DYNAMIC DATA STRUCTURES ON MULTIPLE STORAGE MEDIA

Michiel Smid

November 1989

Contents

Part I

Introduction

Chapter 1

Outline of the thesis

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

A large part of the research is focussed on designing structures that are stored in the main memory of a computer, on which all standard computations can be performed, and which is usually modeled as a Random Access Machine (RAM). 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 in 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. Other examples are B-trees, introduced in 1972 by Bayer and McCreight [4] and BB[α]-trees, introduced in 1973

by Nievergelt and Reingold [37]. These classes of trees also allow search, insert and delete operations to be performed in logarithmic time. In 1975, Van Emde Boas Trees [62, 63] were designed. These trees can store integers from a fixed universe $[0, \ldots, u-1]$, such that the operations search, insert and delete can be performed in $O(\log \log u)$ time.

In 1979, the research on general dynamization techniques was initiated by Bentley [5]. 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 [42] for a detailed account of dynamization techniques.

In this thesis, we study three alternative ways of storing and maintaining dynamic data structures.

In Part II, we consider the problem of maintaining a specific dynamic data structure—a range tree—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. 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.

In the past, considerable research has been devoted to the design of secondary memory data structures. The best-known example is the B-tree, that was mentioned already. (B-trees have also applications as main memory data structure. They were designed, however, for secondary memory applications.) If a B-tree stores n objects, and if it is stored in blocks of size m, 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.

The approach we take here—take a known data structure that was designed for main memory, and investigate how it can be partitioned as efficiently as possible—is relatively new. In some sense, a B-tree can be seen as a partitioned binary search tree. The problem of partitioning priority search trees was investigated by Icking, Klein and Ottmann [27] and Blankenagel and Güting [10]. The partitioning of range trees has not been studied before.

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 Part III, we consider one such multiple representation 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 first appeared in a paper by Torenvliet and van Emde Boas [61], where the reconstruction and optimization of trie hashing functions are investigated. No other studies concerning this problem have appeared. Therefore, Part III is the first general study concerning this subject.

Another case where data is represented more than once is considered in Part IV. 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. The structure that performs the updates is called the *central structure*, whereas the other structures are called the *client structures*.

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

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

The problems are in some sense "dual" to each other: In the reconstruction problem there is a main structure on which queries and updates are performed. After an update has been carried out in this main structure, information is transported to secondary memory, where the shadow administration is updated. In this shadow structure no queries are performed. In the other multiple representation problem, there is a central structure on which only updates are performed. After this central structure is updated, data is transported to the client structures that makes it possible to update them. These client structures are also used for query answering.

Again, the multiple representation problem of Part IV has not been studied before. So Part IV contains the first general study concerning this problem.

Solutions to the three problems that are studied in this thesis have applications in the following areas:

- The theory of data bases.

- Computational geometry. Since in this area often data structures are used requiring more than linear space, it is sometimes possible to improve asymptotically upon the storage requirements.

- Paging dynamic data structures. Part II is completely devoted to this application. The techniques developed there can be applied to many other data structures as well. Furthermore, techniques to maintain shadow administrations in secondary memory can sometimes be used in case the data structure is only stored in secondary memory.

- Parallel implementations of data structures. Techniques to partition a data structure can sometimes be used to implement the

structure on a parallel machine. See [24], where the parallel implementation of partitioned range trees is investigated.

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

- Storing dynamic data structures in write-once memories. The results of the problem of designing efficient client structures give insight in which parts of data structures are actually changed when performing updates.

This text is organized as follows. In Chapter 2, we briefly review the basic concepts of data structures and searching problems. We recall binary search trees, range trees, and some important classes of searching problems. We also introduce the models of main memory and secondary memory.

The rest of the thesis consists of three parts, that can be read independently of each other. Each of these parts contains a final chapter, giving a summary of the most important results of that part.

In Part II, we consider the problem of partitioning a range tree into parts of a small size, such that each query and update needs information from only a small number of parts. We give many partitions of range trees, thereby obtaining a number of trade-offs between the number of disk accesses versus the amount of information that has to be transported. We consider two types of partitions. The first type—the restricted partitions—can easily be extended to many other data structures that have the structure of an augmented binary search tree. The second type is much more complicated, but yields better trade-offs. We

first consider 2-dimensional range trees. Later, all results are extended to arbitrary dimensions. Finally, we prove several lower bounds for partitions of range trees. For the restricted partitions, these lower bounds coincide with the best upper bounds.

In Part III, we study the reconstruction problem. We give a realistic general framework that we use to describe solutions to this problem. We study how to structure the shadow administration for different types of query problems, such as order decomposable set problems and decomposable searching problems. For some types of query problems, the shadow administration can be of an asymptotically smaller size than the main data structure itself, and the data structure can be reconstructed from this shadow administration in an amount of time that is smaller than the time needed to build the data structure from scratch.

We also give a general technique, in which we transport with each update only the changes in the shadow administration to secondary memory. To reconstruct the data structure, we then perform the changes of the most recent updates to get the up-to-date shadow administration. Then we reconstruct the data structure. By sending with the changes also the positions where these changes have to be carried out, we avoid that we have to search the positions of the pieces of information in the shadow administration that have to be updated. This technique gives very fast maintainable shadow administrations, and it is especially useful in case an update changes only a small part of the shadow structure.

Next we study a specific searching problem, the Union-Find problem. We design an efficient main memory data structure for this problem, having a lower worst-case single operation complexity than the best previously known structure. This structure depends on a parameter k, and for many values of k it is optimal in the very general class of data structures solving the union-find problem, as introduced by Blum [12]. This new data structure is designed in such a way that a copy of it can efficiently be maintained in secondary memory. Note that this structure is interesting in its own right.

Finally, we apply the recent idea of deferred data structuring—due to Karp, Motwani and Raghavan [28, 35]—to the reconstruction problem. Here, a data structure is built "on-the-fly" during query and update operations. With each operation, those parts of the data struc-

ture that do not exist, but that are needed, are built. These parts can then be used for future operations. Karp et al. only give static deferred structures, and they ask for dynamic structures. It is shown that using well-known dynamization techniques, it is often possible to design dynamic deferred data structures. Again, the results are interesting in their own right, and have applications in other areas besides the reconstruction problem.

We use dynamic deferred data structures to get a new approach for solving the reconstruction problem. In this approach, we do not require that the data structure is completely reconstructed before we proceed with query answering and performing updates. After a crash, we transport the shadow administration—that stores only the objects that are represented by the data structure—to main memory, and we immediately proceed with performing operations. The data structure is reconstructed in a deferred way.

In Part IV, we investigate the problem of designing efficient central and client structures. We give a general framework in which we describe solutions to this problem. As for the reconstruction problem, we give general techniques for order decomposable set problems and decomposable searching problems. It is shown that, after "preprocessing" an update by the central structure, the clients can often perform the update more efficiently. Also, in some situations the client structures can be of a smaller size than the central structure.

Finally, we give a general technique that is related to the general technique of Part III. In this technique, we send with an update only those parts of the client structure that have been changed. Again, in order to avoid searching in the client structures the positions where the structure has to be changed, we also send these positions. This leads to very fast maintainable client structures. We give classes of binary search trees and range trees, the client versions of which can be updated asymptotically faster than the central versions.

Chapter 2

Preliminaries: Data structures and searching problems

2.1 Introduction

In this chapter, we recall the basic concepts of data structures and searching problems. This chapter is not meant as a general introduction to these topics; we only recall the most important concepts that are used in this thesis. Readers who are not familiar with the material can find a more thorough introduction to data structures in Aho, Hopcroft and Ullman [2, 3], Knuth [30], Mehlhorn [32, 33] and Wirth [67]. For a general introduction to searching problems, the reader can consult Mehlhorn [33], Overmars [42] and Wiedermann [64].

Data structures are meant for storing sets of objects in such a way that questions (also called *queries*) about these objects can be answered efficiently. Often, the kind of questions that are asked is fixed. In this case, we want to structure the objects such that these specific queries can be handled in an efficient way. In this thesis we only consider data structures from this latter point of view. The kind of queries that are asked about the objects is called a *searching problem*. To be more precise:

Definition 2.1.1 Let T_1, T_2 and T_3 be sets of objects. A *searching*

problem is a mapping $PR : T_1 \times P(T_2) \to T_3$.

In this definition, $P(T_2)$ denotes the power set of T_2, i.e., the set of all subsets of T_2.

For example, in the *member searching problem*, we are given a finite set V and a query object q, and we have to decide whether or not q is an element of V. In this case $T_1 = T_2$, $T_3 = \{\mathbf{true}, \mathbf{false}\}$, and $PR(q, V) = (q \in V)$.

Another example is the *orthogonal range searching problem*, which is a central object of study throughout this thesis. Here, we are given a finite set V of points in the d-dimensional space, and an axis-parallel hyperrectangle $q = ([x_1 : y_1], \ldots, [x_d : y_d])$, and we are asked to determine all points $p = (p_1, \ldots, p_d)$ in V, such that $x_1 \leq p_1 \leq y_1, \ldots, x_d \leq p_d \leq y_d$, i.e., all points of V that are in the hyperrectangle q. In this case, T_1 is the set of all axis-parallel hyperrectangles, T_2 is the set of all points in the d-dimensional space, $T_3 = P(T_2)$, and $PR(q, V) = (q \cap V)$.

In the *convex hull searching problem*, we are given a finite set V of points in the d-dimensional real vector space, and a query point q, and we have to find out whether or not q is inside the convex hull of V. Here, the convex hull of V is the (unique) smallest convex set that contains V. Now T_1 and T_2 both are the set of all points in d-dimensional space, $T_3 = \{\mathbf{true}, \mathbf{false}\}$, and $PR(q, V) = (q \in \mathrm{conv}(V))$, where $\mathrm{conv}(V)$ denotes the convex hull of V.

Finally, in the *nearest neighbor searching problem*, we are given a finite set V of points in the plane, and a query point q, and we have to find a point of V that is nearest to q with respect to a given distance. In this case, T_1, T_2 and T_3 are the set of all points in the plane, and $PR(q, V) =$ a nearest neighbor of q in V.

Given a finite set V in $P(T_2)$, a solution to the searching problem PR consists of a *data structure* DS, representing V, such that queries (i.e., $PR(q, V)$ for q in T_1) can be computed efficiently. If the set V is given beforehand and does not change, the data structure is called *static*. The structure is called *dynamic*, if it is possible to insert and delete objects. If only insertions are possible, the structure is called *semi-dynamic*. Insertions and deletions are called *updates*.

The Random Access Machine: In this text we take the Ran-

dom Access Machine (RAM) with real arithmetic as our machine model. (See e.g. [2].) The memory of a RAM consists of an array, the entries of which have unique addresses. The contents of such an array entry can be obtained at constant cost, provided its address, i.e., its index, is known. Data structures are composed of "indivisible pieces of information", such as pointers, names, integers, etc. We assume—as is customary in the theory of algorithms and data structures—that such a piece of information has size one. Each piece of information can be stored in one entry of main memory. We consider a pointer to be an index of an array entry.

On a RAM, the functions addition, subtraction, multiplication and division can be computed in constant time. If functions like logarithms, floor and ceiling functions are needed, we can compute the values that are needed during the preprocessing of the data structure and store them in tables, using the four basic operations. The time and space needed to compute and store these values are in general subsumed by the time and space required to build and store the data structure itself.

The Pointer Machine: In Chapter 10, we design a data structure that can be implemented on a weaker model of main memory, the Pointer Machine, due to Tarjan [58]. In this model, the above standard functions can also be computed in constant time. The memory of a Pointer Machine consists of a collection of nodes that contain the information. These nodes can be linked by pointers. The difference with a RAM is that on a Pointer Machine, no direct addressing is possible. If the algorithm is in some node of the memory, and if it wants to access another node, it has to follow pointers until it reaches that node. As an example, if a set of objects is stored in sorted order in an array, we can not perform binary search in an efficient way. The only way to capture binary search efficiently, is to store the objects in a binary search tree, and to link the nodes by pointers. For more details about the Pointer Machine, see [58].

The complexity of a data structure DS, that stores a set of cardinality n, is given by the following functions:

- $P_{DS}(n)$: the preprocessing time, which is the time needed to build DS.

- $S_{DS}(n)$: the amount of space needed to store DS.
- $Q_{DS}(n)$: the time needed to answer a query in DS.
- $I_{DS}(n)$: the time needed to insert an object into DS. (In case DS is a (semi-)dynamic data structure.)
- $D_{DS}(n)$: the time needed to delete an object from DS. (In case DS is a dynamic data structure.)
- If the insertion and deletion times are equal, we denote the common update time by $U_{DS}(n)$.

If the data structure DS is clear from the context, we just write $P(n)$, $S(n)$, $Q(n)$, $I(n)$, $D(n)$ and $U(n)$ for these complexity measures.

All complexity measures are expressed in terms of words, which is customary in the theory of data structures and algorithms. These complexity measures are *worst-case* complexities, unless stated otherwise, in which case they are *amortized* complexities. Consider an initially empty data structure. Suppose we perform a sequence of n updates in this structure. Let these updates be chosen such that the total time $T(n)$ for performing them is maximal among all sequences of n updates. Then the *amortized update time* of the data structure is defined as $T(n)/n$.

To estimate the complexities we use the following notations. Let $f(n)$ and $g(n)$ be two positive functions, defined on the positive integers.

- $f(n) = O(g(n))$ if there is a constant $c > 0$ and an integer n_0, such that for each $n \geq n_0$, we have $f(n) \leq c\, g(n)$.
- $f(n) = \Omega(g(n))$ if there is a constant $c > 0$ and an integer n_0, such that for each $n \geq n_0$, we have $f(n) \geq c\, g(n)$.
- $f(n) = \Theta(g(n))$ if there are constants $c_1 > 0$ and $c_2 > 0$, and an integer n_0, such that for each $n \geq n_0$ we have $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$.
- $f(n) = o(g(n))$ if $\lim_{n\to\infty} f(n)/g(n) = 0$.

There exists another definition of Ω: $f(n) = \Omega(g(n))$ if there is a constant $c > 0$, such that $f(n) \geq c\, g(n)$ for infinitely many values of n. Our definition, however, has the advantage that the following relation holds:

$$O(f(n))/\Omega(g(n)) = O(f(n)/g(n)).$$

This relation is often used in the proofs of amortized complexity bounds.

We assume that all complexity measures are non-decreasing and *smooth*. Here, a positive function $f(n)$ is called smooth if $f(O(n)) = O(f(n))$. We further assume that the functions $S(n)/n$ and $P(n)/n$ are non-decreasing. In all applications we encounter, these assumptions are satisfied.

To finish this section, we introduce some notations. First, logarithms, and powers of logarithms, are written in the usual way, i.e., we write $\log n$, $(\log n)^2$, etc. (Unless stated otherwise, all logarithms are to the base two.) The k-th iterated logarithm is defined and denoted as follows. If $k = 0$, then $(\log)^0 n = n$. If $k \geq 1$, then $(\log)^k n = \log\left((\log)^{k-1} n\right)$. The function $\log^* n$ is defined by $\log^* n = \min\{k \geq 1 | (\log)^k n \leq 1\}$.

2.2 Binary search trees

The reader is assumed to be familiar with the basic terminology from graph theory. (See e.g. [2, 3].) Many data structures we encounter contain binary search trees as substructures. 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 thesis, 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.) It can be shown by induction on the number of leaves, that a binary tree with n leaves has exactly $2n - 1$ nodes.

