, as before. In order to make the new array A_0 up to date, we perform on this array with each update, two updates from the list of updates from the second phase. (Note that these updates have to be performed in chronological order, since the same object can be inserted and deleted several times!) Then we remove the two updates we just carried out from the list, and the actual update is added at the end of the list.

After this final phase, the client array A_0 is up to date, and the old array A is discarded.

So we end with a client structure consisting of an array A_0 of length $S(l_0)$. Let n be the number of objects that are represented by the structures at this moment. If we can show that $\frac{3}{2}n \leq l_0 \leq 3n$, then we are in the same situation as at our starting point, i.e. before the first phase, and hence we can proceed in the same way.

At the beginning the data structures represented m objects, and after the first $m/4$ updates there were m_0 objects. It follows that

$$\frac{3}{4}m \leq m_0 \leq \frac{5}{4}m.$$

After the third phase, i.e. after another $m/4$ updates, there are n objects. Hence

$$m_0 - \frac{1}{4}m \leq n \leq m_0 + \frac{1}{4}m.$$

Clearly, m and n are related by

$$\frac{1}{2}m \leq n \leq \frac{3}{2}m.$$

It follows that

$$l_0 = 2m_0 = \frac{3}{2}m_0 + \frac{1}{2}m_0 \geq \frac{3}{2}m_0 + \frac{3}{8}m = \frac{3}{2}(m_0 + \frac{1}{4}m) \geq \frac{3}{2}n,$$

and

$$l_0 = 2m_0 \leq 2(n + \frac{1}{4}m) \leq 2n + n = 3n,$$

which shows that we are indeed in the same situation as at our starting point.

The central structure consists of two copies of each of the structures A', FE and FE' , and one copy of a list L (we use the notations of the preceding section). All $m/2$ updates are carried out on one of A', FE and FE'. Hence at each moment the central structure contains an up to date data structure. In the second phase,

in each update we add the object together with information whether it has to be inserted or deleted, to the list L.

It remains to describe what happens with the other structures A', FE and FE'. In the first phase the updates are performed on these structures as usual. During the first $m/16$ updates of the second phase we convert them into new structures A'_0, FE_0 and FE'_0. Here A'_0 is an array of length $S'(l_0)$ that will contain the data structure as it is at the beginning of the second phase, and FE_0 and FE'_0 are the corresponding stacks of free entries in this new array. This converting can be performed in $O(S(l_0)) = O(S(m))$ time. In each of the $m/16$ updates we do an amount of $O(S(m)/m)$ of this converting. It follows from Lemma 1 that the update time for the central structure remains $O(U'(n) + S(m)/m) = O(U'(n))$, where n is the current number of present objects.

During the next $m/16$ updates of the second phase, the first $S(l_0)$ entries of the array A'_0—which contain the new client array A_0—are sent to the clients, as described above. Also, the structures A'_0, FE_0 and FE'_0 are copied; each update we do an amount of $O(U'(n))$ work. During the third phase, we perform each update, two updates from the list L, on both copies of each of the structures A'_0, FE_0 and FE'_0, and we add the actual update at the end of L. (Again we remark that the updates have to be carried out in chronological order.) After this third phase, the structures A', FE and FE' are discarded. We end with two copies of each of the structures A'_0, FE_0 and FE'_0. Hence we are in the same situation as before the first phase.

Before we summarize the result, we remark that a client structure contains at any moment an up to date data structure, that can be used to answer queries.

Theorem 10 *Let DS be a client structure solving some searching problem, with worst case complexity $S(n)$, $Q(n)$ and $C(n)$. Let DS' be the corresponding central structure, with worst case complexity $S'(n)$ and $U'(n)$. We can transform these structures into a multiple representation, such that each client structure*

1. *has size $O(S(n))$,*
2. *has a query time bounded by $O(Q(n))$,*
3. *has $F(n) = O(C(n))$, in the worst case,*
4. *has $G(n) = O(C(n))$, in the worst case.*

The central structure has size $O(S'(n))$, and its worst case update time is bounded by $O(U'(n))$.

