Dynamic data structures on multiple storage media, a tutorial*

Michiel Smid
Peter van Emde Boas

A 15/90

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

July 1990

* This work was supported by the ESPRIT II Basic Research Actions Program, under contract No. 3075 (project ALCOM).

Dynamic Data Structures on Multiple Storage Media, a Tutorial

Michiel Smid*

Fachbereich Informatik, Universität des Saarlandes

D-6600 Saarbrücken, West-Germany

Peter van Emde Boas

Departments of Mathematics and Computer Science

University of Amsterdam, The Netherlands

Abstract

Two problems are considered that deal with dynamic data structures on multiple storage media. In the first problem we want to partition a range tree into parts of a small size, such that queries and updates visit only a small number of parts. In this way, the range tree can be maintained in secondary memory efficiently. The second problem is the reconstruction problem: given a main memory data structure, design a shadow administration, to be stored in secondary memory, such that the main memory structure can be reconstructed after e.g. a system crash.

Key words: Dynamic data structures, binary search trees, range trees, secondary memory, reconstruction problem, union-find problem, deferred data structures.

This paper will be presented at SOFSEM'90, Zotavovna Siréna, Janské Lázně, Krkonoše, Czechoslovakia.

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

*This author was supported by the ESPRIT II Basic Research Actions Program under contract No. 3075 (project ALCOM).

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.

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 [11] 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 the first part of this paper, we consider the problem of storing and maintaining a specific data structure—a range tree—in secondary memory. This data structure was designed for main memory applications. It turns out, however, that the structure can also efficiently be maintained in secondary memory. Furthermore, we believe that the techniques applied to this one data structure can also be applied to many more data structures.

The problem of considering range trees was posed by Mark Overmars, and the research on this problem was done in close collaboration with him, Mark de Berg and Marc van Kreveld.

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 second part of 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. Leen Torenvliet and Peter van Emde Boas investigated this reconstruction and optimization problem for these functions in [19]. A few years later, the authors, together with Leen Torenvliet and Mark Overmars studied this reconstruction problem in a more general setting.

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 binary search trees and the basic data structure of the first part of this paper, the range tree. In Section 4, we show how range trees can be maintained in secondary memory, by giving several efficient partitions. In Section 5, we consider the reconstruction problem. In Section 5.1, 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 5.2, 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 5.3, 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 5.2 and 5.3 contain results that are also interesting in other areas besides the reconstruction problem. We finish the paper in Section 6 with some concluding remarks.

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 Binary search trees and range trees

3.1 Binary search trees

The reader is assumed to be familiar with the basic terminology from graph theory, especially trees. (See e.g. [2].) A *binary tree* is a rooted tree, in which each *node* has either zero or two *sons*. The link between a node and its son is called an *edge*. Nodes without sons are called *leaves*, whereas nodes that do have sons are called *internal nodes*. The two sons of an internal node v are called *left son* and *right son* The node v itself is called the *father* of the two sons. If v is a node of a binary tree, we define the *subtree* of v as the tree having v as its root and that contains all nodes—including v—that can be reached from v by following edges to sons. The *height* of a binary tree is defined as the number of edges in the longest root-to-leaf path. A binary tree consists of *levels*, where a level is the set of all nodes that are at the same distance to the root of the tree. Here the *distance* of two nodes is defined as the number of edges on the path that connects these nodes. The levels of a binary tree are numbered according to their distance to the root of the tree. So the root itself is at level 0, the sons of the roots are at level 1, etc.

A binary tree that stores a set of objects is called a *node search tree*, if the objects are stored in the nodes of the tree—one object in each node—in such a way that for each internal node v it holds that all objects in the left subtree of v are smaller than the object stored in v, and all objects in the right subtree of v are larger than that of v, according to some order.

In this paper, binary trees are almost always used as *leaf search trees*. That is, if we use a binary tree to represent a set of objects, we store these objects in the leaves of the tree, such that for each internal node v, all objects in the left subtree of v are smaller than those in the right subtree of v. Internal nodes of the tree contain information to guide searches. (For example, we can store in each internal node the maximal element in its left subtree.)

Binary search trees can be used to solve the member searching problem. In order to search for an object q, we follow a path in the tree starting at the root. In each node on this path, we compare q with the information stored at that node, and we decide whether the search is finished—in case we have found q or end in a leaf—or proceeds to the left or to the right son. The complexity of this search procedure depends on the height of the tree. Since the height of a binary search tree storing n objects is at least logarithmic in n, the best we can hope for is a search complexity of $\Theta(\log n)$. In the static case, we can build a *perfectly balanced binary search tree*, which is a binary tree in which for each internal node v, the number of leaves in the two subtrees of v differ by at most one. Such trees have logarithmic height, and, hence, member queries can be performed in $O(\log n)$ time.

An insertion or deletion of an object p in a leaf search tree is performed by first searching for p. This search ends in a leaf v. In case of an insertion, we give v two new sons, one son containing p, the other containing the object that was stored in v. We also update the search information that is stored at the nodes on the path to v. In case of a deletion, let w be the other son of v's father. Then we delete the two leaves v and w, and we store the object that was stored in w in its father. Again, we update the search information of the nodes on the search path. The complexity of this update procedure is proportional to the height of the tree.

The problem is how to maintain a logarithmic height after objects have been inserted and deleted in the tree. An interesting class of binary search trees for which this is possible, was introduced in 1973 by Nievergelt and Reingold [10]:

Definition 1 *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 $1 - \alpha$.*

We give a simple technique to maintain BB[α]-trees after objects are inserted or deleted. This technique—the *partial rebuilding technique* of Lueker [9]—gives an *amortized* update complexity of $O(\log n)$. Here, amortized complexity is defined as follows. Suppose we perform a sequence of m updates in a data structure that initially stores n objects. The value of m is assumed to be large. Let $T(n,m)$ be the time to process this sequence. Assume that this sequence is such that $T(n,m)$ is maximal among all sequences of length m. Then the *amortized* update time is defined as $T(n,m)/m$. Hence, in amortized time bounds, we average over long worst-case sequences of updates.