Binary search trees are 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. In 1962, Adel'son-Velskiĭ and Landis introduced the class of *AVL-trees*, as they are called now. These trees satisfy the constraint that for each internal node v, the left and right subtrees of v have heights that differ by at most one. They showed that the logarithmic height of these trees can be maintained by local restructuring techniques—the so-called *single and double rotations*—along the search path. Each such rotation takes $O(1)$ time. Therefore, using AVL-trees, member queries, insertions and deletions all can be carried out in $O(\log n)$ time. For details, see [1, 30, 67].

Later, many other classes of binary search trees were introduced, in which these three operations can be carried out in logarithmic time. Two of these classes have properties that are particularly useful for our purposes. The first one are the BB[α]-trees, introduced by Nievergelt and Reingold [37] in 1973.

Definition 2.2.1 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 $1 - \alpha$.

Nodes that contain only a small number of leaves in their subtree do not have to satisfy this balance condition, except in case of Theorem 2.2.1 below, where all nodes should satisfy the condition. Obviously, in a BB[α]-tree the same balance condition holds for the right subtree of each internal node. The following theorem is due to Blum and Mehlhorn [13].

Theorem 2.2.1 *Let $2/11 < \alpha \leq 1 - \sqrt{2}/2$. A BB[$\alpha$]-tree for a set of n objects has size $O(n)$ and can be built in $O(n \log n)$ time. If we have the objects in sorted order, the tree can be built in $O(n)$ time. In this tree, member queries can be performed in $O(\log n)$ time. Insertions and*

deletions can be performed in $O(\log n)$ time in the worst case, where the tree is rebalanced by means of single and double rotations.

There is another way to maintain BB[α]-trees. This technique—the *partial rebuilding technique* of Lueker [31]—gives an amortized update complexity of $O(\log n)$.

Suppose we want to insert or delete object p in the leaf search BB[α]-tree T. Then we search for p, and we perform the update. 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 2.2.1 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.

Lemma 2.2.1 *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.*

Proof. The proof given here is taken from Overmars [42]. Let n'_v, n'_{lv} and n'_{rv} be the number of leaves in the subtree of v, the left son of v and the right son of v, respectively, at the moment that v is in perfect balance. Assume that $n'_{lv} \leq n'_{rv}$. Clearly, the fastest way for node v to get out of balance, is by deleting objects from its left subtree, and by inserting objects into its right subtree. Suppose N_i insertions have been carried out in the right subtree of v, and N_d deletions in the left subtree of v, at the moment that v gets out of balance. Let n_v resp. n_{lv} be the number of leaves in the subtree of v resp. the left son of v, at the moment v gets out of balance. Then $n_v = n'_v + N_i - N_d$ and $n_{lv} = n'_{lv} - N_d = \lfloor n'_v/2 \rfloor - N_d$. Since at this moment node v is out of balance, we have $n_{lv}/n_v < \alpha$. It follows that

$$\alpha\, n_v > n_{lv} = \lfloor \frac{n'_v}{2} \rfloor - N_d \geq \frac{n'_v}{2} - 1 - N_d = \frac{n_v - N_i + N_d}{2} - 1 - N_d.$$

Hence $\alpha\, n_v > n_v/2 - N_i/2 + N_d/2 - 1 - N_d = n_v/2 - (N_i + N_d)/2 - 1$, or, equivalently, $N_i + N_d > (1-2\alpha)n_v - 2$, i.e., at least $(1-2\alpha)n_v - 2$ updates have been carried out. □

Theorem 2.2.2 *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 2.2.1, 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 $U(n) = O(\log n)$. □

BB[α]-trees have properties that make them useful in many applications. One of the most important properties is the fact that the above partial rebuilding technique can be applied. This fact is fundamental in the next section, where we use these trees as building blocks for range trees. Also, in Chapter 11 these trees turn out to be particularly useful.

BB[α]-trees have as a disadvantage, however, that if they are maintained by means of rotations, $\Omega(\log n)$ rotations may be necessary in one single update. The next class of balanced binary search trees, introduced by Olivié [38, 39, 40], has the interesting property that they can be maintained in logarithmic time using at most a constant number of rotations. (See also Guibas and Sedgewick [23] and Tarjan [59].) In fact, this class is the only known class of binary search trees that has this property.

Definition 2.2.2 Let α be a real number, $0 \leq \alpha \leq 1$. A binary tree is called an *αBB-tree*, if for each internal node v, the length s_v of the

shortest path from v to a leaf and the length l_v of the longest path from v to a leaf satisfy $s_v/l_v \geq \alpha$ if $l_v \geq 1/(1-\alpha)$, and $s_v \geq l_v - 1$ otherwise.

The proof of the following theorem can be found in [38, 39, 40].

Theorem 2.2.3 *An αBB-tree for a set of n objects has size $O(n)$ and can be built in $O(n \log n)$ time. If we have the objects in sorted order, the tree can be built in $O(n)$ time. In this tree, member queries can be performed in $O(\log n)$ time. If $0 < \alpha \leq 1/2$, insertions can be performed in $O(\log n)$ time at the cost of at most 2 rotations. If $\alpha = 1/2$, deletions can be performed in $O(\log n)$ time at the cost of at most 3 rotations. If $\alpha = 1/3$, deletions can be performed in $O(\log n)$ time at the cost of at most 4 rotations.*

2.3 Range trees

We mentioned the orthogonal range searching problem already in Section 2.1. Since the largest part of this thesis is concerned with this problem, we give a formal definition. Apparently, Knuth [30] was the first who mentioned the problem:

Definition 2.3.1 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.

In 1973, Knuth wrote on page 554 of his Volume 3:

> "No really nice data structures seem to be available for such orthogonal range queries."

Since then, many data structures have been proposed to solve the problem. For a survey of the state of the art concerning the range searching problem, up to 1979, see Bentley and Friedman [7]. More recent data structures, besides range trees, can be found in Edelsbrunner [18], Chazelle [15] and Overmars [43].

The structure we consider for the orthogonal range searching problem is the *range tree*, introduced by Bentley [5] and Lueker [31]. See also Willard and Lueker [66].

In this section, we use binary trees as leaf search trees, i.e., the objects are stored in sorted order in the leaves. As usual, internal nodes contain information to guide searches in the tree. These binary trees are the building blocks of range trees.

Definition 2.3.2 Let V be a finite subset of the d-dimensional real vector space. A *d-dimensional range tree* T, representing the set V, is defined as follows.

1. If $d = 1$, then T is a binary search tree, containing the elements of V in sorted order in its leaves.

2. If $d > 1$, then T consists of a binary 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.

For convenience, we assume that no two points in the set V are the same in some coordinate. All results in this text can be proved if this assumption is not satisfied. Then the details become, however, more tedious.

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

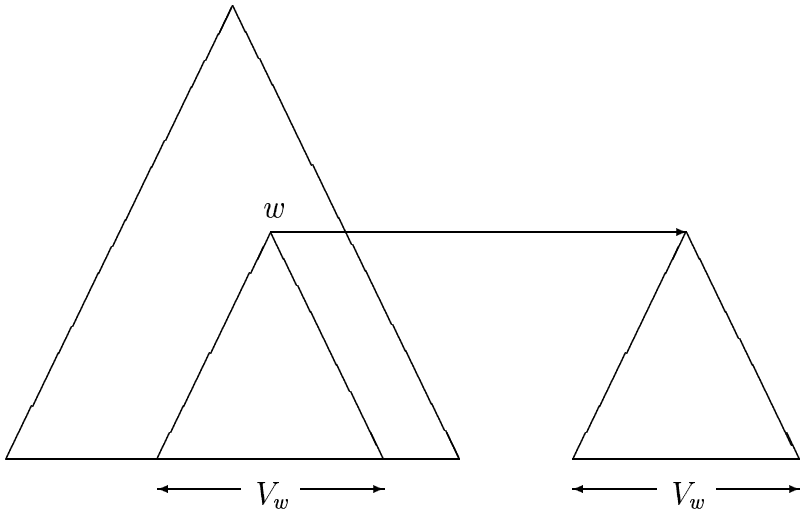

Figure 2.1: A two-dimensional range tree

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 2.1.

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. 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.

Range queries with one or more of the intervals being half-infinite or infinite are also possible. For example, let $([x_1 : \infty], [x_2 : y_2], \ldots, [x_d : y_d])$ be a half-infinite rectangle. Then we search with x_1 in the main tree. For each node v in the main tree for which the search proceeds to the left son, we perform a $(d-1)$-dimensional range query in the associated structure of the right son of v. As another example, if $x_1 = -\infty$ and $y_1 = \infty$, then we perform a $(d-1)$-dimensional range query in the associated structure of the root of the main tree.

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.

These query and update algorithms may take a lot of time, since all trees that are involved may become very unbalanced. By using BB[α]-trees, however, the query time and the amortized update time are low.

Definition 2.3.3 A *BB[α]-range tree* is a range tree, in which all binary trees are BB[α]-trees.

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 update algorithm: To update a BB[α]-range tree, we use Lueker's partial rebuilding technique. See Section 2.2. Suppose point p has to be inserted or deleted in the range tree. Then we search with the first coordinate of p in the main tree to locate its position among the leaves. During this search, we insert or delete p in all associated structures we encounter on the search path. If these associated structures are one-dimensional range trees, we use the update algorithm for BB[α]-trees that uses rotations, see [13] or Section 2.2; otherwise we use the same procedure recursively. Then we insert or delete p among the leaves of the main tree, and 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 2.2.1 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 2.2, 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 2.2.1, 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 [31]—gives the complexity of a BB[α]-range tree. Since range trees occur so often in the rest of

this thesis, and since it is important for the reader to understand this data structure, we include a proof of the theorem.

Theorem 2.3.1 *A d-dimensional BB[α]-range tree for a set of 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 [11, 49]), 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 BB[α]-range tree for a set of n points. Then, by Theorem 2.2.1, $U(n, 1) = O(\log n)$, even in the worst case. 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 BB[α]-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 2.2.1, 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 $U(n,d) = O((\log n)^d)$. □

In fact, Willard and Lueker [66] have shown that insertions and deletions can even be performed in time $O((\log n)^d)$ in the worst case, but their method is very complicated and highly unpractical.

2.4 Decomposable searching problems

There is a special class of searching problems that has been studied extensively by several authors, and that we will consider several times. These problems—the decomposable searching problems—were introduced by Bentley [5]. In fact, Bentley's paper can be marked as the beginning of the research in general dynamization techniques.

For decomposable searching problems, a query for a set of objects can be answered by merging the answers for a partition of the set. It turns out that the most interesting results can be obtained for searching problems where two answers can be merged in constant time.

Definition 2.4.1 A searching problem $PR : T_1 \times P(T_2) \to T_3$ is called *decomposable*, if there is a function $\square : T_3 \times T_3 \to T_3$, such that for any partition $V = A \cup B$ of any finite subset V of T_2, and for any query object q in T_1, we have

$$PR(q,V) = \square(PR(q,A), PR(q,B)),$$

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

For example, the member searching problem is decomposable with $\square = \vee$. Also the orthogonal range searching problem is decomposable

with $\square = \cup$. Note that, since we require the sets A and B to be disjoint, we can take in this latter problem the union of $PR(q, A)$ and $PR(q, B)$ in constant time. (Here we assume that the points that satisfy the query may be reported in any order. So we can represent the answer e.g. in a list. Clearly, two such lists can be merged in constant time.) The convex hull searching problem is not decomposable, since knowledge whether or not q is inside the convex hulls of A and B does not always tell us the position of q with respect to the convex hull of $A \cup B$.

As mentioned already, decomposable searching problems have been studied extensively. A number of techniques have been developed to design efficient dynamic data structures for these problems, especially in the case where only insertions are performed. See Bentley [5], Bentley and Saxe [8], and Overmars [42]. The main idea in these techniques is to partition the set of objects into subsets, and then to store each subset in a static data structure. In the resulting semi-dynamic data structure, queries are answered by querying the static structures separately, and by combining the answers using the function $\square$. Insertions are performed by rebuilding some small static structures together with the inserted object.

The logarithmic method: In this section we consider one dynamization technique, the logarithmic method, due to Bentley. Let DS be a static data structure for the decomposable searching problem PR. As usual, we assume that $S_{DS}(n)/n$ an $P_{DS}(n)/n$ are non-decreasing. Let V be a set of n objects, for which we want to solve the problem PR. Write n in the binary number system, i.e., $n = \sum_{i \geq 0} a_i 2^i$, where $a_i \in \{0, 1\}$. Then partition the set V into subsets V_0, V_1, V_2, etc., such that either V_i is empty or $|V_i| = 2^i$. (So $|V_i| = a_i 2^i$, $i \geq 0$.)

Our semi-dynamic data structure DS' is obtained by storing each non-empty set V_i in a static structure DS_i of type DS.

An insertion of an object p is performed as follows. Let i be the smallest index for which $a_i = 0$. Then we discard the structures DS_j for $0 \leq j \leq i - 1$. Next we build a new structure DS_i out of $V_i := V_0 \cup V_1 \cup \ldots \cup V_{i-1} \cup \{p\}$, and we set $V_0 := V_1 := \ldots := V_{i-1} := \emptyset$. Note that this new DS_i indeed represents 2^i objects.

To perform a query in the structure DS', we query each structure DS_i separately, and we combine all partial answers using the function

$\square$.

The complexity of this structure DS' is given in the following theorem. The proof can be found in [5, 8, 42]. See also Section 9.2 and Bezem and van Leeuwen [9].

Theorem 2.4.1 *The performances of the data structure DS' are given by:*

1. *The storage is bounded by $O(S_{DS}(n))$.*
2. *The query time is bounded by $O(Q_{DS}(n))$ if $Q_{DS}(n)/n^{\epsilon}$ is non-decreasing for some $\epsilon > 0$, and $O(Q_{DS}(n) \times \log n)$ otherwise.*
3. *The amortized insertion time is bounded by $O(P_{DS}(n)/n)$ if $P_{DS}(n)/n^{1+\epsilon}$ is non-decreasing for some $\epsilon > 0$, and $O((P_{DS}(n)/n) \times \log n)$ otherwise.*

The logarithmic method transforms a static data structure into a semi-dynamic one, i.e., a structure that only allows insertions. If we restrict ourselves to a subclass of the decomposable searching problems, it is possible to transform static structures into fully dynamic ones. Roughly speaking, we restrict ourselves to decomposable searching problems where the function $\square$ has an inverse, that can also be computed in constant time. This class of problems was introduced by Bentley and Saxe [8]. See also [42].

Definition 2.4.2 A decomposable searching problem $PR : T_1 \times P(T_2) \rightarrow T_3$ is called a *decomposable counting problem*, if there is a function $\triangle : T_3 \times T_3 \rightarrow T_3$, such that for any finite subset V of T_2, for any subset A of V, and for any query object q in T_1, we have

$$PR(q, V \setminus A) = \triangle(PR(q, V), PR(q, A)),$$

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

Most counting variants of decomposable searching problems are decomposable counting problems. For example, in the two-dimensional *orthogonal range counting problem*, we are given a finite set V of points

in the plane, and an axis-parallel rectangle $q = ([x_1 : y_1], [x_2 : y_2])$, and we are asked to compute the number of points of V that are in the rectangle q. This problem is clearly a decomposable searching problem. It is also a decomposable counting problem, since the number of points of $V \setminus A$ that are in the rectangle q is equal to the number of points of V in q minus the number of points of A in q.