Proof. The size of the central structure is bounded by $O(S'(n) + n) = O(S'(n))$, where the $O(n)$ term is due to the list of updates. The rest of the proof follows from the above discussion. □

7 Examples

As we have seen in Section 6, we can bound the update time for the client structures by $O(C(n))$, which is the size of the changes in the structure. Hence our goal is to design structures for searching problems for which $C(n)$ is small. It is not important whether the changes can be found efficiently (although this would make the amount of work on the central structure small).

7.1 Binary search trees

Most classes of balanced binary search trees, such as AVL-trees, BB[α]-trees, etc., have the property that in an update $O(\log n)$ rotations are necessary to rebalance them. Hence for such trees, an update changes $O(\log n)$ nodes. Binary trees from the class of αBB-trees, as introduced by Olivié [7,8], however, have the interesting property that they can be maintained in logarithmic time, by at most a constant number of rotations (if $\alpha \in \{\frac{1}{2}, \frac{1}{3}\}$).

So let T be an αBB-tree, where $\alpha \in \{\frac{1}{2}, \frac{1}{3}\}$, without the balance information of the nodes. Suppose T contains a set of n objects in its nodes. In this tree, member queries can be solved in $O(\log n)$ time. By the above mentioned result of Olivié, we can maintain T by means of $O(1)$ rotations. Hence an update changes only $O(1)$ nodes in T. (Note that if the tree would contain balance information, an update would change $O(\log n)$ nodes, since then the balance information would have to be adapted.) Applying Theorem 10, we get

Theorem 11 *For solving the member searching problem, there exists a client structure with complexity*

1. $S(n) = O(n)$,
2. $Q(n) = O(\log n)$,
3. $F(n) = O(1)$,
4. $G(n) = O(1)$.

Furthermore, the central structure has size $O(n)$, and can be maintained in $O(\log n)$ time.

In the solution just given we stored the objects in the nodes of the tree. There are applications, however, in which we want to store the objects in sorted order in the leaves of the tree. Then, in order to be able to search in the tree, we have to store information in the internal nodes to guide these searches (in each node we must decide in some way whether we proceed to the left or to the right son). Suppose we store in each node the maximal element in its subtree. Clearly, we can use this information to solve member queries in time proportional to the longest path in the tree. If we now delete the maximal element in the tree, then in each

node on the rightmost path the search information has to be changed. Therefore, if the tree is balanced, an update changes $O(\log n)$ nodes. So we have to take care what 'search information' we store in the internal nodes.

Suppose now that we store in each internal node v, the maximal element in the left subtree of v. Note that this maximal element is stored in the unique leaf that is reached by making one step to the left in node v, followed by a maximal number (possibly none) of steps to the right. It is not difficult to prove that in this case an update changes $O(1)$ nodes, if we do not rebalance the tree.

So let T be an αBB-tree, containing a set of n elements in sorted order in its leaves, without balance information. Each internal node contains the maximal element in its left subtree. Then, in T member queries can be solved using the search information of the internal nodes in $O(\log n)$ time. Now let $\alpha \in \{\frac{1}{2}, \frac{1}{3}\}$. Then it follows from the above that an update changes only $O(1)$ nodes in T. (Note that the search information in a node is changed iff the maximal element in its left subtree is changed.) Applying Theorem 10, we get

Theorem 12 *For solving the member searching problem, we can take for the client structures a leaf search tree, having complexity*

1. $S(n) = O(n)$,
2. $Q(n) = O(\log n)$,
3. $F(n) = O(1)$,
4. $G(n) = O(1)$.

Furthermore, the central structure has size $O(n)$, and can be maintained in $O(\log n)$ time.

In the next section we shall use αBB-trees to design an efficiently maintainable class of data structures solving the orthogonal range searching problem.