Suppose we want to insert or delete object p in the leaf search BB[α]-tree T. We search for p, until we end in a leaf. Then we insert or delete object p. Next, we walk back to the root of T, and we find the highest node v that does not satisfy the balance condition of Definition 1 anymore. We rebalance the tree by rebuilding the entire subtree of v as a perfectly balanced tree. Clearly, if v is high in the tree, this takes a lot of time. For example, if v is the root of T, the update takes $O(n)$ time. In this case, however, it takes $\Omega(n)$ updates before we again have to rebuild the entire tree. In this way, the amortized update complexity is bounded by $O(\log n)$. To prove this, we need the following lemma, the proof of which can be found in [11, page 53].

Lemma 1 *Let v be a node in a BB[α]-tree that is in perfect balance. Let n_v be the number of leaves in the subtree of v at the moment it gets out of balance. Then there have been at least $(1-2\alpha)n_v - 2$ updates in the subtree of v.*

Theorem 1 *If in a leaf search BB[α]-tree, updates are performed by means of the partial rebuilding technique, the amortized time for an update is bounded by $O(\log n)$.*

Proof. Let $U(n)$ denote this amortized update complexity for a BB[α]-tree with n leaves. To perform an update, we start at the root of the tree, and we decide whether we proceed to the left or to the right son. If the entire tree is not rebuilt, we spend $O(1)$ time in the root. Otherwise, we spend $O(n)$ time to rebuild the tree, since we have the objects already in sorted order. By Lemma 1, this rebuilding has to be done at most once every $\Omega(n)$ updates. It follows that the amortized time due to our visit to the root is bounded by $O(1)$. The amortized time we spend in the subtree in which the update proceeds, is bounded by $U((1-\alpha)n)$, since this subtree has at most $(1-\alpha)n$ leaves. Hence $U(n) \leq O(1) + U((1-\alpha)n)$, from which it follows that $U(n) = O(\log n)$. $\square$

3.2 Range trees

Range trees are used to solve the following problem:

Definition 2 *Let V be a set of points in d-dimensional space, and let $([x_1 : y_1], [x_2 : y_2], \ldots, [x_d : y_d])$ be some hyperrectangle. The* orthogonal range searching problem *asks for all points $p = (p_1, p_2, \ldots, p_d)$ in V, such that $x_1 \leq p_1 \leq y_1, x_2 \leq p_2 \leq y_2, \ldots, x_d \leq p_d \leq y_d$.*

The range searching problem has applications in e.g. computer graphics and database design. As an example, consider a salary administration, in which the information for each registered person includes age and salary. We can view each person as a point in 2-dimensional space, with as first coordinate the age, and as second coordinate the salary. Then a question like "give all persons with age between 20 and 25, having a salary between \$ 30,000 and \$ 35,000 a year" is an example of a range query. Range trees were introduced by Bentley [4] and Lueker [9]:

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

1. *If $d = 1$, then T is a $BB[\alpha]$-tree, containing the elements of V in sorted order in its leaves.*
2. *If $d > 1$, then T consists of a $BB[\alpha]$-tree, called the* main tree, *which contains in its leaves the elements of V, ordered according to their first coordinates. Each internal node w of this main tree contains (a pointer to) an* associated structure, *which is a $(d-1)$-dimensional range tree for those elements of V that are in the subtree rooted at w, taking only the second to d-th coordinate into account.*

Let T be a range tree, representing the set V, and let w be a node of T (w is a node of the main tree, or of an associated structure, or of an associated structure of an associated structure, etc.). Let V_w be the set of those points of V that are in the subtree of w. Then node w is said to *represent* the set V_w.

For example, a 2-dimensional range tree for a set V consists of a binary tree, containing in its leaves the points of V ordered according to their x-coordinates. For any internal node w of this tree, let V_w be the subset of V represented by w. Then node w contains (a pointer to) a binary tree, representing the set V_w, ordered according to their y-coordinates. See Figure 1.

The building algorithm: Let V be a set of n points in d-dimensional space. To build a range tree for V, we order the points of V according to their d-th coordinates.

Let $d > 1$. We build a perfectly balanced $(d-1)$-dimensional range tree for the set V, taking only the second to d-th coordinate into account. This range tree becomes the associated structure of the root of the main tree of the final structure. Next, we divide the set V in two subsets V_1 and V_2 of equal size, such that the first coordinates of the points in V_1 are less than those in V_2. This splitting is done in such a way that the points in both sets V_1 and V_2 remain ordered according to their last coordinates. Then we build recursively two d-dimensional range trees for the sets V_1 and V_2.

The query algorithm: Orthogonal range queries are solved as follows. We first consider the one-dimensional case. Let $[x_1 : y_1]$ be a query interval. Then we search in the binary tree with both x_1 and y_1. Assume w.l.o.g. that $x_1 < y_1$. We have to report all leaves that lie between the paths to x_1 and y_1. Let u be that node in the tree for which x_1 lies in the left subtree of u, and y_1 lies in the right subtree of u.

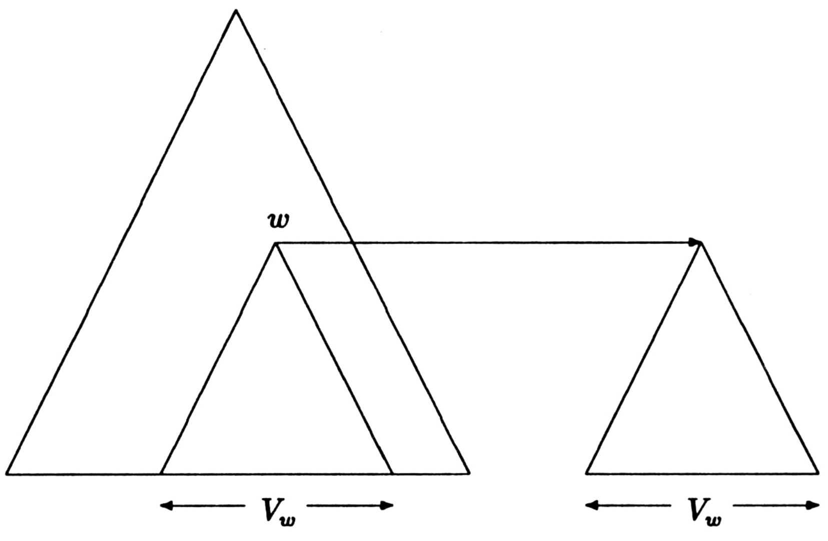

Figure 1: A two-dimensional range tree

Then for each node $v \neq u$ on the path from u to x_1, for which the search proceeds to the left son of v, we report the leaves in the right subtree of v. Similarly, for each node $w \neq u$ on the path from u to y_1, for which the search proceeds to the right son of w, we report the leaves in the left subtree of w. Finally, we check the two leaves in which the paths end.