The dynamic data structure: For decomposable counting problems we can design a fully dynamic data structure as follows. Given a static structure for the problem, we can apply the logarithmic method—or any other dynamization technique—to obtain a semi-dynamic structure. Now the dynamic data structure consists of two semi-dynamic structures DS_1 and DS_2. Initially, DS_1 stores all objects that are present, and DS_2 is empty. New objects are inserted into DS_1, whereas a deletion is performed by inserting the object that is to be deleted into DS_2. If the structure DS_2 becomes too large, we completely rebuild the structures, by building a new DS_1 storing all objects that are present at that moment, and by initializing an empty DS_2. A query is solved by querying the two structures DS_1 and DS_2, and by "subtracting" the two obtained answers from each other, using the function $\triangle$.

The complexity of the resulting dynamic data structure is given in the following theorem, the proof of which can be found in [8, 42].

Theorem 2.4.2 *Given a semi-dynamic data structure DS for the decomposable counting problem PR, there exists a fully dynamic structure solving PR, with performances:*

1. *The storage is bounded by $O(S_{DS}(n))$.*
2. *The query time is bounded by $O(Q_{DS}(n))$.*
3. *The insertion time is bounded by $O(I_{DS}(n))$.*
4. *The amortized deletion time is bounded by $O(I_{DS}(n)+P_{DS}(n)/n)$.*

2.5 Order decomposable set problems

We next consider another subclass of the searching problems. In a *set problem* we are given a set of objects, and we are asked some question about this set. To be more precise, if T_1 and T_2 are sets of objects, then a set problem is a mapping $PR : P(T_1) \rightarrow T_2$. We can consider this as a searching problem by introducing a dummy query object q.

For example, in the *convex hull problem*, we are given a finite set V of points in the d-dimensional euclidean space, and we are asked to compute the convex hull of V. Here T_1 is the set of all points in d-dimensional space, and T_2 is the set of all convex polytopes.

In this section we want to solve the problem of maintaining the answer to a set problem under insertions and deletions of objects. We restrict ourselves to set problems, the answers of which can be merged efficiently. That is, once the answers for two separated "halves" of a set are known, the answer for the entire set can be obtained fast. For such a class of set problems, we maintain the answer for the entire set, by decomposing the set into subsets, and by maintaining the answers for these subsets. These set problems were introduced by Overmars [41, 42]. See also Gowda [20] and Gowda and Kirkpatrick [21].

Definition 2.5.1 A set problem $PR : P(T_1) \rightarrow T_2$ is called $M(n)$-*order decomposable*, if there is an order ORD on T_1, and a function $\Box : T_2 \times T_2 \rightarrow T_2$, such that for any set $V = \{p_1 < p_2 < \ldots < p_n\}$, ordered according to ORD, and for any i, $1 \leq i < n$, we have

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

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

For example, Preparata and Hong [46] showed that the three-dimensional convex hull problem is $O(n)$-order decomposable, where ORD is the order according to x-coordinate.

Clearly, by using the divide-and-conquer technique, we can compute the answer to an $M(n)$-order decomposable set problem in $O(ORD(n) + R(n))$ time, where $ORD(n)$ is the time needed to order the n objects according to ORD, and $R(n) = O(\sum_{i=0}^{\log n} 2^i M(n/2^i))$ is the solution of the recurrence $R(n) = 2\,R(n/2) + M(n)$.

A dynamic data structure: Let PR be an $M(n)$-order decomposable set problem. We shortly recall a dynamic data structure solving PR, the details of which are given in [41, 42]. Let V be a set of cardinality n, for which we want to maintain the answer to PR. For simplicity we assume here that the answer $PR(V)$ takes $O(M(n))$ space to store. (In [41, 42] it is shown that this assumption is not essential).

Let $f(n)$ be a smooth integer function, such that $1 \leq f(n) \leq n$. We first order the elements of the set V according to ORD. Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be the resulting set. Now partition V into subsets $V_1 = \{p_1, \ldots, p_{f(n)}\}$, $V_2 = \{p_{f(n)+1}, \ldots, p_{2f(n)}\}$, etc. The dynamic data structure consists of the following.

- Each set V_i is stored in a balanced binary search tree T_i. Let r_i be the root of T_i. These roots are ordered according to $r_1 < r_2 < r_3 < \ldots$.
- The roots of the trees T_i are stored in the leaves of a perfectly balanced binary search tree T. Each node v of T contains the following additional information. Suppose the subtree of T with root v has $r_i, r_{i+1}, \ldots, r_j$ as its leaves. Then node v contains the answer to the set problem PR for the set $V_i \cup V_{i+1} \cup \ldots \cup V_j$. Node v also contains information to guide searches in the tree.

In particular, the root of T contains the answer to PR for the entire set V.

Update algorithm: An insertion of an object p is performed as follows. We walk down tree T to find the appropriate root r_i, and we insert p in the tree T_i. Then we rebuild the answer $PR(V_i)$, and we walk back to the root of T. For each node v we encounter during this walk, we copy the answers stored in the left and right sons of v, and we merge these copies using the function $\square$. The resulting answer is stored in v. The deletion procedure is similar.

Initially, the set V contains n elements, and each subset V_i—except the "last" one—contains $f(n)$ elements. As soon as at least one set V_i contains either $f(n)/2$ or $2f(n)$ elements—as a result of insertions and deletions—we rebuild the entire data structure. Note that we rebuild the data structure at most once every $\Omega(f(n))$ updates.

Before we state the results, we introduce the following functions $M'(n)$, $M''(n)$ and $R(n)$:

$$\begin{aligned} M'(n) &= \sum_{i=0}^{\log(n/f(n))} 2^i\, M(n/2^i), \\ M''(n) &= \sum_{i=0}^{\log(n/f(n))} M(n/2^i), \\ R(n) &= \sum_{i=0}^{\log n} 2^i\, M(n/2^i). \end{aligned}$$

The functions $M'(n)$ and $M''(n)$ satisfy:

$$M'(n) = \begin{cases} O(n/(f(n))^{1-\epsilon}) \text{ if } M(n) = O(n^\epsilon) \text{ for some } 0 \leq \epsilon < 1, \\ O(M(n)) \text{ if } M(n)/n^{1+\epsilon} \text{ is non-decreasing for some } \epsilon > 0, \\ O(M(n)\log(n/f(n))) \text{ if } M(n)/n \text{ is non-decreasing.} \end{cases}$$

$$M''(n) = \begin{cases} O(M(n)) \text{ if } M(n)/n^{\epsilon} \text{ is non-decreasing for some } \epsilon > 0, \\ O(M(n)\log(n/f(n))) \text{ if } M(n) \text{ is non-decreasing.} \end{cases}$$

The function $R(n)$ is the solution of the recurrence $R(n) = 2\,R(n/2) + M(n)$, and satisfies:

$$R(n) = \begin{cases} O(n) \text{ if } M(n) = O(n^\epsilon) \text{ for some } 0 \leq \epsilon < 1, \\ O(M(n)) \text{ if } M(n)/n^{1+\epsilon} \text{ is non-decreasing for some } \epsilon > 0, \\ O(M(n)\log n) \text{ if } M(n)/n \text{ is non-decreasing.} \end{cases}$$

Note that we have not covered here all possibilities for the function $M(n)$. The functions $M(n)$ that give rise to especially efficient solutions to the problems in Parts III and IV, however, satisfy one of the above constraints.

The proof of the following theorem can be found in [41, 42].

Theorem 2.5.1 *Let $f(n)$ be a smooth integer function, such that $1 \leq f(n) \leq n$. For an $M(n)$-order decomposable set problem, there exists a dynamic data structure, with performances:*

1. $S(n) = O(n + M'(n))$.
2. $P(n) = O(n\log n + (n/f(n)) \times R(f(n)) + M'(n))$.

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

4. $U(n) = O(\log n + R(f(n)) + M''(n) + P(n)/f(n))$, *amortized.*

A very interesting subclass are the $O(n)$-order decomposable set problems. Taking $f(n) = \lceil n/\log n \rceil$, the above theorem leads to the following corollary.

Corollary 2.5.1 *For an $O(n)$-order decomposable set problem, there exists a dynamic data structure, with performances:*

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

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

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

4. $U(n) = O(n)$, *amortized.*

Examples of $O(n)$-order decomposable set problems are computing the three-dimensional convex hull of a set of n points, see Preparata and Hong [46]; finding the intersection of a set of n halfspaces in three dimensions, see Brown [14]; and computing the view of a set of n line segments in the plane from some fixed direction, see Edelsbrunner, Overmars and Wood [19].

2.6 Storage and computation models

Until now, we assumed that there is only one storage medium in which all computations take place, and in which the data structures are stored. This one medium is called *main memory*. Most studies in the area of data structures consider this case. There are many applications, however, in which the data structure is too large to be stored in main memory. Then, the structures have to be stored in *secondary memory*. Also in case of the reconstruction problem, we have to store information in secondary memory.

Before we introduce our model of secondary memory, we say something more about main memory. All computations take place in main memory, which is modeled as a Random Access Machine (RAM). (See

Section 2.1.) As we saw already, 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.)

Next we introduce our second storage medium: *Secondary memory*. The model of secondary memory we consider is the *Indexed Sequential Model*. Just as in case of a RAM, the 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 there is the ability of direct block access: 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.

Part II

Maintaining range trees in secondary memory

Chapter 3

Introduction

3.1 The partitioning problem

In this part of the thesis 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 (a situation that very often occurs in databases). In Section 2.6 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—since operations can be viewed as paths in the data structure, we say that the operation "passes through these parts"—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 3.1.1 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 $h(n)$.

Note that it follows from 1, that the number of parts is $\Omega(S(n)/f(n))$. The relation of this definition to the above should be clear. It states, that we can store the data structure in secondary memory—which we model as the Indexed Sequential Model, see Section 2.6—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.

We study partition schemes for range trees. See Section 2.3 for the definition and the query and update algorithms for this data structure. The reader should understand this data structure thoroughly, before reading the rest of this part. In Chapters 4 and 5, we give several efficiently maintainable classes of range trees that can be partitioned in various ways. This gives a number of trade-offs between the number of disk accesses and the amount of memory that has to be transported. In each section, we change the balance condition of range trees somewhat, in order that the partition of that section can be maintained in an efficient way. The structure of the range tree, however, remains present.

In Chapter 6 we prove lower bounds for partitions of range trees. These lower bounds are proved for any range tree, in particular we do not require the trees to be balanced. Also, the lower bounds apply to any range tree (rather than to some tree) in the class of range trees. Therefore the lower bounds not only apply to worst-case bounds, but also to amortized bounds.

Since this part contains many theorems, we give a summary of the most important results in Chapter 7.

Considerable research has been done in the area of secondary memory data structures. The best-known examples are the B-trees (see

Bayer and McCreight [4], Comer [17]), which form a class of data structures that are designed for storing one-dimensional objects. In some sense, a B-tree can be seen as a partitioned binary search tree.

Another example is the class of Grid Files (see Nievergelt et al. [36], Hinrichs [25, 26]), which is a secondary memory structure for solving orthogonal range queries. The main advantage of our methods, compared to the grid files, is that we always have a worst-case upper bound on the number of disk accesses for a query in terms of the number of points and the number of reported answers. In a grid file, however, a range query can access each block in secondary memory without finding a single answer. (This will, however, not be the case in most practical situations.) Also, for our methods, the number of disk accesses for an update can always be bounded as a function of the number of points. If updates are performed in a grid file as described in [25], this is not possible, although this number will be small for most practical situations.

The idea we consider here, namely of taking a known data structure (although we make some small modifications) that was designed primarily for main memory, and investigating how it can be partitioned as efficiently as possible, is relatively new. As we said already, a B-tree is in some sense a partitioned binary tree. The partitioning of range trees has not been studied before. Other partitioned data structures, in particular priority search trees, are given in Icking, Klein and Ottmann [27] and Blankenagel and Güting [10].

3.2 Storage considerations

Before we start with our study of partitions, we consider the amount of space used in secondary memory by the partitioned data structure. It might seem that this is exactly the same amount as if the data structure were stored in main memory, but this is not true. When a part of the partition is changed during an update, the new part has to replace the old part in secondary memory. This new part only fits in the old space, if its size is not larger. But sizes of parts grow when n—the number of objects represented by the data structure—grows. When the part does not fit in the same block in secondary memory, we either have to find a

new block for it, or we have to split it. The first solution creates gaps in the file and, hence, increases the amount of space in secondary memory. The second solution increases the number of disk accesses necessary to write the part, something we clearly want to avoid.

To solve this problem, we reserve larger blocks for storing parts than is actually necessary. In this way, the block has enough room to store the part, even when it grows. To be more precise, consider an $(f(n), g(n), h(n))$-partition of a dynamic data structure. We assume that $f(n)$ is smooth and non-decreasing. Now suppose at some moment, at which the set represented by the data structure contains n_0 objects, we rebuild the entire data structure in secondary memory. Rather than using blocks of size $f(n_0)$, we use blocks of size $f(2n_0)$. As a result, as long as n—the current number of objects—is at most $2n_0$, parts still fit in their blocks. At the moment when $n = 2n_0$, we rebuild the entire data structure in secondary memory. When n becomes very small, because of a large number of deletions, the amount of storage in secondary memory becomes too large. To avoid this, we also rebuild the entire structure when $n \leq n_0/2$.

Theorem 3.2.1 *The partitioned data structure can be stored in secondary memory, using $O(S(n))$ storage, without increasing the amortized update costs—i.e. disk accesses and data transport—in order of magnitude.*

Proof. The number of parts is bounded by $O(S(n)/f(n))$. Each part requires $f(2n_0) \leq f(4n) = O(f(n))$ storage. The storage bound follows. When the entire structure has to be rebuilt, there must have been $\Omega(n_0)$ updates. Clearly, the costs for rebuilding a structure for n objects are never larger than the costs that are required for n insertions. As $n = O(n_0)$, the amortized update costs are never increased by more than a constant factor. □

Chapter 4

Partitions of 2-dimensional range trees

In this chapter, we design various partition schemes for classes of balanced two-dimensional range trees, that are all based on BB[α]-range trees. 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. In a restricted partition of a two-dimensional range tree, parts have size $\Omega(n)$, since the associated structure of the root of the main tree has size $\Omega(n)$. The second type of partitions are those in which also associated structures are partitioned into parts.

First, we give some trivial partitions of a two-dimensional BB[α]-range tree. By storing the entire tree as one part, we get an $(O(n \log n), 1, 1)$-partition. In the other extreme case, each node (either of the main tree, or of an associated structure) forms a part on its own. This gives an $(O(1), O((\log n)^2 + t),\ O((\log n)^2))$-partition, where t is the number of answers to the query. Finally, we can put each level of the main tree, together with its associated structures, in one part, leading to an $(O(n), O(\log n), O(\log n))$-partition. Note that if n is too large, so that main memory cannot contain $O(n \log n)$ data, the first partition is not a solution to our problem. However, if main memory can contain $O(n \log n)$ data, the third partition is worse than the first one: We

still have to transport an amount of $O(n \log n)$ data, and this requires $O(\log n)$ disk accesses rather than one.