7.2 Range trees

The orthogonal range searching problem, was mentioned already in Section 5. Bentley [1], Lueker [5] and Willard and Lueker [14] designed an efficient data structure for this problem, the so-called *range tree*, that can be maintained in $O((\log n)^d)$ time. In this section, we show that by modifying the balance conditions somewhat, we get a class of d-dimensional range trees, for which an update changes only $O((\log n)^{d-1})$ nodes.

We define range trees as follows. (For the definitions of BB[α]-trees and αBB-trees, we refer the reader to Nievergelt and Reingold [6], Blum and Mehlhorn [3], and Olivié [7,8].)

Definition 3 *Let S be a set of points in the d-dimensional euclidean space. A* d-dimensional range tree *T, representing the set S, is defined as follows.*

1. *If $d = 1$, then T is an αBB-tree, containing the points of S in sorted order in its leaves.*
2. *If $d > 1$, then T consists of a BB[α']-tree, called the* main tree, *containing in its leaves the points of S, ordered according to their first coordinate. Each node v of this main tree contains an* associated structure, *which is a $(d-1)$-dimensional range tree for those points of S that are in the subtree rooted at v, taking only the 2nd to d-th coordinate into account.*

So in our notion of range trees there are two kind of binary trees. The trees representing points in multi-dimensional space belong to the class of BB[α']-trees, and the trees representing one-dimensional points belong to the class of αBB-trees. Note that all trees are used as leaf search trees.

Let T be a d-dimensional range tree, and suppose we want to insert or delete a point p. Then we search with p in the main tree to locate its position among the leaves, and we insert or delete p in all the associated structures we encounter on our search path (if these associated structures are one-dimensional range trees, we apply the update algorithm for αBB-trees using rotations; otherwise we use the same procedure recursively). Next we insert or delete p among the leaves in the main tree, and we walk back to the root. During this walk, we rebalance the main tree: each node that is out of balance is rebalanced by means of rotations. Note that we have to rebuild the associated structures of the nodes that are involved in these rotations, and this will take a lot of time when these structures are large. However, it turns out that the average time to update T in this way, will be low.

The following theorem gives the complexity of range trees. For a proof, we refer the reader to [1,5,9,14].

Theorem 13 *Let S be a set of n points in d-dimensional space. Then a d-dimensional range tree, representing the set S, has size $O(n(\log n)^{d-1})$, and can be built in $O(n(\log n)^{d-1})$ time. In this tree, updates can be performed in time $O((\log n)^d)$ on the average, and orthogonal range queries can be solved in time $O((\log n)^d + t)$, where t is the number of reported answers, without using the balance information stored in the nodes.*

Let T be a d-dimensional range tree for a set of n points, without the balance information. We store in internal nodes of the trees search information as in Section 7.1 (note that our trees are leaf search trees). We take for the one-dimensional structures αBB-trees with $\alpha \in \{\frac{1}{2}, \frac{1}{3}\}$. It was stated that the average update time of T is bounded by $O((\log n)^d)$. We shall show now that the average number of nodes that are changed in T in an update is bounded by $O((\log n)^{d-1})$. Let $C(n,d)$ denote this average number.

Lemma 2 $C(n,d) = O((\log n)^{d-1})$.

Proof. We have seen in Section 7.1 already that $C(n,1) = O(1)$. Let $d > 1$. To perform an update we start in the root of the main tree, and we update its associated structure. This changes on the average at most $C(n, d-1)$ nodes. Then we repeat the same procedure for the appropriate son of the root, which is the root of a range tree for at most $(1-\alpha')n$ points. Hence this changes on the average at most $C((1-\alpha')n, d)$ nodes. If the root of the main tree gets out of balance, we perform a rotation and, hence, we have to rebuild the associated structures of the sons of the root. Since these associated structures are $(d-1)$-dimensional range trees, this changes $O(n(\log n)^{d-2})$ nodes. It was shown by Blum and Mehlhorn [3] that for a proper choice of α' the root of the main tree gets out of balance at most once every $\Omega(n)$ updates. Hence the average number of nodes that are changed due to our visit of the root of the main tree is bounded by $O((\log n)^{d-2})$. It follows that $C(n,d)$ satisfies the following recurrence:

$$C(n,d) \leq C(n,d-1) + C((1-\alpha')n, d) + O((\log n)^{d-2}).$$

This proves the lemma. □

So we have a class of range trees that can be maintained in time $O((\log n)^d)$ on the average, whereas in the structures without balance information an update changes only $O((\log n)^{d-1})$ nodes, also on the average. Hence, by Theorem 9, we have