Now let $d > 1$ and let $([x_1 : y_1], [x_2 : y_2], \ldots, [x_d : y_d])$ be a query rectangle. Then we begin by searching with both x_1 and y_1 in the main tree. Assume w.l.o.g. that $x_1 < y_1$. Let u be that node in the main tree for which x_1 lies in the left subtree of u, and y_1 lies in the right subtree of u. Then we have to perform a range query with the last $d-1$ coordinates on all points that lie between x_1 and y_1 in the main tree. It is not difficult to see that it is sufficient to perform recursively a $(d-1)$-dimensional range query in the associated structure of the right son of each node $v \neq u$ on the path from u to x_1 for which the search proceeds to the left son of v, and in the associated structure of the left son of each node $w \neq u$ on the path from u to y_1 for which the search proceeds to the right son of w. We also have to check the points in the two leaves of the main tree in which the paths end. The answer to the entire query is the union of the answers of these partial queries. Note that each point in the query rectangle is reported exactly once.

The update algorithm: Suppose we want to insert or delete a point p in the range tree. Then we search with the first coordinate of 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 usual insertion/deletion algorithm for binary trees; otherwise we use the same procedure recursively. Next, we insert or delete

p among the leaves of the main tree. In order to keep the trees balanced, we use Lueker's partial rebuilding technique. After we have inserted or deleted p among the leaves of the main tree, we walk back to the root. During this walk, we locate the highest node v that is out of balance, i.e., does not satisfy the balance condition of Definition 1 anymore. Then we rebalance at node v by rebuilding the entire structure rooted at v as a perfectly balanced range tree.

Just as in Section 3.1, if in this update algorithm node v is the root of the main tree, we have to rebuild the entire range tree. We saw in Lemma 1, however, that in this case $\Omega(n)$ updates must occur before we again have to rebuild the entire structure. The following theorem—due to Lueker [9]—gives the complexity of a range tree.

Theorem 2 *A d-dimensional range tree for n points, can be built in $O(n \log n + n(\log n)^{d-1})$ time, and requires $O(n(\log n)^{d-1})$ space. Using this tree, orthogonal range queries can be solved in $O((\log n)^d + t)$ time, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^d)$.*

Proof. We first prove the bound on the building time. Let V be a set of n points in d-dimensional space. It takes $O(n \log n)$ time to order the points of V according to their d-th coordinates. Let $P(n, d)$ be the time to build a perfectly balanced d-dimensional range tree for n points, that are ordered according to their last coordinates. Then $P(n, 1) = O(n)$. Let $d > 1$. The building of the associated structure of the root of the main tree takes $P(n, d-1)$ time. Using a linear time median algorithm (see [5]), the splitting of the set V in two equal sized subsets V_1 and V_2 can be done in $O(n)$ time. This splitting can be done such that the points in both sets V_1 and V_2 remain ordered according to their last coordinates. Finally, it takes $2\,P(n/2, d)$ time to build two d-dimensional range trees for the sets V_1 and V_2.

We have proved that $P(n, d) = 2\,P(n/2, d) + P(n, d-1) + O(n)$ for $d > 1$. It follows that $P(n, d) = O(n(\log n)^{d-1})$. This proves the bound on the building time. The bound on the size of the data structure can be proved in a similar way.

The bound on the query time follows by induction on d, since in the above described query algorithm, the paths in the main tree give rise to $O(\log n)$ $(d-1)$-dimensional range queries. A one-dimensional range query takes $O(\log n + t)$ time, since the height of a BB[α]-tree is bounded by $O(\log n)$. We saw already that each point in the query rectangle is reported exactly once.

Let $U(n, d)$ be the amortized update time in a d-dimensional range tree for a set of n points. Then, by Theorem 1, $U(n, 1) = O(\log n)$. Let $d > 1$. To perform an update, we start in the root of the main tree and we update its associated structure. This takes, amortized, $U(n, d-1)$ time. Then we repeat the same procedure for the appropriate son of the root, which is the root of a d-dimensional range tree for at most $(1-\alpha)n$ points. Therefore, this takes, amortized, at most $U((1-\alpha)n, d)$ time. If the root of the main tree gets out of balance, we rebuild the entire tree, which takes $O(n(\log n)^{d-1})$ time. According to Lemma 1, this happens at most once every

$\Omega(n)$ updates. So this rebuilding adds $O((\log n)^{d-1})$ to the amortized update time. We have proved that for $d > 1$

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

It follows that $U(n,d) = O((\log n)^d)$. $\square$

4 Maintaining range trees in secondary memory

4.1 The partitioning problem

In this section, we study the problem of storing and maintaining range trees in secondary memory. If a data structure is too large to be stored in main memory, it has to be stored in secondary memory. In Section 2, we saw that a data structure is stored in secondary memory by partitioning it into a number of parts, and by distributing the parts over blocks of some predetermined size. In order to answer queries and to perform updates, parts of the data structure that are needed in the operation are transported from secondary memory to main memory, and vice versa. Since the complexity of an operation is expressed by the number of disk accesses and by the amount of data that is transported, it is necessary to partition the data structure into parts, such that queries and updates pass through only a small number of parts, each of which has small size. This leads to the following definition.

Definition 4 *A partition of a dynamic data structure, representing a set of n points, is called an* $(f(n), g(n), h(n))$-partition, *if:*

1. *Each part has size at most* $f(n)$.
2. *There are* $O(S(n)/f(n))$ *parts, where* $S(n)$ *is the amount of space required to store the data structure.*
3. *Each query passes through at most* $g(n)$ *parts.*
4. *The amortized number of parts through which an update passes is at most* $h(n)$.

The relation of this definition to the above should be clear. It states, that we can store the data structure in secondary memory, such that a query requires at most $g(n)$ disk accesses and $f(n) \times g(n)$ data transport. Also, an update takes—amortized—at most $h(n)$ disk accesses and $f(n) \times h(n)$ data transport.