4.1 Restricted partitions

In this section we consider restricted partitions of two-dimensional range trees. Although we give in later sections more efficient partitions, it is useful to consider these restricted partitions, because these partitions 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 (see e.g. Preparata and Shamos [47]), structures solving set problems like maintaining a convex hull, maintaining a Voronoi diagram, etc. (see Section 2.5), and structures for adding range restrictions to searching problems (see e.g. Bentley [5], Willard and Lueker [66]).

First we give a restricted $(O(n), O(\log\log n), O(\log\log n))$-partition of a slightly modified BB[α]-range tree. (Later, we improve this partition considerably. We include it here, however, to introduce the ideas.) The idea is as follows. Suppose we have a perfectly balanced range tree. Cut the main tree at level $\log\log n$. Each level, together with its associated structures, above level $\log\log n$ forms a part. Each such part has size $O(n)$: The associated structures on a fixed level are binary trees for subsets of the n points represented by the entire data structure, and each of these n points is in exactly one such binary tree. This gives us $O(\log\log n)$ parts, each of size $O(n)$. Each subtree having its root at level $\log\log n$, is a two-dimensional range tree, representing $O(n/\log n)$ points. Hence such a subtree has size $O((n/\log n) \times \log(n/\log n)) = O(n)$ and, hence, it can form a part. This gives us $O(\log n)$ parts, each of size $O(n)$. So in total we have $O(\log n)$ parts of size $O(n)$, provided the tree is perfectly balanced. However, as soon as we insert or delete points, the tree is not perfectly balanced anymore. In fact, the number of points represented by a subtree having its root at level $\log\log n$ can become $\Omega((1-\alpha)^{\log\log n} \times n)$.

Hence such a subtree may have size $\Omega((1-\alpha)^{\log\log n} \times n \log n)$, which is too large to form a part, since $0 < \alpha < 1/2$. Of course, we can cut the main tree at a level $\geq \log\log n$. Then, however, the number of subtrees having their root at this level, and hence the number of parts, becomes too large.

In order to avoid that subtrees having their root at level $\log\log n$, become too large, we modify the definition of range trees 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 4.1.1 A *modified range tree*, representing the set V, is defined as follows.

1. Each set V_i is stored in a two-dimensional BB[α]-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.

Note that the associated structures of the roots r_i are stored only once, whereas the roots r_i themselves are stored twice. This implies that the structure of a range tree is not changed, only the balance conditions are different.

Query and update algorithms: 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 BB[α]-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 4.1.1 *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 2.3.1. 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. □

Hence the modified range tree has (asymptotically) the same complexity as a BB[α]-range tree.

Theorem 4.1.2 *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. 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 4.1.3 *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 4.1.2 considerably. We need the following lemma. Recall our notation $(\log)^k n$ for the k-th iterated logarithm, and the definition of the function $\log^* n$. See Section 2.1.

Lemma 4.1.1 *Let the integer sequence* (a_k) *be given by* $a_0 = 0, a_{k+1} = 2^{a_k} + a_k$*, for* $k \geq 0$*. Let* n *and* d *be integers, such that* $d = \log\log n + O(1)$*. (We assume that* n *is sufficiently large.) Let* $m = \min\{i \geq 0 | a_i > d\}$*. Then* $m \leq \log^* n + O(1)$*.*

Proof. We prove by induction on i that

$$(\log)^i d \geq a_{m-i-1} \quad \text{for} \quad i = 1, 2, \ldots, m-3. \tag{4.1}$$

By definition of m, we have $d \geq a_{m-1} = 2^{a_{m-2}} + a_{m-2} \geq 2^{a_{m-2}}$. Hence $(\log)^1 d = \log d \geq a_{m-2}$. Now let $1 \leq i < m-3$, and suppose that $(\log)^i d \geq a_{m-i-1}$. Then

$$(\log)^i d \geq a_{m-i-1} = 2^{a_{m-i-2}} + a_{m-i-2} \geq 2^{a_{m-i-2}}.$$

Since $a_{m-i-2} \geq 0$, we have $(\log)^i d \geq 1$. Hence $(\log)^{i+1} d$ exists, and $(\log)^{i+1} d \geq a_{m-i-2}$, which proves (4.1).

Now take $i = m-3$ in (4.1). Then $(\log)^{m-3} d \geq a_2 = 3$, and hence $(\log)^{m-2} d \geq \log 3 > 1$. By the definition of $\log^* d$, it follows that $m - 2 < \log^* d$. Then, by using the relations $\log^*(N + O(1)) = \log^* N + O(1)$, and $\log^* N = 1 + \log^*(\log N)$, we get

$$m - 2 < \log^* d = \log^*(\log\log n + O(1)) = \log^* n + O(1).$$

□

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. The reader should note that the tree T is (and remains) perfectly balanced. So a node on level i indeed represents $\Theta(n/2^i)$ points (cf. the discussion at the beginning of this section). 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}.

Theorem 4.1.4 *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 follows from Lemma 4.1.1, 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 $n \leq 2^{65536}$.

Next, we generalize Theorem 4.1.3. Again we change the definition of range trees.

Definition 4.1.2 Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. A *k-fold modified range tree* is defined as follows.

1. For $k = 1$, a 1-fold modified range tree is a BB[α]-range tree.

2. Let $k > 1, m = \lceil n\,(\log)^k n/(\log)^{k-1} n\rceil$. Partition the set V into subsets $V_1 = \{p_1, p_2, \ldots, p_m\}, V_2 = \{p_{m+1}, \ldots, p_{2m}\}$, etc. Then a

k-fold modified range tree consists of the following. Each set V_i is stored in a $(k-1)$-fold modified 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. These roots are ordered according to $r_1 < r_2 < r_3 < \ldots$. We store these roots in a perfectly balanced binary leaf search 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 for the set $V_i \cup V_{i+1} \cup \ldots \cup V_j$, ordered according to their y-coordinates.

Query and update algorithms: In the above definition, the structure of a range tree is not changed, only the balance conditions are different. Therefore, the query algorithm in a k-fold modified range tree is similar to that of an ordinary range tree. An insertion or deletion of a point p is performed as follows. If $k = 1$, we use the update algorithm for BB[α]-range trees. Let $k > 1$. 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 the search path. Then we insert or delete p in T_i, using the update algorithm for a $(k-1)$-fold modified range tree. In order to keep the structure balanced, we completely rebuild it as soon as at least one set V_i contains either $m/2$ or $2\,m$ points.

The following theorem shows that a k-fold modified range tree has the same performances as a BB[α]-range tree.

Theorem 4.1.5 *A k-fold modified range tree, representing n points, can be built in $O(n \log n)$ time, and takes $O(n \log n)$ space to store. 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 proof is by induction on k. For $k = 1$, the theorem follows from Theorem 2.3.1. So let $k > 1$, and suppose the theorem is proved for $k-1$. Each tree T_i has size $O(m \log m)$, where $m = \lceil n\,(\log)^k n/(\log)^{k-1} n \rceil$. Since there are $O((\log)^{k-1} n/(\log)^k n)$ such trees, they take together an amount of space bounded by

$$O\left(m \log m \; \frac{(\log)^{k-1} n}{(\log)^k n}\right) = O(n \log n).$$

Each level of tree T, together with its associated structures, has size $O(n)$. Since T has height

$$O(\log(n/m)) = O\left(\log\left(\frac{(\log)^{k-1}n}{(\log)^k n}\right)\right) = O((\log)^k n),$$

this tree, together with its associated structures, has size $O(n(\log)^k n)$. Hence the entire data structure has size $O(n \log n)$. The bounds on the building time and the query time can be proved in an analogous way.

If the entire data structure is not rebuilt after an update, the given procedure takes, amortized, $O((\log)^k n \times \log n + (\log m)^2) = O((\log n)^2)$ time. Since the structure has to be rebuilt at most once every $\Omega(m)$ updates, and since this rebuilding takes $O(n \log n)$ time, it follows that the amortized update time of the k-fold modified range tree is bounded by $O((\log n)^2)$. □

The partition of a k-fold modified range tree: If $k = 1$, the entire data structure forms a part on its own. Let $k > 1$. Each tree T_i is a $(k-1)$-fold modified range tree. We partition each such tree T_i recursively into parts. Finally, the tree T, together with its associated structures, forms one part of the partition.

Theorem 4.1.6 *For a k-fold modified range tree, representing a set of n points, there exists an $(O(n(\log)^k n), 2k-1, k+o(1))$-partition.*

Proof. Again, the proof is by induction on k. For $k = 1$, the claim is obvious. So let $k > 1$, and suppose the theorem is proved for $k-1$. We saw in the proof of Theorem 4.1.5, that the tree T, together with its associated structures, has size $O(n(\log)^k n)$. So this part of the partition has the correct size. Each tree T_i is a $(k-1)$-fold modified range tree, representing $\Theta(m)$ points, where $m = \lceil n\,(\log)^k n/(\log)^{k-1} n \rceil$. By the induction hypothesis, this tree T_i is partitioned into parts of size $O(m(\log)^{k-1} m) = O(n(\log)^k n)$, such that each query passes through at most $2(k-1)-1$ parts, and each update passes, amortized, through at most $(k-1)+o(1)$ parts. Hence the entire data structure is partitioned into parts of size $O(n(\log)^k n)$. An update in a k-fold modified range tree passes, amortized, through $k+o(1)$ parts, if we do not have to rebuild the structure. Since the structure is rebuilt at most once every $\Omega(m)$

updates, and since in that case $O(\log n/(\log)^k n)$ parts are involved in the update, it follows that each update passes, amortized, through at most $k+o(1)+O((\log n/(\log)^k n)/m) = k+o(1)$ parts. We are left with the bound on the number of parts through which a query passes. Let $h(k)$ be the maximal number of parts through which the "left path" of a query in a k-fold modified range tree passes. Then $h(1) = 1$ and $h(k) \leq 1 + h(k-1)$ for $k > 1$, since we also store associated structures in the leaves of T. Hence $h(k) \leq k$. It follows that a query in the data structure passes through at most $2h(k) - 1 \leq 2k - 1$ parts: $h(k)$ parts for the left path, $h(k)$ for the right path, -1 since we counted the top part of the tree twice. This proves the theorem. □

Note that the value of k should be less than or equal to $\log^* n$, since otherwise $(\log)^k n \leq 0$, or is not even defined. Hence in practical situations, we have $k \leq 5$.

4.2 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 4.1.4. Although we prove in Theorem 6.3.1 that this is optimal for restricted partitions of normal range trees, we will 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 4.2.1 A *reduced range tree* representing the set V consists of the following.

1. Each set V_i is stored in a two-dimensional BB[α]-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 BB[α]-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 4.2.1 *A reduced range tree, representing a set of n points, can be built in $O(n \log n)$ time, and takes $O(n \log n)$ space to store. 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 BB[α]-range trees (cf. Theorem 2.3.1). 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. □

It follows that we have a new data structure for the orthogonal range searching problem, having the same performances as a BB[α]-range tree. We show that this new data structure can be partitioned efficiently.

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 4.2.2 *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. □

Remark. We prove in Theorem 6.3.1 that if a two-dimensional range tree is partitioned, in the restricted sense, such that each update passes through at most 2 parts, there must be a part of size $\Omega(n \log\log n)$. This is not in conflict with the partition of Theorem 4.2.2: A reduced range tree does not have the structure of a range tree, and therefore the lower bound does not apply. (Strictly speaking, the partition of Theorem 4.2.2 is not even restricted, since the associated structure of the root of T_i is not contained in the same part as the root itself. However, the data structure can easily be adapted such that the partition is restricted.)

4.3 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. Unfortunately, this makes the number of visited parts for a query dependent on t, the number of answers.

Definition 4.3.1 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 BB[α]-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 be 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.

In this definition, the condition $g(n) \times h(n) \geq n/\log n$ is to assure that the data structure has size $O(n \log n)$. 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

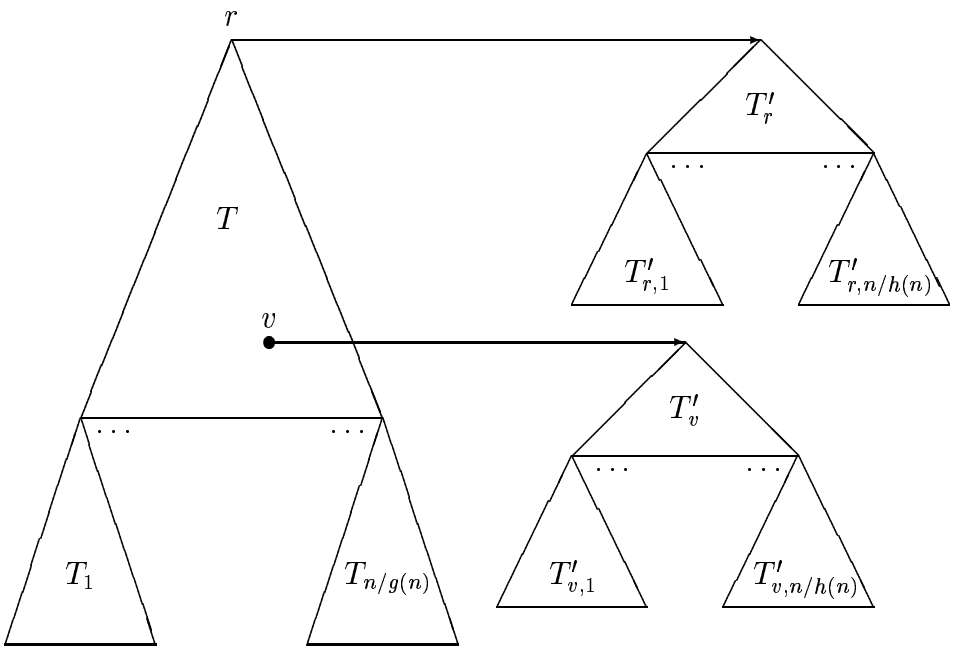

Figure 4.1: A $(g(n), h(n))$-range tree. $T_1, \ldots, T_{n/g(n)}$ are 2-dimensional BB[α]-range trees, the other trees are binary trees.

top tree T'_r associated with the root is, and remains, perfectly balanced. See Figure 4.1 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 BB[α]-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 4.3.1 *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 bound on the building time can be proved in an analogous way.

In each associated structure of a node in tree T, one-dimensional range queries can be solved in $O(\log(n/h(n)) + \log h(n)) = O(\log n)$ time. One-dimensional range queries in an associated structure of a tree T_i take $O(\log g(n)) = O(\log n)$ time. To solve a two-dimensional range query, we have to solve $O(\log n)$ one-dimensional range queries in associated structures. It follows that the query time of the data structure is bounded by $O((\log n)^2 + t)$. We are left with the update time. Suppose the data structure is not rebuilt. The update of range tree T_i takes amortized $O((\log g(n))^2)$ time, and only one such tree has to be updated. Furthermore, the update of an associated structure in T takes $O(\log n)$ time. Since $O(\log(n/g(n)))$ such associated structures are updated, the total update time is bounded by $O((\log g(n))^2 + \log n \times \log(n/g(n))) = O((\log n)^2)$. Every $\Omega(\min(g(n), h(n)))$ updates, the data structure is rebuilt at most once. Therefore, the amortized update time of the data structure is bounded by $O((\log n)^2 + (n \log n)/\min(g(n), h(n)))$. This proves the theorem. □

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 4.1 to get an impression of this partition.)