Theorem 14 *For solving the orthogonal range searching problem, there exists a client structure with complexity*

1. $S(n) = O(n(\log n)^{d-1})$,
2. $Q(n) = O((\log n)^d + t)$, *where t is the number of reported answers,*
3. $F(n) = O((\log n)^{d-1})$ *on the average,*
4. $G(n) = O((\log n)^{d-1})$ *on the average.*

8 Conclusions

We have studied the problem of maintaining multiple representations of dynamic data structures: Suppose there are a number of processors, each containing the same data structure. Then updates have to be performed in all these structures. In order to save time, we first 'preprocess' the update in a central structure. Then we broadcast information about the update to the processors, and using this information each of these processors adapts its structure.

In this way there are two different types of structures. First there are the client structures, that are stored in the processors. These client structures contain information such that queries can be answered efficiently. Also, preprocessed

updates can be carried out fast on these client structures. The other structure is the central structure, in which the updates are preprocessed. It turns out that it is not necessary that the client structures are exact copies of the central structure. For example, often a dynamic data structure contains information that is only used for efficiently updating it. Since the client structures can use the information gathered during the update of the central structure, they do not need to have this information in their structure. A typical example is a dynamic data structure that maintains the answer of an order decomposable set problem. The main part of this structure is used to perform updates, whereas only a relatively small part of it contains the answer to the set problem.

We have given a powerful general technique that solves the multiple representation problem, such that a client structure can be updated in time proportional to the size of the changes in this structure. As an example, we have shown that there is a class of range trees that can be maintained in $O((\log n)^d)$ time, whereas in the version of this tree containing no balance information, only $O((\log n)^{d-1})$ nodes are changed in an update. Hence by applying this general technique, we can maintain the client version of the range tree in $O((\log n)^{d-1})$ time.

There remain several problems and directions for further research:

In order to apply our general technique, data structures are needed for which $C(n)$—the amount of data that is changed in an update—is small. It would be interesting to have more examples of such data structures.

In the present paper, we performed 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?

We have seen some techniques to solve the multiple representation problem for decomposable searching problems. It might be possible to design other schemes for these problems.

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] J.L. Bentley. *Decomposable Searching Problems.* Inform. Proc. Lett. **8** (1979), pp. 244-251.

[2] J.L. Bentley and J.B. Saxe. *Decomposable Searching Problems I: Static to Dynamic Transformations.* J. of Algorithms **1** (1980), pp. 301-358.

[3] N. Blum and K. Mehlhorn. *On the Average Number of Rebalancing Operations in Weight-Balanced Trees.* Theor. Comp. Sci. **11** (1980), pp. 303-320.

[4] D.G. Kirkpatrick. *Optimal Search in Planar Subdivisions.* SIAM J. Computing **12** (1983), pp. 28-35.

[5] 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.

[6] J. Nievergelt and E.M. Reingold. *Binary Search Trees of Bounded Balance.* SIAM J. Computing **2** (1973), pp. 33-43.

[7] H.J. Olivié. *A Study of Balanced Binary Trees and Balanced One-Two Trees.* Ph.D. Thesis, University of Antwerp, Department of Mathematics, 1980.

[8] H.J. Olivié. *A New Class of Balanced Trees: Half Balanced Binary Search Trees.* RAIRO Informatique Théorique **16** (1982), pp. 51-71.

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

[10] 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.

[11] 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, Department of Computer Science, 1987.

[12] 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.

[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, Department of Computer Science, 1987.

[14] D.E. Willard and G.S. Lueker. *Adding Range Restriction Capability to Dynamic Data Structures.* Journal of the ACM **32** (1985), pp. 597-617.