In this section, we design various partition schemes for range trees. For simplicity, we only consider the two-dimensional case. We consider two types of partitions. The first type are the so-called *restricted partitions*. In a restricted partition, only the main tree is partitioned into parts, whereas associated structures are never subdivided. In such a partition, a node of the main tree and its associated structure are contained in the same part. The second type of partitions are those in which also associated structures are partitioned into parts.

4.2 Restricted partitions

We first consider restricted partitions of range trees. Although there exist more efficient partitions, it is useful to consider these restricted partitions, because they are a lot easier to implement. Also, the techniques developed here apply to other data structures. In fact, any data structure that has the form of an augmented binary tree, with some reasonable properties of the query and update algorithms, can be partitioned in the way described in this section. Examples of such structures are segment trees, structures solving set problems like maintaining a convex hull, maintaining a Voronoi diagram, etc., and structures for adding range restrictions to searching problems (see e.g. Bentley [4] and Overmars [11]).

Note that in a restricted partition, parts have size $\Omega(n)$, because the associated structure of the root of the main tree has size $\Theta(n)$.

In order to be able to give efficient restricted partitions, we modify the definition of range tree somewhat. Let V be a set of n points in the plane. We suppose that the points of $V = \{p_1 < p_2 < p_3 < \ldots < p_n\}$ are ordered according to their x-coordinates. Partition V into subsets $V_1 = \{p_1, p_2, \ldots, p_{h(n)}\}$, $V_2 = \{p_{h(n)+1}, \ldots, p_{2h(n)}\}$, etc., where $h(n) = \lceil n/\log n \rceil$.

Definition 5 *A modified range tree, representing the set V, is defined as follows.*

1. *Each set V_i is stored in a two-dimensional range tree T_i. In the root of T_i we do not store an associated structure. Let r_i be the root of T_i. The roots are ordered according to $r_1 < r_2 < r_3 < \ldots$.*
2. *The roots r_i are stored in the leaves of a perfectly balanced binary tree T. Let v be any node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$ (v may be a leaf of T). Then v contains an associated structure, which is a BB[α]-tree, representing the set $V_i \cup V_{i+1} \cup \ldots \cup V_j$, ordered according to their y-coordinates.*

Query and update algorithms: First, note that the structure of a range tree is not changed, only the balance conditions are different. Therefore, in a modified range tree, range queries are solved in the same way as in ordinary range trees. An insertion or deletion of a point p is performed as follows. First we walk down tree T, to find the appropriate root r_i. During this walk we insert or delete p in all associated structures we encounter on our search path. Then we insert or delete p in T_i, using the update algorithm for range trees.

Suppose at the moment we build this structure, the set V contains n points. Then each set V_i (except for the "last" one) contains $\lceil n/\log n \rceil$ points. As soon as at least one set V_i contains either $\lceil n/\log n \rceil/2$ or $2\lceil n/\log n \rceil$ points, we rebuild the entire data structure.

Theorem 3 *A modified range tree, representing n points, can be built in $O(n \log n)$ time, and takes $O(n \log n)$ space to store. Range queries can be solved, using this tree, in $O((\log n)^2 + t)$ time, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^2)$.*

Proof. The bounds for the size, the building time and the query time can be proved in the same way as in Theorem 2. If the entire data structure is not rebuilt, an update takes amortized $O((\log n)^2)$ time, since each set V_i contains $\Theta(n/\log n)$ points. The data structure is rebuilt at most once every $\Omega(n/\log n)$ updates. Since this rebuilding takes $O(n \log n)$ time, this adds $O((\log n)^2)$ to the amortized update time. □

Theorem 4 *For a modified range tree, there exists an* $(O(n), \log\log n + O(1), \log\log n + O(1))$*-partition.*

Proof. Each tree T_i represents $\Theta(n/\log n)$ points. So it has size $O(n)$ and, hence, it can form a part of the partition. This gives us $O(\log n)$ parts. Each level of the tree T, together with its associated structures, forms a part, again of size $O(n)$. Since tree T is perfectly balanced, it has height $\log\log n + O(1)$. So this gives us $\log\log n + O(1)$ parts. A query passes through all levels of T, and through at most 2 trees T_i (since we store associated structures in the leaves of T). Hence it passes through $\log\log n + O(1)$ parts. An update passes through $\log\log n + O(1)$ parts, if we do not have to rebuild the data structure. If we have to rebuild the structure, $O(\log n)$ parts are involved. Since this has to be done at most once every $\Omega(n/\log n)$ updates, the amortized number of parts through which an update passes is at most $\log\log n + O(1) + O((\log n)^2/n) = \log\log n + O(1)$. □

Theorem 5 *For a modified range tree, there exists an* $(O(n \log\log n), 3, 2 + o(1))$*-partition.*

Proof. The tree T, together with its associated structures, forms a part on its own, of size $O(n \log\log n)$. Furthermore, we put sets of $\lceil \log\log n \rceil$ trees T_i together in one part. A query passes through at most 3 parts: The part containing tree T, and at most 2 parts containing trees T_i (again we use the fact that we also store associated structures in the leaves of T). An update passes through exactly 2 parts, if the data structure is not rebuilt. Since rebuilding of the structure has to be done at most once every $\Omega(n/\log n)$ updates, and since $O(\log n/\log\log n)$ parts are involved in this rebuilding, the amortized number of parts through which an update passes is $2 + o(1)$. □

Next we improve Theorem 4 considerably. We want to partition a modified range tree into parts of size $O(n)$. Since each tree T_i has size $O(n)$, it can form a part on its own.

We are left with the tree T and its associated structures. We first sketch how these structures are partitioned. The root of T, together with its associated structure, forms a part. This removes the top level of T. Now consider the two sons v and w of the root. Look at the subtree consisting of v and its two sons. It takes, together with its associated structures, $O(n)$ storage and, hence, can form a part. Similarly for w. This removes two more levels of T; so we are left with 8 sons. For each son u, we make a part consisting of the subtree with root u, of depth 8, where the *depth* of a tree equals the number of levels. This subtree, of course with its associated structures, uses $O(n)$ space. We now have removed 11 levels. So we are left with 2^{11} sons. For each son, we take a subtree of depth 2^{11}, with associated structures, which takes $O(n)$ storage. Next we are left with $2^{2^{11}+11}$ sons, etc. Note that the tree T is perfectly balanced. So a node on level i represents $\Theta(n/2^i)$ points. We describe the above more precisely.