Theorem 4.3.2 *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. □

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

Corollary 4.3.1 *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 $3 + o(1)$.*

The partition in this corollary is the best result for partitions in which an update passes through at most 3 parts.

4.4 k-divided range trees

We now generalize the $(g(n), h(n))$-range tree of the preceding section, to get a class of range trees that can be partitioned such that queries and updates visit a constant number of parts.

Range trees in this new class are composed of k-divided binary trees, which are defined as follows.

Definition 4.4.1 Let k be a positive integer, and let $V = \{p_1 < p_2 < \ldots < p_n\}$ be an ordered set of n objects. A *k-divided binary tree*, representing the set V, is defined as follows.

1. For $k = 1$, a 1-divided binary tree is a BB[α]-tree, containing the elements of V in sorted order in its leaves.

2. Let $k > 1$, and let $m = \lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$. Partition V into subsets $V_1 = \{p_1, \ldots, p_m\}$, $V_2 = \{p_{m+1}, \ldots, p_{2m}\}$, etc. A k-divided binary tree consists of the following. Each set V_j is stored in a $(k-1)$-divided binary tree B_j. The roots of the trees B_j are stored in sorted order in the leaves of a perfectly balanced binary tree B.

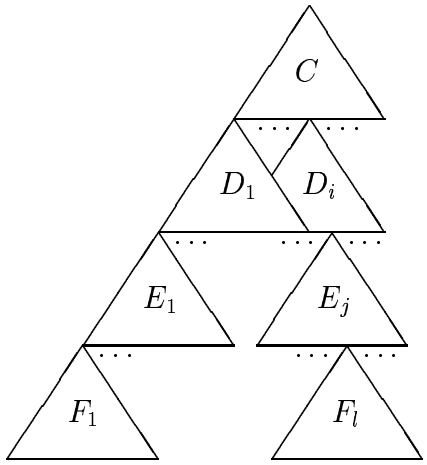

Figure 4.2: A 4-divided binary tree.

Definition 4.4.2 Consider a k-divided binary tree T representing a set V. This tree contains i-divided subtrees for $1 \leq i \leq k$. If $i > 1$, each such i-divided subtree contains a top tree, which is a binary tree storing the leaves of its $(i-1)$-divided subtrees. Such a top tree is called a *tree-part*. The BB[α]-subtrees of T that contain the objects of V—these are 1-divided binary trees—are called *bottom-parts*.

See Figure 4.2 for a pictorial representation of a 4-divided binary tree. In this figure, C, D_i and E_j are tree-parts, and the 1-divided binary tree F_l is a bottom-part. Note that a 1-divided binary tree does not contains tree-parts.

Update algorithm: An update in a k-divided binary tree is performed as follows. If $k = 1$, we use the update algorithm for BB[α]-trees that uses rotations. Let $k > 1$. To update a k-divided binary tree, we walk down tree B to find the appropriate $(k-1)$-divided binary tree B_j where the update has to be carried out. Then we perform the update in this tree B_j, using the same algorithm recursively. If this tree B_j—which initially has $m = \lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$ leaves—has either $m/2$ or $2m$ leaves, we rebuild the entire k-divided binary tree.

Lemma 4.4.1 *A k-divided binary tree representing a set of n elements has size $O(n)$ and can be built in $O(n)$ time if we have the n elements in sorted order. The tree has a height bounded by $O(\log n)$.*

Proof. The proof is trivial. □

Lemma 4.4.2 *Consider a k-divided binary tree T for a set of n elements. Let i be an integer, $1 \leq i \leq k$. Each i-divided subtree of T has size*

$$\Theta\left(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)}\right).$$

If $k > 1$, each tree-part of T has size

$$\Theta\left((n \log n)^{1/(k+1)}\right).$$

Each bottom-part has size

$$\Theta\left(n^{2/(k+1)}/(\log n)^{(k-1)/(k+1)}\right).$$

Each path in T from the root to a leaf passes through exactly $k-1$ tree-parts and one bottom-part.

Proof. Let $m_i = n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)}$. The proof of the sizes of the i-divided subtrees is by induction on i. For $i = k$, the claim is obvious. Let $1 \leq i < k$, and suppose that each $(i+1)$-divided subtree of T has size $\Theta(m_{i+1})$. Let T' be such an $(i+1)$-divided subtree. Then by definition, T' contains i-divided subtrees, each of which has size

$$\Theta\left(m_{i+1}^{(i+1)/(i+2)}/(\log m_{i+1})^{1/(i+2)}\right) = \Theta(m_i).$$

Hence the claim is true for i, since each i-divided subtree is a subtree of some $(i+1)$-divided subtree.

It is clear that each root-to-leaf path in T passes through $k-1$ tree-parts and one bottom-part. Each tree-part is the top tree of an i-divided binary tree for some $1 < i \leq k$. The size of such a tree-part is $\Theta(m_i/m_{i-1}) = \Theta((n \log n)^{1/(k+1)})$. Finally, each bottom-part has size $\Theta(m_1) = \Theta(n^{2/(k+1)}/(\log n)^{(k-1)/(k+1)})$. □

Definition 4.4.3 Let k be a positive integer. Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in the plane, ordered according to their x-coordinates. A *k-divided range tree*, representing the set V, is defined as follows. For $k = 1$, a 1-divided range tree is a BB[α]-range tree. Let $k > 1$, $m = \lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$. Partition V into subsets $V_1 = \{p_1, \ldots, p_m\}$, $V_2 = \{p_{m+1}, \ldots, p_{2m}\}$, etc. A k-divided range tree for the set V consists of the following.

1. Each set V_i is stored in a $(k-1)$-divided 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. Let r be the root of T.

3. The root r of T contains an associated structure, which is a k-divided binary tree, representing the points of V, ordered according to their y-coordinates. Let T'_r be the part of this associated structure without the bottom-parts. So T'_r consists of all tree-parts of the associated structure of r.

4. Let w be any node of T, $w \neq r$, and let V_w be the set of points represented by w. Then w contains an associated structure having the following form. The upper part is a copy of T'_r. Each leaf of this copy contains a pointer to a BB[α]-tree that contains in its leaves the—possibly none—points of V_w that "belong there", ordered according to their y-coordinates. So the entire associated structure of w contains the set V_w in its leaves, ordered according to their y-coordinates.

Finally, each node of any associated structure contains the following extra information:

- Two mark bits which state whether the left and right subtree contain points of V;

- Two extra pointers, one for the left, and one for the right subtree. One extra pointer points to the (unique) first node in the left subtree for which both subtrees contain points of V, or else (if no such node exists) to the only point of V in the left subtree. If

there are no points of V at all in the left subtree, the pointer is not used. The other extra pointer has the same meaning for the right subtree.

(End of definition.)

Note that a k-divided range tree contains a main tree as usual, which is a k-divided binary tree. In Figure 4.3, a 4-divided range tree is sketched. The highest tree-parts in all associated structures of nodes in T—denoted here by C—are identical (except for the mark bits and the extra pointers). Also, for fixed i, all tree-parts D_i in all associated structures of nodes in T are identical. The same holds for the tree-parts E_j for fixed j. The points that are contained in the BB[α]-tree F_l' in the associated structure of w form a subset of the points that are stored in the bottom-part F_l. This BB[α]-tree F_l' may be empty, whereas the bottom-part F_l is not empty. In this figure, $T_1, \ldots, T_{n/m}$ are $(k-1)$-divided range trees.

Let v be any internal node in an associated structure, and let w be its left son. If the extra information (the mark bits and the extra pointers) of node w is known, then we can compute the extra information for the left subtree of v in constant time. A similar remark holds if w is the right son of v. Hence, the extra information for an associated structure can be computed in a bottom-up fashion.

For $k = 2$, we nearly get the $(g(n), h(n))$-range tree of the preceding section. The difference is that in a 2-divided range tree, the upper parts of associated structures are identical, whereas in a $(g(n), h(n))$-range tree this is not necessarily the case.

Definition 4.4.4 Consider a k-divided range tree.

1. The BB[α]-subtrees of an associated structure that contain the points are called *bottom-parts*. The upper part T_r' of an associated structure (i.e. the tree without the bottom-parts) consists of *tree-parts*, which are defined as in Definition 4.4.2.

2. Two tree-parts (or bottom-parts) of two associated structures are *located at the same position*, if the paths for reaching these parts are identical. In other words, when the same left-right decisions are taken in each associated structure in reaching the parts.

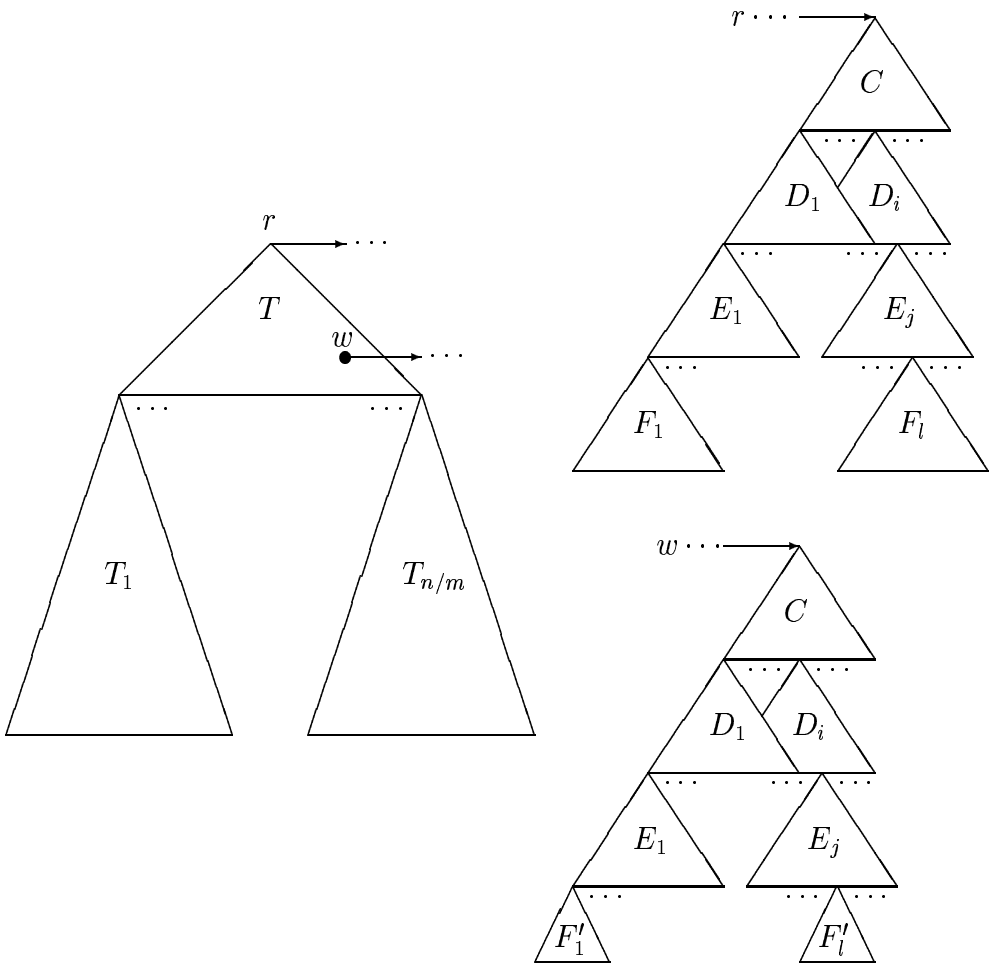

Figure 4.3: A 4-divided range tree.

So in Figure 4.3, the subtrees F_l and F'_l are bottom-parts, whereas C, D_i and E_j are tree-parts. The two tree-parts E_j—one in the associated structure of r, and the other in the associated structure of w—are located at the same position.

Building algorithm: A k-divided range tree is built as follows. If $k = 1$, we use the standard building algorithm for BB[α]-range trees. Let $k > 1$. We first order the n points to both coordinates. This takes $O(n \log n)$ time. Then we build recursively the $(k-1)$-divided range trees $T_1, \ldots, T_{n/m}$ for sets of m points, where $m = \lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$. Next, we build the tree T, in which we store the roots of these $(k-1)$-divided range trees. Then we build the associated structure of the root r of T, and we copy the part T'_r, that does not contain the bottom-parts, $O(n/m)$ times. Each copy becomes an associated structure. We complete such an associated structure, by traversing it and adding the bottom-parts containing the points that belong there, and setting the extra pointers and mark bits.

Lemma 4.4.3 *A k-divided range tree for a set of n points can be built in $O(n \log n)$ time and takes $O(n \log n)$ space to store.*

Proof. We prove the bound on the building time, by induction on k. For $k = 1$, the lemma is obvious. So let $k > 1$, and suppose that a $(k-1)$-divided range tree can be built in $O(n \log n)$ time. We build the k-divided range tree as described above. It takes $O(n \log n)$ time to order the n points to both coordinates. By the induction hypothesis, the building of the $(k-1)$-divided range trees $T_1, \ldots, T_{n/m}$ takes an amount of time bounded by $O((n/m) \times m \log m) = O(n \log n)$. Clearly, the tree T can be built in $O(n/m)$ time. The size of T'_r is equal to the size of the entire associated structure of r—which is $O(n)$—divided by the size of a bottom-part. Hence it follows from Lemma 4.4.2 that this part T'_r has size $O((n \log n)^{(k-1)/(k+1)})$. Therefore, the time to build the associated structure of r and copying T'_r $O(n/m)$ times is bounded by

$$O\left(n + (n/m) \times (n \log n)^{(k-1)/(k+1)}\right) = O(n).$$

(Note that therefore the total size of these copies is bounded by $O((n/m) \times (n \log n)^{(k-1)/(k+1)}) = O(n)$.) The completing of each copy to an associated structure takes in total $O(n)$ time for all associated structures of

a fixed level of T, since we have the points ordered according to their y-coordinates. Since there are $O(\log(n/m))$ levels in T, the total time for completing all associated structures of T is bounded by $O(n \log n)$. This proves the bound on the building time. The bound on the size can be proved in a similar way. □

Update algorithm: If $k = 1$, we use the update algorithm for BB[α]-range trees.

Let $k > 1$. An insertion or deletion of a point in a k-divided range tree is performed as follows. We walk down tree T, to find the appropriate $(k-1)$-divided range tree T_i where the update has to be carried out. During this walk we have to update all associated structures we encounter. The update in the associated structure of the root r of T is performed by using the update algorithm for k-divided binary trees.

First suppose that no rebuilding operation is necessary in the associated structure of the root r of T. (Here, a rotation in a bottom-part is not considered to be a rebuilding operation.) Then the other associated structures along the search path in T are updated in the standard way, and no rebuilding operations are carried out. In each associated structure, we adjust the extra information—the mark bits and the extra pointers—by walking backwards the path to the leaf where the point is inserted or deleted. To adjust this extra information, constant time is needed for each node on the path. Also, in case a rotation is carried out—in a bottom-part—the extra information of the nodes involved can be updated in constant time.

Otherwise, if a rebuilding operation is necessary, an i-divided subtree of the associated structure of r is rebuilt, for some i. We repeat this rebuilding in all associated structures of T. More precisely, let T_i' be the upper part of the new i-divided subtree, i.e., the tree without the bottom-parts. Then we copy T_i' $O(n/m)$ times, where $m = \lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$. In each associated structure of T we replace the old subtree by a copy of T_i', and we complete each copy by traversing it and adding the bottom-parts containing the points, and setting the extra pointers and mark bits.