The partition: Each tree T_i forms a part on its own. Let $a_0 = 0$ and $a_{k+1} = 2^{a_k} + a_k$ for $k \geq 0$. Let d be the height of tree T, and let $m = \min\{i \geq 0 | a_i > d\}$. The tree T and its associated structures are partitioned as follows. For each $k, 0 \leq k \leq m-1$, there are 2^{a_k} parts. Each such part is a subtree of T, together with its associated structures, having its root at level a_k, of depth 2^{a_k}.

Before we state the result, we introduce a very slowly growing function. Let $(\log)^k n$ denote the k-th iterated logarithm. Then we define the function $\log^* n$ by $\log^* n := \min\{k \geq 1 | (\log)^k n \leq 1\}$.

Theorem 6 *For a modified range tree, there exists an* $(O(n), 4\log^* n + O(1), \log^* n + O(1))$*-partition.*

Proof. We saw already that each tree T_i has size $O(n)$. Furthermore, there are $O(\log n)$ such trees. Since the tree T is perfectly balanced, we have $d = \log\log n + O(1)$. The tree T is partitioned into

$$\sum_{k=0}^{m-1} 2^{a_k} = O(2^{a_{m-1}}) = O(2^d) = O(2^{\log\log n + O(1)}) = O(\log n)$$

parts. Each such part is a subtree of T, together with its associated structures, having its root at level a_k, of depth 2^{a_k}. Since this root represents $n/2^{a_k}$ points, such a part has size $O(n)$.

Now let $([x_1 : y_1], [x_2 : y_2])$ be a query rectangle, and consider the path in T from the root to x_1. Look at a node v through which this path passes, and let Π be the part of the partition containing this node. If this path proceeds to the left son, we have to search the associated structure of the right son of v. If v is not at the bottom level of Π, these left and right sons are also contained in Π. Otherwise, these two sons are contained in two different parts. So, since the number of parts through which this left path passes is m, the left path of the query passes through at most $2m + 1$ parts ($2m$ parts in tree T, and one part containing a tree T_i). Hence the number of parts through which a query passes is at most $4m + 2$. It can be shown, that $m \leq \log^* n + O(1)$. Therefore, a query passes through at most $4\log^* n + O(1)$ parts. Finally, an update passes through $m \leq \log^* n + O(1)$ parts of T and through one part containing a tree T_i, if we do not have to rebuild the data structure. If we take the cost of rebuilding into account, we see that—amortized—$\log^* n + O(1) + O((\log n)^2/n) = \log^* n + O(1)$ parts are involved in an update. □

This result means that we can query and maintain a modified range tree, stored in secondary memory, by transporting $O(\log^* n)$ parts of size $O(n)$. Observe that although $\log^* n$ goes to infinity as n does, for all practical values of n, we have $\log^* n \leq 5$. In fact, $\log^* n \leq 5$ for all $n \leq 2^{65536}$.

4.3 Changing range trees to make them partitionable

The best restricted partition into parts of size $O(n)$ we have seen so far, is the $(O(n)$, $O(\log^* n)$, $O(\log^* n))$-partition of Theorem 6. Although it is proved in [16] that this is optimal for restricted partitions of normal range trees, we show now that, making some slight changes, the bounds can be improved.

Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. We partition the set V into subsets $V_1 = \{p_1, \ldots, p_{h(n)}\}$, $V_2 = \{p_{h(n)+1}, \ldots, p_{2h(n)}\}$, etc., where $h(n) = \lceil n/\log n \rceil$.

Definition 6 *A* reduced range tree *representing the set V consists of the following.*

1. *Each set V_i is stored in a two-dimensional range tree T_i. Let r_i be the root of T_i.*
2. *These roots r_i are stored in the leaves of a perfectly balanced binary tree T.*

So in a reduced range tree, nodes that are high in the main tree (i.e., nodes representing many points) do not have an associated structure.

Query and update algorithms: To perform a query with range $([x_1 : y_1], [x_2 : y_2])$, we do the following. We search with x_1 and y_1 in tree T for the appropriate roots, say r_i and r_j. If $i = j$, we perform a query, with the rectangle $([x_1 : y_1], [x_2 : y_2])$, in the range tree T_i. Otherwise, if $i < j$, we perform queries, with the strip $([x_1 : \infty], [x_2 : y_2])$ in tree T_i, and with $([-\infty : y_1], [x_2 : y_2])$ in tree T_j. Furthermore, we perform one-dimensional range queries, with query interval $[x_2 : y_2]$ in the associated structures of the roots of the trees $T_{i+1}, \ldots, T_{j-1}$.

An insertion or deletion of a point p is performed as follows. First, we walk down tree T, to find the appropriate root r_i, and we insert or delete p in the tree T_i, using the update algorithm for range trees. Just as for modified range trees, we completely rebuild the data structure as soon as one set V_i contains either $\lceil n/\log n \rceil/2$ or $2\lceil n/\log n \rceil$ points.

Theorem 7 *A reduced range tree, can be built in $O(n \log n)$ time, and has size $O(n \log n)$. In this tree, range queries can be solved in $O((\log n)^2 + t)$ time, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^2)$.*

Proof. The bounds on the building time, the space requirement and the update time can be proved in the same way as for range trees (cf. Theorem 2). Consider the query algorithm for reduced range trees as described above. The time to find the roots r_i and r_j is proportional to the height of tree T, which is $O(\log \log n)$. If

$i = j$, we have to query the tree T_i, which takes $O((\log(n/\log n))^2) = O((\log n)^2)$ time. If $i < j$, we query the trees T_i and T_j, which takes $O((\log n)^2)$ time. Furthermore, the one-dimensional range queries in the associated structures of the roots of $T_{i+1}, \ldots, T_{j-1}$ take $O(\log n \times \log(n/\log n)) = O((\log n)^2)$ time, since there are $O(\log n)$ such associated structures, and each has a query time of $O(\log(n/\log n))$. Of course we have to add $O(t)$ to the total query time for reporting the answers. This proves the theorem. $\square$

The partition of a reduced range tree: We put the tree T, together with the associated structures of the roots of the trees T_i in one part. Furthermore, each tree T_i, without the associated structure of its root, forms one part of the partition.

Theorem 8 *For a reduced range tree, there exists an $(O(n), 3, 2 + o(1))$-partition.*

Proof. The part that contains T and the associated structures of the roots of the trees T_i, has size $O(\log n + \log n \times (n/\log n)) = O(n)$. It is clear that each T_i without the associated structure of its root has size $O(n)$. There are $O(\log n)$ such trees. Clearly, a query passes through at most 3 parts. Also, if the data structure is not rebuilt, an update passes through exactly 2 parts. If the structure is rebuilt, which happens at most once every $\Omega(n/\log n)$ updates, $O(\log n)$ parts are involved. Hence an update passes, amortized, through at most $2 + o(1)$ parts of the partition. $\square$

4.4 A partition in which updates pass through 3 parts

We now look at general partitions that also allow splitting associated structures. As a result we can reduce the size of the parts to be asymptotically less than n.

Definition 7 *Let $g(n)$ and $h(n)$ be integer functions, such that $1 \leq g(n) \leq n, 1 \leq h(n) \leq n$, and $g(n) \times h(n) \geq n/\log n$. Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. We partition the set V into subsets $V_1 = \{p_1, \ldots, p_{g(n)}\}$, $V_2 = \{p_{g(n)+1}, \ldots, p_{2g(n)}\}$, etc. Order the points of V according to their y-coordinates. Let $V = \{q_1 < q_2 < \ldots < q_n\}$ be the resulting set. We partition this set into subsets $W_1 = \{q_1, \ldots, q_{h(n)}\}$, $W_2 = \{q_{h(n)+1}, \ldots, q_{2h(n)}\}$, etc. A $(g(n), h(n))$-*range tree *is defined as follows.*

1. *Each set V_i is stored in a two-dimensional range tree T_i. Let r_i be the root of T_i.*
2. *these roots are stored in the leaves of a perfectly balanced binary tree T. Let v any node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$. Then v represents the set $V_{ij} = V_i \cup V_{i+1} \cup \ldots \cup V_j$. Let $I_v = \{k | V_{ij} \cap W_k \neq \emptyset\}$. node v contains an associated structure, representing the set V_{ij}, having the following form. There is a* top tree *T'_v, which is a BB[α]-tree, containing the set I_v in its leaves. Furthermore, each leaf k of this top tree, contains a BB[α]-tree T'_{vk}, containing in its leaves the points of $V_{ij} \cap W_k$, ordered according to their y-coordinates.*

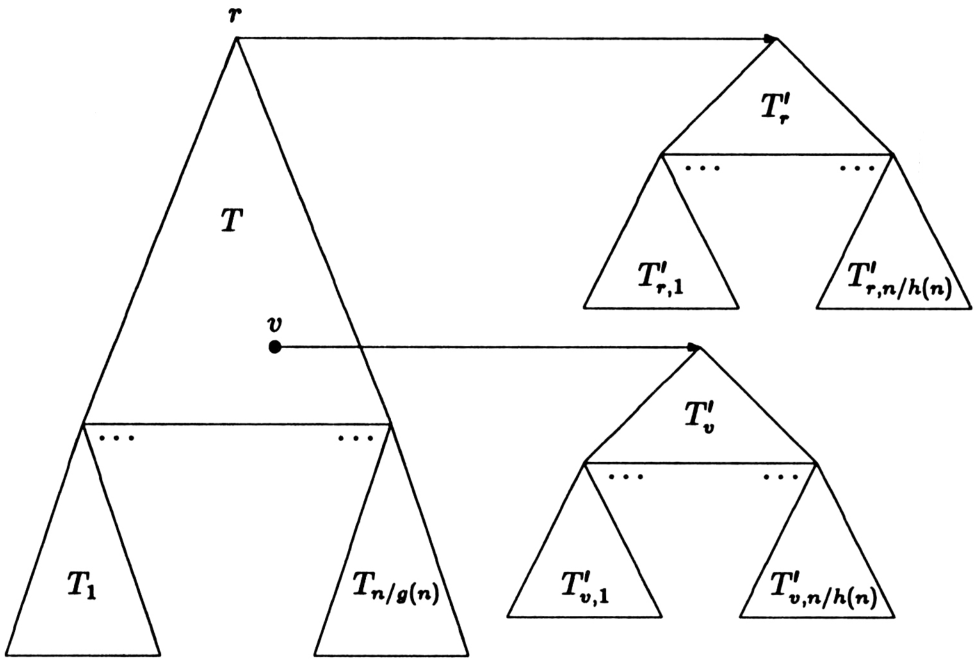

Figure 2: A $(g(n), h(n))$-range tree. $T_1, \ldots, T_{n/g(n)}$ are 2-dimensional range trees, the other trees are binary trees.

Observe that the associated structure of a node v of the tree T contains the points of $\bigcup_{k \in I_v} (V_{ij} \cap W_k) = V_{ij}$, ordered according to their y-coordinates. Also, for such a node v, we have $|I_v| = O(n/h(n))$. If r is the root of tree T, the set I_r contains all values of indices for which there is a set W_k. Therefore, the top tree T'_r associated with the root is perfectly balanced. See Figure 2 for a pictorial representation of a $(g(n), h(n))$-range tree.

Query and update algorithms: Since $(g(n), h(n))$-range trees have the same structure as ordinary range trees, the query algorithm for this data structure will be clear. An insertion or deletion of a point p is performed as follows. First we walk down tree T, to find the appropriate root r_i. During this walk, we have to update all associated structures we encounter on the search path. The first associated structure we encounter is that of the root r of T. We search in its top tree T'_r, to find the set W_k in which p has to be inserted or deleted. Then we update the corresponding tree T'_{rk}. Now for each node $v \neq r$ of T, that is on our search path, we do the following. We search in the top tree T'_v for k. (We know the value of k.)