Finally, we perform the update in the appropriate $(k-1)$-divided range tree T_i, using the same algorithm recursively. If this tree T_i—which initially represents m points—represents either $m/2$ or $2m$ points,

we rebuild the entire k-divided range tree.

Lemma 4.4.4 *In a k-divided range tree, representing a set of n points, insertions and deletions can be performed in amortized time $O(n^{1/(k+1)} \times (\log n)^{(k+2)/(k+1)})$.*

Proof. The proof is by induction on k. For $k = 1$, the lemma is obvious. So let $k > 1$, and suppose that an update in a $(k-1)$-divided range tree takes, amortized, $O(n^{1/k} \times (\log n)^{(k+1)/k})$ time.

In the above update algorithm, the update of the $(k-1)$-divided range tree T_i takes—by the induction hypothesis—amortized

$$O\left(m^{1/k} \times (\log m)^{(k+1)/k}\right) = O\left(n^{1/(k+1)} \times (\log n)^{(k+2)/(k+1)}\right)$$

time. The entire k-divided range tree is rebuilt at most once every $\Omega(m)$ updates. So this rebuilding adds

$$O\left((n \log n)/m\right) = O\left(n^{1/(k+1)} \times (\log n)^{(k+2)/(k+1)}\right)$$

to the amortized update time.

We are left with the update time for the associated structures of T. If no rebuilding is done, these associated structures are updated in $O((\log n)^2)$ time.

Otherwise, in the associated structure of the root r of T, an i-divided subtree is rebuilt, for some i. We repeat this in all other associated structures of T, as described above. Note that $1 < i \leq k$, since we never rebuild a 1-divided subtree. The upper part T_i' has a size which is equal to the size of an i-divided subtree divided by the size of a bottom-part. Hence, by Lemma 4.4.2, this upper part has size $O((n \log n)^{(i-1)/(k+1)})$. Therefore, the time needed to build the i-divided subtree and copying the upper part $O(n/m)$ times is bounded by

$$O\left(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)}\right) + O\left((n/m) \times (n \log n)^{(i-1)/(k+1)}\right)$$

$$= O\left(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)}\right),$$

where the leftmost term is the size of the i-divided subtree. It takes an amount of $O(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)})$ time to complete the associated structures of the nodes on a fixed level of T, since by Lemma 4.4.2,

an i-divided subtree contains $O(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)})$ points, and each of these points is contained in exactly one associated structure. (Note that all $O(n/m)$ copies of T_i' can indeed be traversed in $O(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)})$ time.) Since tree T has $O(\log(n/m))$ levels, the total amount of time to complete all associated structures of T is bounded by

$$O\left(\left(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)}\right) \times \log(n/m)\right) = O\left((n\log n)^{(i+1)/(k+1)}\right).$$

So the total time to rebuild the associated structures—for this value of i—is bounded by

$$O\left(n^{(i+1)/(k+1)}/(\log n)^{(k-i)/(k+1)} + (n\log n)^{(i+1)/(k+1)}\right)$$

$$= O\left((n\log n)^{(i+1)/(k+1)}\right). \tag{4.2}$$

(Hence the total size of all changed parts of the associated structures of T is also bounded by $O((n\log n)^{(i+1)/(k+1)})$.)

An i-divided subtree is rebuilt at most once every $\Omega(n^{i/(k+1)}/(\log n)^{(k-i+1)/(k+1)})$ updates: The subtree is rebuilt if an $(i-1)$-divided subtree gets out of balance. So the rebuilding for this value of i adds $O(n^{1/(k+1)} \times (\log n)^{(k+2)/(k+1)})$ to the amortized update time.

This can happen for $k-1$ values of i. Therefore, the amortized update time for the associated structures of T is bounded by $O(n^{1/(k+1)} \times (\log n)^{(k+2)/(k+1)})$, since k is a constant.

We have proved that the amortized update time for the entire k-divided range tree is bounded by $O(n^{1/(k+1)} \times (\log n)^{(k+2)/(k+1)})$. □

Query algorithm: The query algorithm for a k-divided range tree is as follows. Let $([x_1 : y_1], [x_2 : y_2])$ be a query rectangle. Note that the k-divided range tree contains a main tree as in the ordinary case.

1. Perform a range query in the main tree with $[x_1 : y_1]$, as in the ordinary case, and select the associated structures to be queried.

2. For each associated structure selected in step 1, perform a one-dimensional range query with $[x_2 : y_2]$ as follows: Follow the paths to x_2 and y_2, and select the subtrees in which the answers must lie. For every selected subtree, if it does not contain points, do

nothing. Otherwise, report the points it contains by following the extra pointers. (We know from the mark bits whether a subtree contains points.)

Lemma 4.4.5 *In a k-divided range tree, representing a set of n points, range queries can be solved in $O((\log n)^2+t)$ time, where t is the number of reported answers.*

Proof. The first step of the above query algorithm selects $O(\log n)$ associated structures and takes $O(\log n)$ time. The first part of the second step takes $O(\log n)$ time for each associated structure. This gives a total time of $O((\log n)^2)$. The second part of the second step guarantees that each visited node that is not on the path to x_2 or y_2 gives an extra answer to the query. More precisely, if t is the number of answers to the query, it can be shown that at most $t-1$ internal nodes are visited that are not on the path to x_2 or y_2. Hence, the second part of the second step visits at most $2t-1$ nodes. It follows that the total query time is bounded by $O((\log n)^2+t)$. □

Theorem 4.4.1 *A k-divided range tree for a set of 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 can be performed in amortized time $O(n^{1/(k+1)} \times (\log n)^{(k+2)/(k+1)})$.*

Proof. The proof follows from Lemmas 4.4.3, 4.4.4 and 4.4.5. □

The partition: We partition the k-divided range tree as follows. A 1-divided range tree forms one part on its own.

Let $k > 1$. Then we partition the $(k-1)$-divided range trees T_i recursively. We are left with the tree-part T of the main tree and its associated structures. We store the tree-parts of all associated structures of the tree-part T, that are located at the same position, in one part of the partition. (See Definition 4.4.4 for the notion "located at the same position".) Also, the bottom-parts of all associated structures of T, that are located at the same position, are stored in one part of the partition. Finally, we put the tree-part T itself in that part

of the partition that contains the highest tree-parts of the associated structures.

So, in Figure 4.3, all tree-parts C of all associated structures of T, together with T itself, are put in one part of the partition. Let i be fixed. Then all tree-parts D_i of all associated structures of T, form one part of the partition. Similarly, for fixed j, all tree-parts E_j form one part of the partition. For fixed l, all bottom-parts F_l, F'_l, etc. of all associated structures of T are stored in one part of the partition. The trees $T_1, \ldots, T_{n/m}$ are partitioned recursively.

Lemma 4.4.6 *If a k-divided range tree is partitioned as described above, each part has size $\Theta((n \log n)^{2/(k+1)})$.*

Proof. For $k = 1$, the lemma is obvious, since a 1-divided range tree is a BB[α]-range tree. So let $k > 1$, and suppose the lemma is proved for $k - 1$. Each $(k-1)$-divided range tree T_i represents $\Theta(m)$ points, where $m = \lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$. By the induction hypothesis, such a tree T_i is partitioned into parts of size

$$\Theta\left((m \log m)^{2/k}\right) = \Theta\left((n \log n)^{2/(k+1)}\right).$$

Each tree-part in a k-divided range tree has size $\Theta((n \log n)^{1/(k+1)})$. (This follows from Lemma 4.4.2.) Since we store $\Theta(n/m)$ tree-parts of associated structures of T that are located at the same position in one part of the partition, such a part has size

$$\Theta\left((n/m) \times (n \log n)^{1/(k+1)}\right) = \Theta\left((n \log n)^{2/(k+1)}\right).$$

The part of the partition that contains the tree-part T has an additional number of $\Theta(n/m)$ nodes. So this part still has size $\Theta((n \log n)^{2/(k+1)})$.

Now consider a part of the partition that contains those bottom-parts of all associated structures of tree-part T, that are located at the same position. By Lemma 4.4.2, the bottom-part of the associated structure of the root of T contains $\Theta(n^{2/(k+1)}/(\log n)^{(k-1)/(k+1)})$ points. The bottom-parts of all associated structures of a fixed level of T, that are located at the same position, together contain the same points. Hence the part of the partition containing these bottom-parts has size

$$\Theta\left(\log(n/m) \times n^{2/(k+1)}/(\log n)^{(k-1)/(k+1)}\right) = \Theta\left((n \log n)^{2/(k+1)}\right),$$

since there are $\Theta(\log(n/m))$ levels in T. Hence the lemma is true for k. $\square$

Remark. We see that all parts in the partition have asymptotically the same size. This explains our choice $\lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$ for the integer m in Definitions 4.4.1 and 4.4.3.

Lemma 4.4.7 *Each update in a k-divided range tree, partitioned as described above, passes, amortized, through at most $k(k+1)/2 + o(1)$ parts of the partition. Here, $o(1)$ is to be interpreted for $n \to \infty$.*

Proof. For $k = 1$, the lemma is obvious, since a 1-divided range tree forms one part on its own. Let $k > 1$ and suppose the lemma is proved for $k - 1$.

First suppose that we perform an update such that the entire k-divided range tree is not rebuilt, and that no rebuilding operation is necessary in the associated structure of the root of T, except for possible rotations in a bottom-part of this associated structure. By Lemma 4.4.2, this update passes through $k - 1$ tree-parts and one bottom-part of the associated structure of the root of T. Note that a rotation takes place within one bottom-part, so no extra tree- or bottom-parts are involved. The important observation is this: If the update passes through a tree-part T' of the associated structure of the root of T, this update passes through the tree-parts of other associated structures, that are located at the same position as T', and all these tree-parts are stored in the same part of the partition. The same is true for a bottom-part of the associated structure of the root of T. Since the tree T itself is stored in the same part as the highest tree-parts of the associated structures, it follows that the update of T and its associated structures passes through exactly k parts of the partition. By the induction hypothesis, the update visits, amortized, at most $k(k-1)/2+o(1)$ parts in a $(k-1)$-divided range tree T_i. So if no rebuilding is necessary, the update passes, amortized, through at most $k(k+1)/2 + o(1)$ parts of the partition.

The entire k-divided range tree is rebuilt at most once every $\Omega(m)$ updates: As soon as a tree T_i represents $m/2$ or $2m$ points, where $m = \lceil n^{k/(k+1)}/(\log n)^{1/(k+1)} \rceil$. In such a rebuilding operation, $O((n \log n)^{(k-1)/(k+1)})$

parts of the partition are involved, since by Lemma 4.4.6 all parts have size $\Theta((n \log n)^{2/(k+1)})$. It follows that this rebuilding operation adds $O((n \log n)^{(k-1)/(k+1)}/m) = o(1)$ to the amortized number of parts through which the update passes.

Finally, consider the case, where the associated structure of the root of T gets out of balance. Suppose that an i-divided subtree B is rebuilt, for some $1 < i \leq k$. This rebuilding is repeated in all other associated structures of T. The tree-parts and bottom-parts of these other associated structures that are involved, are stored in the same parts of the partition as the tree- and bottom-parts of the subtree B. All tree-parts and bottom-parts that are involved in this rebuilding operation, together have size $O((n \log n)^{(i+1)/(k+1)})$ (see Equation (4.2)), and they are partitioned into parts of size $\Theta((n \log n)^{2/(k+1)})$. Hence there are $O((n \log n)^{(i-1)/(k+1)})$ parts of the partition involved in this rebuilding operation. Since this rebuilding is done at most once every $\Omega(n^{i/(k+1)}/(\log n)^{(k-i+1)/(k+1)})$ updates—an $(i-1)$-divided subtree must get out of balance—this adds $o(1)$ to the amortized number of parts visited in an update. This can happen for $k-1$ values of i. Since k is a constant, rebuilding of the associated structures adds $o(1)$ to the number of visited parts. This completes the proof. □

Lemma 4.4.8 *A query in a k-divided range tree, partitioned as described above, passes through at most $2k^2-2k+2t$ parts of the partition, where t is the number of answers to the query.*

Proof. Let $g(k)$ denote the number of parts of the partition through which a query passes in a k-divided range tree, and let $h(k)$ denote this number for a query with the first interval being half-infinite. We do not count here the number of reported answers. Then $g(1) = 1$, and $g(k) \leq 2k - 1 + 2\,h(k-1)$ for $k > 1$. Also, $h(1) = 1$, and $h(k) \leq 2k-1+h(k-1)$ for $k > 1$. It follows that $h(k) \leq k^2$, and hence $g(k) \leq 2k^2 - 2k + 1$.

The subtrees to be reported together contain t points, thus at most $t-1$ internal nodes not on the query-paths are needed to reach the t leafs containing these t answers. (See the query algorithm and the proof of Theorem 4.4.1.) It is possible that all these points and internal nodes are situated in different parts of the partition. The number of

parts through which a query passes therefore is at most $g(k)+2t-1 \leq 2k^2-2k+2t$. $\square$

By combining Lemmas 4.4.6, 4.4.7 and 4.4.8, we get the final result.

Theorem 4.4.2 *For a k-divided range tree, there exists a partition into parts of size $\Theta((n\log n)^{2/(k+1)})$, such that an update passes, amortized, through at most $k(k+1)/2+o(1)$ parts, and a query passes through at most $2k^2-2k+2t$ parts, where t is the number of answers to the query. The term $o(1)$ is valid for $n \to \infty$.*

Of course, if the number of answers to a query is about n, it is not possible that $\Theta(n)$ parts of the partition are needed, since the partition contains only $O((n\log n)^{(k-1)/(k+1)})$ parts.

Remark. Consider again a k-divided range tree. In our definition, several associated structures are stored twice. For example, in Definition 4.4.3, we store an associated structure in the root r_i of the $(k-1)$-divided range tree T_i. This associated structure is also stored in a leaf of the tree-part T. To guarantee the upper bound for the number of parts visited in a query, this latter associated structure is needed. The associated structures in the roots of the i-divided range subtrees for $1 \leq i < k$ can be removed. Then the results of Theorems 4.4.1 and 4.4.2 still hold. In fact, we have done the same already in Definitions 4.1.1 and 4.1.2.

Chapter 5

Partitions of d-dimensional range trees

5.1 Restricted partitions of d-dimensional range trees

The restricted partitions of Section 4.1 can easily be generalized to the multi-dimensional case, as we show now. In a restricted partition of a multi-dimensional range tree, only the main tree is partitioned. Just as in the two-dimensional case, a node of the main tree and its associated structure are contained in the same part. Since the associated structure of the root of the main tree—a $(d-1)$-dimensional range tree—has size $\Omega(n(\log n)^{d-2})$, as will be shown in Subsection 6.2.2, this implies that in a restricted partition there is a part of size $\Omega(n(\log n)^{d-2})$.

First, we define modified d-dimensional range trees. Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in d-dimensional space, ordered according to their first coordinates. We split this set 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 5.1.1 A *modified d-dimensional range tree*, representing the set V, is defined as follows.

1. Each set V_i is stored in a d-dimensional BB[α]-range tree T_i. Let r_i be the root of T_i. In the roots r_i we do not store associated structures.