1. Suppose that k is present in this top tree. Then we insert or delete p in the tree T'_{vk}. If T'_{vk} becomes empty, we delete k from the top tree T'_v.
2. Otherwise, k is not present in the top tree. (Then, point p is not present in the data structure, and therefore the update is an insertion: If p was present, then

k was present in the top tree T'_v. If we had to delete the point p, we would have noticed that it is not present during the update of the associated structure of the root r, and the update procedure would have stopped.) In this case, we insert k into the top tree, together with a tree T'_{vk} containing p.

Finally, point p is inserted or deleted in the appropriate range tree T_i, using the update algorithm for range trees.

In order to keep the data structure balanced, we completely rebuild it as soon as one set V_i contains either $g(n)/2$ or $2\,g(n)$ points, or as soon as one set W_j contains either $h(n)/2$ or $2\,h(n)$ points.

Theorem 9 *Let $g(n)$ and $h(n)$ be as before. A $(g(n), h(n))$-range tree, representing n points, can be built in $O(n \log n)$ time, and takes $O(n \log n)$ space to store. Using this tree, range queries can be solved in $O((\log n)^2 + t)$ time, where t is the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^2 + (n \log n)/\min(g(n), h(n)))$.*

Proof. Each tree T_i represents $O(g(n))$ points. Hence it has size $O(g(n) \log g(n))$. Since there are $O(n/g(n))$ such trees, they take together $O(n \log g(n))$ space. The tree T takes $O(n/g(n))$ space. Each top tree T'_v, where v is a node of T, has size $O(n/h(n))$. Hence all top trees together have size $O((n/g(n)) \times (n/h(n)))$. Consider a fixed level of T. The trees T'_{vk} of the associated structures on this level together represent the set V, and, hence, they have size $O(n)$. Since T has height $O(\log(n/g(n)))$, all these trees T'_{vk} together take $O(n \log(n/g(n)))$ space. Hence the size of the entire data structure is bounded by

$$O(n \log g(n)) + O((n/g(n)) \times (n/h(n))) + O(n \log(n/g(n))) = O(n \log n),$$

since $g(n) \times h(n) \geq n/\log n$. The bounds on the building, query and update time can be proved in an analogous way. $\square$

The partition: We partition a $(g(n), h(n))$-range tree as follows. Each tree T_i forms a part on its own. Next we put the tree T together with all top trees T'_v in one part of the partition. Finally, for each fixed k, the trees T'_{vk}, where v ranges over all nodes in T, are put together in one part. (Use Figure 2 to get an impression of this partition.)

Theorem 10 *A $(g(n), h(n))$-range tree, representing a set of n points, can be partitioned into parts of size $\Theta(f(n))$, where*

$$f(n) = \max\left(g(n) \log g(n), (n/g(n)) \times (n/h(n)), h(n) \log(n/g(n))\right),$$

such that a query passes through at most $5 + O(t/h(n))$ parts, where t is the number of reported answers, and the amortized number of parts through which an update passes is at most $3 + O((n \log n)/(f(n) \times \min(g(n), h(n))))$.

Proof. Each tree T_i forms a part of size $O(g(n)\log g(n))$. This gives us $O(n/g(n))$ parts. The tree T has size $O(n/g(n))$. There are $O(n/g(n))$ top trees, and each of them has size $O(n/h(n))$. So the part of the partition containing T and all top trees has size $O(n/g(n)) + O((n/g(n)) \times (n/h(n))) = O((n/g(n)) \times (n/h(n)))$. Take a fixed k. All trees T'_{vk}, where v ranges over the nodes in T, form one part. Consider a level of T. Let $v_1, v_2, \ldots, v_m$ be the nodes on this level. The trees $T'_{v_1k}, \ldots, T'_{v_mk}$ together represent the set W_k, which has size $O(h(n))$. So for this fixed k, all trees T'_{vk} together have size $O(h(n)\log(n/g(n)))$, since tree T has height $O(\log(n/g(n)))$. Since there are $O(n/h(n))$ possible values for k, this gives us $O(n/h(n))$ parts, each of size $O(h(n)\log(n/g(n)))$.

To summarize, we have $O(n/g(n))$ parts of size $O(g(n)\log g(n))$, one part of size $O((n/g(n)) \times (n/h(n)))$, and $O(n/h(n))$ parts of size $O(h(n)\log(n/g(n)))$. Then, in order to get the desired partition, we merge parts into $O((n\log n)/f(n))$ new parts of size $O(f(n))$.

Now consider an insertion or a deletion of a point, such that the data structure is not rebuilt. Let W_k be the set in which the point is inserted or deleted. Then this update passes through exactly three parts: The part containing T and the top trees; the part containing the trees T'_{vk}; and a part containing the appropriate range tree T_i. If the structure is rebuilt, $O((n\log n)/f(n))$ parts are involved in the update. Since this has to be done at most once every $\Omega(\min(g(n), h(n)))$ updates, it follows that the amortized number of parts through which an update passes is at most $3 + O((n\log n)/f(n) \times 1/\min(g(n), h(n)))$. The bound on the number of parts through which a query passes can be proved in a similar way. $\square$

Now we choose the functions $g(n)$ and $h(n)$ such that the sizes of the parts are minimal.

Corollary 1 *Let $g(n) = h(n) = \lceil n^{2/3}/(\log n)^{1/3}\rceil$. In a $(g(n), h(n))$-range tree, updates can be performed in amortized time $O(n^{1/3} \times (\log n)^{4/3})$. This range tree can be partitioned into parts of size $\Theta((n\log n)^{2/3})$, such that a query passes through at most $5 + O(t \times (\log n)^{1/3}/n^{2/3})$ parts, where t is the number of answers to the query, and the amortized number of parts through which an update passes is at most $3 + o(1)$.*

5 The reconstruction problem

5.1 A model for the reconstruction problem

To study and analyze solutions to the reconstruction problem, we use the following conceptual model:

- *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*.

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. We assume here that the copy *CSH* is stored in consecutive blocks. Therefore, 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.

5.2 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].)

In this section we are interested in the single-operation time complexity of the union-find problem. Blum [6] 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.

Theorem 11 (Blum [6]) *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.

5.2.1 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.

Definition 8 *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 w.l.o.g. 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 w.l.o.g. 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 12 *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 8 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. □

Since the given data structure is contained in Blum's class $\mathcal{B}$, Theorems 11 and 12 yield:

Corollary 2 *The data structure of Theorem 12 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 $\epsilon > 0$.*

5.2.2 An efficient shadow administration

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 2 *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 2, 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 5.2.1. 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 w.l.o.g. 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 w.l.o.g. 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 13 *Let k and n be integers, such that $2 \leq k \leq n$. For the data structure of Theorem 12, 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.

5.3 Another approach: deferred data structuring

In the approach we have taken so far for solving 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 [8], 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.

5.3.1 The static deferred binary search tree

We first recall the static solution of [8] 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 [5]. 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.

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 $|L(v)|$. The operation *expand* is performed as follows:

First we compute the median m of $L(v)$, and we determine the sets $V_1 = \{x \in L(v) | x < m\}$ and $V_2 = \{x \in L(v) | x > m\}$. 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 $|V_1| > 1$, we set $L(v_1) := V_1$, $N(v_1) := |V_1|$ and $key(v_1) :=$ undefined. If $|V_1| = 1$, 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 Section 5.3.2.

Theorem 14 *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 $k > n$.*

5.3.2 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 $\textit{key}(v)$ 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 $\textit{key}(v)$ is equal to the median of the objects stored in this subtree.

If $\textit{key}(v)$ 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 $|L(v)|$, 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 $O(|L(w)|)$ 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.

We are left with the problem of keeping the deferred tree balanced. For the class of BB[α]-trees, the balance criterion depends only on the size of its subtrees. 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 $N(v)$. Therefore, we can generalize Lueker's partial rebuilding technique to deferred binary search trees:

The partial dismantling technique: 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 $\textit{key}(v)$:= undefined. Note that the value of $N(v)$ is already equal to $|L(v)|$. 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. By Lemma 1, however, this does not occur too often.

Theorem 15 *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. According to Lemma 1, 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 1. 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 $f(n,k) \leq c_2 k^2$ 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 $\Omega(n \log \sqrt{n}) = \Omega(n \log n)$ 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 [7,13,14].

5.3.3 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*.

For the update algorithm of *SH* and *INF*, see [14]. We concentrate on the reconstruction procedure. Suppose all information in main memory is destroyed. Then we transport the structure *SH* from secondary memory to main memory, and we build the binary tree *INF*. This can be done in one disk access, $O(n)$ 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 5.3.1 and 5.3.2, 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 $x_1 \leq p_1 \leq y_1, \ldots, x_d \leq p_d \leq y_d$.

A somewhat extended form of range tree can be used for solving range counting queries in $O((\log n)^d)$ time. This structure has size $O(n(\log n)^{d-1})$, and can be built in $O(n(\log n)^{d-1})$ time.

Using a similar technique as in Subsection 5.3.1, 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 [8].)

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 the update algorithm for range trees. The result is expressed in the following theorem.

Theorem 16 *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 $O(n(\log k)^{d-1} + k(\log n)^d)$.*

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*, and 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 16, we can perform $\Theta(n/\log n)$ operations in $O(n(\log n)^{d-1})$ time.

Using the 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 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.

6 Concluding remarks

We have given an overview of two data structuring problems: maintaining a dynamic data structure in secondary memory and the reconstruction problem. In [12,14], many more partitions of range trees are given. In [16], lower bounds are proved, from which it follows that many restricted partitions are optimal.

In [14,18], 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.

In the present paper, we have considered only one multiple representation problem. Another case where data is represented more than once is investigated in [14,17]: 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. Indeed, most of the techniques for designing shadow administrations can be generalized to this second multiple representation problem. For details, the reader is referred to [14,17].

References

[1] G.M. Adel'son-Vel'skiĭ and E.M. Landis: *An algorithm for the organization of information.* Soviet Math. Dokl., 3, 1962, pp. 1259-1262.

[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman: *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[3] R. Bayer and E.M. McCreight: *Organisation and maintenance of large ordered indexes.* Acta Informatica, 1, 1972, pp. 173-189.

[4] J.L. Bentley: *Decomposable searching problems.* Inform. Proc. Lett., 8, 1979, pp. 244-251.

[5] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest and R.E. Tarjan: *Time bounds for selection.* J. Comput. System Sci., 7, 1973, pp. 448-461.

[6] N. Blum: *On the single-operation worst-case time complexity of the disjoint set union problem.* SIAM J. Comput., 15, 1986, pp. 1021-1024.

[7] Y.T. Ching, K. Mehlhorn and M. Smid: *Dynamic deferred data structuring.* To appear in: Inform. Proc. Lett.

[8] R.M. Karp, R. Motwani and P. Raghavan: *Deferred data structuring.* SIAM J. Comput., 17, 1988, pp. 883-902.

[9] 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.

[10] J. Nievergelt and E.M. Reingold: *Binary search trees of bounded balance.* SIAM J. Comput., 2, 1973, pp. 33-43.

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

[12] M.H. Overmars, M.H.M. Smid, M.T. de Berg and M.J. van Kreveld. *Maintaing range trees in secondary memory, part I: partitions.* Acta Informatica, 27, 1990, pp. 423-452.

[13] M.H.M. Smid: *Dynamic deferred data structures.* ITLI Prepublication Series CT-89-01, Departments of Mathematics and Computer Science, University of Amsterdam, 1989.

[14] M.H.M. Smid: *Dynamic Data Structures on Multiple Storage Media.* Ph.D. Thesis. University of Amsterdam, 1989.

[15] M.H.M. Smid: *A data structure for the union-find problem having good single-operation complexity.* Algorithms Review, (Newsletter of the ESPRIT II AL-COM Project), 1, 1990, pp. 1-11.

[16] M.H.M. Smid and M.H. Overmars. *Maintaining range trees in secondary memory, part II: lower bounds.* Acta Informatica, 27, 1990, 453-480.

[17] M.H.M. Smid, M.H. Overmars, L. Torenvliet and P. van Emde Boas: *Maintaining multiple representations of dynamic data structures.* Information and Computation, 83, 1989, pp. 206-233.

[18] M.H.M. Smid, L. Torenvliet, P. van Emde Boas and M.H. Overmars: *Two models for the reconstruction problem for dynamic data structures.* J. Inform. Process. Cybernet. EIK, 25, 1989, pp. 131-155.

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