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 $r_i, r_{i+1}, \ldots, r_j$. Then v contains an associated structure, which is a $(d-1)$-dimensional BB[α]-range tree for the set $V_i \cup V_{i+1} \cup \ldots \cup V_j$, taking only the last $d-1$ coordinates into account.

The query and update algorithms of a modified d-dimensional range tree are similar to those in the two-dimensional case. Again, we completely rebuild the structure as soon as one set V_i contains either $\lceil n/\log n \rceil/2$ or $2\lceil n/\log n \rceil$ points. The next theorem shows that this modified range tree has the same complexity as a BB[α]-range tree.

Theorem 5.1.1 *A modified d-dimensional range tree, representing n points, can be built in $O(n(\log n)^{d-1})$ time, and takes $O(n(\log n)^{d-1})$ space to store. In this tree, 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. The proof is the same as in the two-dimensional case. Now rebuilding of the structure takes $O(n(\log n)^{d-1})$ time, and has to be done at most once every $\Omega(n/\log n)$ updates. □

It will be clear that Theorems 4.1.2 and 4.1.4 generalize to the following ones (the proofs are the same).

Theorem 5.1.2 *For a modified d-dimensional range tree, representing n points, there exists an $(O(n(\log n)^{d-2}),\ \log\log n + O(1), \log\log n + O(1))$-partition.*

Theorem 5.1.3 *For a modified d-dimensional range tree, representing n points, there exists an $(O(n(\log n)^{d-2}),\ 4\log^* n + O(1), \log^* n + O(1))$-partition.*

Next, we define k-fold modified d-dimensional range trees.

Definition 5.1.2 Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in d-dimensional space, ordered according to their first coordinates. A *k-fold modified d-dimensional range tree* is defined as follows.

1. For $k = 1$, a 1-fold modified d-dimensional range tree is a BB[α]-range tree for the set V.

2. Let $k > 1, m = \lceil n\,(\log)^k n/(\log)^{k-1} n \rceil$. Partition the set V into subsets $V_1 = \{p_1, \ldots, p_m\}$, $V_2 = \{p_{m+1}, \ldots, p_{2m}\}$, etc. Then a k-fold modified d-dimensional range tree consists of the following. Each set V_i is stored in a $(k-1)$-fold modified d-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. We store these roots in the leaves of a perfectly balanced binary tree T. Let v be any node of T, representing the roots $r_i, r_{i+1}, \ldots, r_j$. Then v contains an associated structure, which is a $(d-1)$-dimensional BB[α]-range tree for the set $V_i \cup V_{i+1} \cup \ldots \cup V_j$, taking only the last $d-1$ coordinates into account.

Also in this case, the query and update algorithms are similar to those in Section 4.1. We rebuild the data structure as soon as one set V_i contains either $m/2$ or $2m$ points.

Theorem 5.1.4 *A k-fold modified d-dimensional range tree, representing a set of n points, can be built in $O(n(\log n)^{d-1})$ time, and takes $O(n(\log n)^{d-1})$ space to store. In this tree, 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. The proof is the same as that of Theorem 4.1.5. □

Hence the k-fold modified d-dimensional range tree has the same complexity as a BB[α]-range tree.

Theorem 5.1.5 *Let k be a positive integer. For a k-fold modified d-dimensional range tree, there exists an $(O(n(\log n)^{d-2}(\log)^k n), 2k-1, k + o(1))$-partition.*

Proof. The proof is the same as that of Theorem 4.1.6. □

5.2 d-dimensional reduced range trees

The reduced range tree, which we defined in Section 4.2 for the two-dimensional case, can be generalized in two ways.

The first generalization is straightforward: Sets V_i of cardinality $\Theta(n/\log n)$ are stored in d-dimensional BB[α]-range trees, and the roots of these trees are stored in a perfectly balanced binary tree. See Definition 4.2.1 for the notation. In exactly the same way as in Theorem 4.2.1, it follows that this data structure has the same performances as a BB[α]-range tree. Also, Theorem 4.2.2 generalizes to an $(O(n(\log n)^{d-2}), 3, 2+o(1))$-partition. The details are left to the reader.

We now give the other generalization, leading to a partition into parts of size $O(n)$. Suppose we split the set V into subsets V_i of cardinality $\Theta(n/(\log n)^{d-1})$. Then we store each V_i in a d-dimensional BB[α]-range tree T_i. The roots r_i of these T_i's are stored in a binary tree T that contains no associated structures. This leads to an $(O(n), 3, 2 + o(1))$-partition, in exactly the same way as above. The query time, however, becomes

$$O\left((\log n)^{d-1} \times \left(\log\left(\frac{n}{(\log n)^{d-1}}\right)\right)^{d-1}\right) = O((\log n)^{2d-2}).$$

We can avoid this high query time, by storing associated structures in every $\log\log n$-th level in T. These associated structures may sometimes be too large to be put in one part of the partition. In that case we also split up these structures. We formalize this idea.

Definition 5.2.1 Let k and d be integers, where $k \geq -1$, $d \geq 1$ and $k < d$. Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in d-dimensional space, ordered according to their first coordinates. A *d-dimensional k-reduced range tree*, representing the set V is defined as follows.

1. A d-dimensional (-1)-reduced range tree is empty.
2. A d-dimensional 0-reduced range tree is a d-dimensional BB[α]-range tree for the set V.

3. Let $k \geq 1$. 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 $h(n) = \lceil n/\log n \rceil$. Then a d-dimensional k-reduced range tree has the following structure.
   (a) Each set V_i is stored in a d-dimensional $(k-1)$-reduced range tree T_i.
   (b) The roots r_i of these trees T_i are stored in the leaves of a perfectly balanced binary tree T.
   (c) Each root r_i contains an associated structure T_i', which is a $(d-1)$-dimensional $(k-2)$-reduced range tree, representing the set V_i, taking only the last $d-1$ coordinates into account.

Definition 5.2.2 A *d-dimensional reduced range tree* is a d-dimensional $(d-1)$-reduced range tree.

In Figure 5.1, a 4-dimensional reduced range tree is sketched. Nodes at the highest $3 \log\log n$ levels of the main tree do not contain associated structures, except those at level $\log\log n$, which contain 3-dimensional 1-reduced range trees; and those at level $2\log\log n$, which contain 3-dimensional BB[α]-range trees. Below level $3\log\log n$, the structure is the same as for BB[α]-range trees. All binary trees BT in this figure have size $\leq \log n$, but still $\Omega(\log n)$. Hence their height is $\sim \log\log n$.

Update algorithm: An update in a d-dimensional k-reduced range tree B is performed as follows. If $k = -1$, nothing is done. If $k = 0$, we use the update algorithm for BB[α]-range trees. If $k \geq 1$ we search in T for the T_i and T_i' we have to update. Then we perform the update in T_i and T_i' using the same algorithm recursively. If after the update T_i—which initially represents $\lceil n/\log n \rceil$ points—represents either $\lceil n/\log n \rceil/2$ or $2\lceil n/\log n \rceil$ points, we completely rebuild B.

Query algorithm: A query in a d-dimensional k-reduced range tree, with query rectangle $([x_1 : y_1], \ldots, [x_d : y_d])$ is solved as follows. If $k = -1$, nothing has to be done. If $k = 0$, we use the query algorithm for an ordinary range tree.

If $k \geq 1$, we do the following. Search with x_1 and y_1 in T. We then find roots r_i and r_j. If $i = j$ we perform a query with $([x_1 : y_1], \ldots, [x_d : y_d])$ in T_i. Otherwise, if $i < j$, we

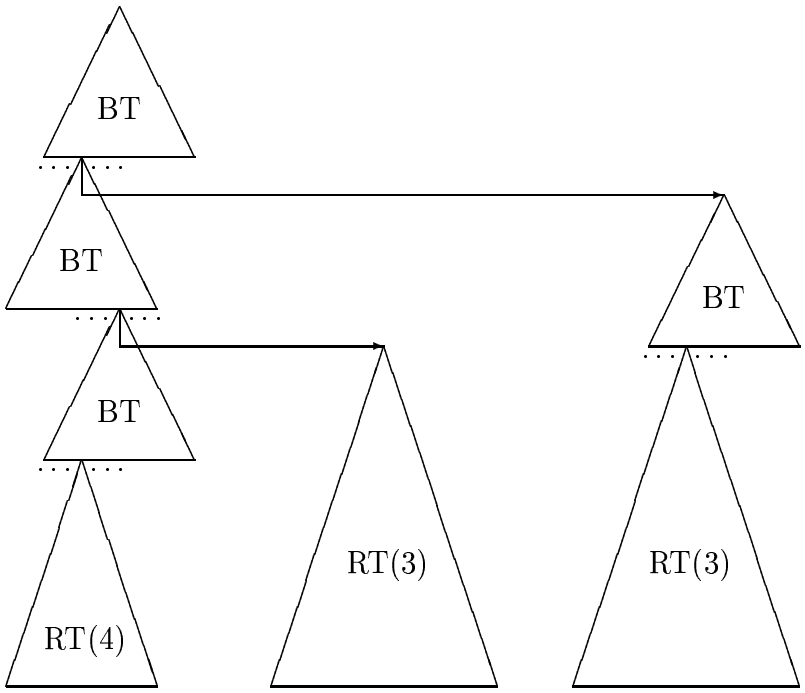

Figure 5.1: A 4-dimensional reduced range tree. BT denotes a binary tree of height $\sim \log\log n$, RT(d) a d-dimensional BB[α]-range tree.

1. perform a query with $([x_1 : \infty], [x_2 : y_2], \ldots, [x_d : y_d])$ in T_i;

2. perform a query with $([-\infty : y_1], [x_2 : y_2], \ldots, [x_d : y_d])$ in T_j;

3. (a) if $k > 1$, perform queries with $([x_2 : y_2], \ldots, [x_d : y_d])$ in the trees T'_l for all $i < l < j$;

   (b) if $k = 1$, perform queries with $([x_2 : y_2], \ldots, [x_d : y_d])$ in the associated structures of the roots of the trees T_l for all $i < l < j$. (Since $k = 1$, these associated structures are BB[α]-range trees.)

To answer a query with the half-infinite rectangle $([x_1 : \infty], [x_2 : y_2], \ldots, [x_d : y_d])$, we find the root r_i corresponding to x_1. Then we perform step 1 of the above algorithm. Finally, we perform step 3 for all $l > i$.

Theorem 5.2.1 *A d-dimensional reduced range tree, representing n points, can be built in $O(n(\log n)^{d-1})$ time and takes $O(n(\log n)^{d-1})$ space to store. Using this tree, range queries can be solved in $O((\log n)^d + t)$ time, t being the number of reported answers. Insertions and deletions in this tree can be performed in amortized time $O((\log n)^d)$.*

Proof. The bounds on the building time and the space requirements are obvious, since a reduced range tree is just a BB[α]-range tree with omission of some of the associated structures. Whether or not an associated structure has to be omitted can be decided in $O(1)$ time. The proof of the amortized update time is similar to that of the ordinary case.

Let $Q(d, k, n)$ be the worst-case query time for a d-dimensional k-reduced range tree, representing n points. We do not count in $Q(d, k, n)$ the number of reported answers. Let $R(d, k, n)$ be the worst-case query time for the same tree, for a query rectangle with the first interval being half-infinite (as in steps 1 and 2 of the above query algorithm). Again we do not count the number of answers. Then it follows from the above algorithm, the correctness of which can be seen easily, that

the following recurrence holds:

$$\begin{aligned} Q(d,0,n) &= O((\log n)^d), \\ Q(d,1,n) &\le c\log\log n \;+\; 2\,R(d,0,n/\log n) \\ &\quad +\; \log n \times O((\log(n/\log n))^{d-1}), \\ Q(d,k,n) &\le c\log\log n \;+\; 2\,R(d,k-1,n/\log n) \\ &\quad +\; \log n \times Q(d-1,k-2,n/\log n), \end{aligned}$$

for some constant c. Here, the first term on the right hand side of the last inequality is the time to find r_i and r_j; the second term is the time for steps 1 and 2; and the third term is the time for step 3. (At most $\log n$ queries are involved in this third step.) Since a query with a rectangle, one of its intervals being half-infinite, is a special instance of an orthogonal range query (e.g. in step 1 of the above query algorithm, we can choose y_1 sufficiently large), we have $R(d,k,n) \le Q(d,k,n)$. Hence

$$\begin{aligned} Q(d,1,n) &\le c\log\log n + 2\,Q(d,0,n/\log n) + \log n \times O((\log(n/\log n))^{d-1}) \\ &= O((\log n)^d), \end{aligned}$$

and for $k > 1$

$$\begin{aligned} Q(d,k,n) &\le c\log\log n + 2\,Q(d,k-1,n/\log n) \\ &\quad +\; \log n \times Q(d-1,k-2,n/\log n) \\ &\le 2\,Q(d,k-1,n/\log n) + 2\,\log n \times Q(d-1,k-2,n/\log n), \end{aligned}$$

if n is sufficiently large. It can be shown that there are constants c_j such that $Q(j,k,n) \le c_j 4^k (\log n)^j$. Hence the query time for a d-dimensional reduced range tree is bounded above by $Q(d,d-1,n) \le c_d 4^{d-1}(\log n)^d = O((\log n)^d)$. Of course, we have to add the number of reported answers. □

The partition: We inductively partition a d-dimensional k-reduced range tree. A d-dimensional (-1)-reduced range tree is empty, so it need not be stored. A d-dimensional 0-reduced range tree forms one part on its own.

Let $k \ge 1$. A d-dimensional k-reduced range tree is partitioned as follows: We store the tree T in one "special" part, that is going to

contain all trees, the nodes of which do not have associated structures. Then we partition each T_i and T_i' recursively. (To put it another way, each 0-reduced range tree forms a part on its own. The rest of the data structure is contained in the "special part".)

So in Figure 5.1, all binary trees BT are stored in the "special" part of the partition. Each of the structures RT(3) and RT(4) forms a part of the partition on its own.

Lemma 5.2.1 *If a d-dimensional reduced range tree is partitioned as described above, each part has size $O(n)$.*

Proof. The d-dimensional reduced range tree for a set of n points consists of various d_1-dimensional k-reduced range trees. It is easy to prove by induction that such a d_1-dimensional k-reduced range tree represents $\Theta(n/(\log n)^{d_1-k-1})$ points. In particular, a d_1-dimensional 0-reduced range tree, which is just an ordinary d_1-dimensional BB[α]-range tree, represents $O(n/(\log n)^{d_1-1})$ points, and hence it has size $O(n)$. Hence each part of the partition storing a 0-reduced range tree has size $O(n)$.

It remains to prove that our "special" part has size $O(n)$. Let $g(d,k,n)$ be the size of this part for a d-dimensional k-reduced range tree. Then $g(d,-1,n)=0$, $g(d,0,n)=0$, and for $k \geq 1$

$$\begin{aligned} g(d,k,n) \;\leq\; & \log n + \log n \times g(d,k-1,n/\log n) \\ & + \log n \times g(d-1,k-2,n/\log n). \end{aligned}$$

It follows that the size of our "special" part, which is $g(d,d-1,n)$, is bounded by $O((\log n)^{d-1}) = O(n)$. This proves the lemma. □

Lemma 5.2.2 *The amortized number of parts through which an update passes in a d-dimensional reduced range tree, partitioned as described above, is at most*

$$\left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^d + \frac{3}{2} \right\rfloor + o(1), (n \to \infty).$$

Proof. Suppose we have to perform an update in a d-dimensional k-reduced range tree. The algorithm we use has been given above. Note that if no rebuilding has to be done, the number of parts visited in an update in a d-dimensional k-reduced range tree only depends on the value of k.

Let s_k be the number of parts of the partition, and a_k the number of 0-reduced range trees, through which an update passes in a d-dimensional k-reduced range tree, in case no rebuilding is necessary. Then $s_k = a_k + 1$, $a_0 = 1$, $a_1 = 1$, and $a_k = a_{k-1} + a_{k-2}$ if $k \geq 2$. It follows that s_{d-1}, the number of parts we visit when updating a d-dimensional reduced range tree, is equal to the d-th Fibonacci number plus one, which is equal to (see [22, page 286])

$$\left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^d + \frac{3}{2} \right\rfloor .$$

Now we have to charge the costs we make when rebuilding the tree. Suppose we have to rebuild a d_1-dimensional k-reduced range tree B, where $k \geq 0$. Let t_k be the number of parts of the partition, and b_k the number of 0-reduced range trees that are involved. Then $t_k = b_k + 1$, $b_0 = 1$, $b_1 \leq \log n$, and $b_k \leq \log n \times (b_{k-1} + b_{k-2})$ if $k \geq 2$. It follows that $t_k = O((\log n)^k)$. Let m be the number of points represented by B. We saw already that $m = \Theta(n/(\log n)^{d_1-k-1})$. Now B has to be rebuilt at most once every $\Omega(m/\log m)$ updates. Dividing the number of visited parts for rebuilding among these $\Omega(m/\log m)$ updates gives us $O((\log n)^{d_1}/n) = O((\log n)^d/n)$ parts per update. An update can be assigned costs from every reduced range tree on the search path for this update. Let c_k be the number of (non-empty) reduced range trees we encounter on a search path for an update in a k-reduced range tree. Then $c_0 = 1$, $c_1 = 2$, and for $k \geq 2$, $c_k = 1 + c_{k-1} + c_{k-2}$. Hence $c_k \leq 2^k$.

So we have to charge every update in a d-dimensional reduced range tree for an extra $c_{d-1} \times O((\log n)^d/n) = o(1)$ visited parts for rebuilding. This proves the lemma. □

Lemma 5.2.3 *When performing a query in a d-dimensional reduced range tree, partitioned as described above, we visit at most $1+2^d(\log n)^{\lceil (d-1)/2 \rceil}$ parts.*

Proof. Let S_k be the number of 0-reduced range trees needed for a query in a k-reduced range tree. Let R_k be the number of 0-reduced range trees needed for a query with the first interval half-infinite. The total number of parts of the partition that are needed to perform a query on a d-dimensional reduced range tree is thus $S_{d-1}+1$. We then have the following recurrences: $S_0 = 1$, $S_1 \leq 2 + \log n$, and $S_k \leq 2R_{k-1} + \log n \times S_{k-2}$ if $k \geq 2$. Furthermore, $R_0 = 1$, $R_1 \leq 1 + \log n$, and $R_k \leq R_{k-1} + \log n \times S_{k-2}$ if $k \geq 2$. It follows that for $k \geq 2$,

$$R_k \leq (1 + \log n) + \log n \times \sum_{i=0}^{k-2} S_i,$$

and hence

$$S_k \leq 2 + 2\log n + 2\log n \times \sum_{i=0}^{k-3} S_i + \log n \times S_{k-2}. \tag{5.1}$$

¿From this it can be shown that $S_k \leq 2^{k+1}(\log n)^{\lceil k/2 \rceil}$. Hence a query visits at most $S_{d-1} + 1 \leq 2^d (\log n)^{\lceil (d-1)/2 \rceil} + 1$ parts of the partition. □

Combining Lemmas 5.2.1, 5.2.2 and 5.2.3 gives the final result.

Theorem 5.2.2 *For a d-dimensional reduced range tree, there exists a partition into parts of size $\Theta(n)$, such that an update passes, amortized, through at most*

$$\left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^d + \frac{3}{2} \right\rfloor + o(1), (n \to \infty)$$

parts, and a query passes through at most

$$1 + 2^d (\log n)^{\lceil (d-1)/2 \rceil}$$

parts.

Remark. We can improve the factor $(\log n)^{\lceil (d-1)/2 \rceil}$ to $(\log n)^{\lfloor (d-1)/2 \rfloor}$. What we need is $S_1 = 3$ and $R_1 = 2$ in the proof of Lemma 5.2.3. This is not true for the partition given above, but it is not hard to change the partition slightly such that $S_1 = 3$ and $R_1 = 2$ hold: We just partition

the 1-reduced range trees in the same way as the 2-dimensional reduced range trees of Section 4.2.

Remark. The number of visited parts for a query can be high, as the above theorem shows, but in practice this will seldom be the case. The number of visited parts is namely strongly dependent on the number of answers in the first coordinate. When for example the number of answers in the first coordinate is $\leq n/(\log n)^{d-1}$, only two parts are visited: The "special" part, and exactly one part containing a 0-reduced range tree. In fact, (5.1) is an equality only if the two search paths pass through the outermost T_i's. So (5.1) is an equality only when the number of answers in the first coordinate is $\geq n - 2n/(\log n)^{d-1}$.

5.3 d-dimensional k-divided range trees

The d-dimensional k-divided range tree generalizes its two-dimensional counterpart. Again, these range trees are composed of divided binary trees, defined as follows.

Definition 5.3.1 Let $s \geq 1$ and $k \geq 1$ be integers, and let $V = \{p_1 < p_2 < \ldots < p_n\}$ be an ordered set of n objects. An (s,k)-*divided binary tree*, representing the set V, is defined as follows.

1. For $k = 1$, an $(s,1)$-divided binary tree is a BB[α]-tree, containing the elements of V in sorted order in its leaves.

2. Let $k > 1$, and let $m = \lceil n^{(k+s-2)/(k+s-1)}/(\log n)^{(s-1)/(k+s-1)} \rceil$. Partition V into subsets $V_1 = \{p_1, \ldots, p_m\}$, $V_2 = \{p_{m+1}, \ldots, p_{2m}\}$, etc. An (s,k)-divided binary tree consists of the following. Each set V_j is stored in an $(s,k-1)$-divided binary tree B_j. The roots of the trees B_j are stored in sorted order in the leaves of a perfectly balanced binary tree B.

Definition 5.3.2 Consider an (s,k)-divided binary tree T representing a set V. This tree contains (s,i)-divided subtrees for $1 \leq i \leq k$. If $i > 1$, each such (s,i)-divided subtree contains a top tree, which is a binary tree storing the leaves of its $(s,i-1)$-divided subtrees. Such a top tree

is called a *tree-part*. The BB[α]-subtrees of T that contain the objects of V—these are $(s,1)$-divided binary trees—are called *bottom-parts*.

Note that (s,k)-divided binary trees have the same structure as the k-divided binary trees of Definition 4.4.1. The difference is in the choice of the integer m. The reason for the choice of m will become clear in the rest of this section.

Update algorithm: The update algorithm for an (s,k)-divided binary tree is similar to that of a k-divided binary tree. If $k=1$, we use the update algorithm for BB[α]-trees that uses rotations. If $k>1$, we walk down tree B to find the appropriate $(s,k-1)$-divided binary tree B_j where the update has to be carried out. Then we perform the update in this tree B_j, using the same algorithm recursively. If this tree B_j—which initially has $m = \lceil n^{(k+s-2)/(k+s-1)}/(\log n)^{(s-1)/(k+s-1)} \rceil$ leaves—has either $m/2$ or $2m$ leaves, we rebuild the entire (s,k)-divided binary tree.

Clearly, an (s,k)-divided binary tree for a set of n objects has size $O(n)$, can be built in $O(n)$ time if we have the n objects in sorted order, and has a height that is bounded by $O(\log n)$.

Lemma 5.3.1 *Consider an (s,k)-divided binary tree T for a set of n elements. Let i be an integer, $1 \leq i \leq k$. Each (s,i)-divided subtree of T has size*

$$\Theta\left(n^{(i+s-1)/(k+s-1)}/(\log n)^{(s-1)(k-i)/(k+s-1)}\right).$$

If $k>1$, each tree-part of T has size

$$\Theta\left(n^{1/(k+s-1)} \times (\log n)^{(s-1)/(k+s-1)}\right).$$

Each bottom-part has size

$$\Theta\left(n^{s/(k+s-1)}/(\log n)^{(s-1)(k-1)/(k+s-1)}\right).$$

Each path in T from the root to a leaf passes through exactly $k-1$ tree-parts and one bottom-part.

Proof. The proof is the same as that of Lemma 4.4.2. □

Now we define the multi-dimensional k-divided range tree.

Definition 5.3.3 Let s, d and k be integers, such that $1 \leq d \leq s$ and $k \geq 1$. Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be a set of n points in d-dimensional space, ordered according to their first coordinates. A *d-dimensional (s,k)-divided range tree*, representing the set V, is defined as follows.

For $k = 1$, a d-dimensional $(s,1)$-divided range tree is a d-dimensional BB[α]-range tree. This tree is also called a *bottom-part*, in accordance with Definition 5.3.2.

For $d = 1$, a 1-dimensional (s,k)-divided range tree is an (s,k)-divided binary tree. This tree consists of tree-parts and bottom-parts as defined in Definition 5.3.2.

Let $k \geq 2$, $d \geq 2$, $m = \lceil n^{(k+s-2)/(k+s-1)}/(\log n)^{(s-1)/(k+s-1)} \rceil$. Partition V into subsets $V_1 = \{p_1, \ldots, p_m\}$, $V_2 = \{p_{m+1}, \ldots, p_{2m}\}$, etc. A d-dimensional (s,k)-divided range tree for the set V consists of the following.

1. Each set V_i is stored in a d-dimensional $(s, k-1)$-divided range tree T_i. For this tree T_i, the notions of tree-parts and bottom-parts are recursively defined. 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. This tree T is called a *tree-part*. Let r be the root of T.

3. This root r contains an associated structure, which is a $(d-1)$-dimensional (s,k)-divided range tree, representing the points of V, taking only the last $d-1$ coordinates into account. For this associated structure, the notions of tree-parts and bottom-parts are again recursively defined. Let T'_r be the upper part of this associated structure, i.e., the tree without the bottom-parts. So T'_r consists of all tree-parts of the associated structure of r.

4. Let w be any node of T, $w \neq r$, and let V_w be the set of points represented by w. Then w contains an associated structure, having the following form. The upper part is a copy of T'_r. This copy

is completed by adding to each leaf a BB[α]-range tree of the appropriate dimension, that stores the—possibly none—points of V_w that "belong there". These BB[α]-range trees are called *bottom-parts*.

Each node in a 1-dimensional (s, k)-divided range tree contains

- two mark bits which state whether the left and right subtree contain points of V;
- two extra pointers, one for the left, and one for the right subtree. One such extra pointer points to the first node in the left subtree for which both subtrees contain points of V, or else (if no such node exists) to the only point of V in the left subtree. If there are no points of V at all in the left subtree, the pointer is not used. The other extra pointer has the same meaning for the right subtree.

(End of definition.)

So a d-dimensional (s, k)-divided range tree contains bottom-parts as subtrees. Each such bottom-part is a d'-dimensional BB[α]-range tree for some $1 \leq d' \leq d$. The range tree without the bottom-parts consists of tree-parts, which are (one-dimensional) binary trees, and which are defined as in Definition 5.3.2.

Definition 5.3.4 Let $d \geq 1$ and $k \geq 1$ be integers. A *d-dimensional k-divided range tree* is a d-dimensional (d, k)-divided range tree.

Definition 5.3.5 Consider a d-dimensional (s, k)-divided range tree. Two tree-parts (or bottom-parts) of two associated structures are *located at the same position*, if the paths for reaching these parts are identical. In other words, when the same left-right decisions are taken in each associated structure in reaching the parts.

Lemma 5.3.2 *Consider a d-dimensional (s, k)-divided range tree, representing a set of n points. Let T' be the upper part of this range tree, i.e., the range tree without the bottom-parts. If $k \geq 2$, the size of T' is bounded by*

$$\Theta\left(n^{(k+d-2)/(k+s-1)} \times (\log n)^{(s-1)(k+d-2)/(k+s-1)}\right).$$

Proof. Let $R(n, d, s, k)$ be the size of the upper part of a d-dimensional (s, k)-divided range tree for a set of n points. Then $R(n, d, s, 1) = 0$, since an $(s, 1)$-divided range tree does not have an upper part. For $d = 1$ and $k \geq 2$, the size of the upper part is proportional to the size of the binary tree divided by the size of a bottom-part. Hence it follows from Lemma 5.3.1, that

$$R(n, 1, s, k) = \Theta\left(n^{(k-1)/(k+s-1)} \times (\log n)^{(s-1)(k-1)/(k+s-1)}\right).$$

Now let $d \geq 2$ and $k \geq 2$. Then we have the following recurrence relation, which is explained below:

$$R(n, d, s, k) = \Theta(n/m) + \Theta(n/m) \times R(m, d, s, k-1) + \Theta(n/m) \times R(n, d-1, s, k),$$

where $m = \lceil n^{(k+s-2)/(k+s-1)} / (\log n)^{(s-1)/(k+s-1)} \rceil$. Here, the term $\Theta(n/m)$ is the size of the tree T that contains the roots of the d-dimensional $(s, k-1)$-divided range trees T_i. The upper part in each such range tree T_i has size $R(m, d, s, k-1)$. This explains the term $\Theta(n/m) \times R(m, d, s, k-1)$. Finally, the upper parts of all associated structures of nodes in T—all these upper parts are identical—together have size $\Theta(n/m) \times R(n, d-1, s, k)$. From this recurrence relation, the lemma can be proved by induction. □

Lemma 5.3.3 *A d-dimensional (s, k)-divided range tree, representing a set of n points, has size $O(n(\log n)^{d-1})$ and can be built in $O(n(\log n)^{d-1})$ time.*

Proof. For $k = 1$, the lemma clearly holds. For $k \geq 2$, the size of a d-dimensional (s, k)-divided range tree is equal to the sum of the size of the upper part and the total size of all bottom-parts. By Lemma 5.3.2, the upper part has size

$$O\left(n^{(k+d-2)/(k+s-1)} \times (\log n)^{(s-1)(k+d-2)/(k+s-1)}\right) = o(n),$$

since $d \leq s$. All bottom-parts together can never be larger than a d-dimensional BB[α]-range tree for a set of n points. Therefore the size of a d-dimensional (s, k)-divided range tree is bounded by $O(n(\log n)^{d-1})$. The bound on the building time can be proved in a similar way as in Lemma 4.4.3. □

Update algorithm: An update in a d-dimensional (s,k)-divided range tree is performed as follows. If $k = 1$, we use the update algorithm for BB[α]-range trees. If $d = 1$, we use the algorithm for (s,k)-divided binary trees.

Let $k \geq 2$, $d \geq 2$, and $m = \lceil n^{(k+s-2)/(k+s-1)}/(\log n)^{(s-1)/(k+s-1)} \rceil$. Then we walk down tree T, to find the appropriate d-dimensional $(s, k-1)$-divided range tree T_i where the update has to be carried out. Then we update this tree T_i, using the same algorithm recursively. As soon as one such tree T_i represents either $m/2$ or $2m$ points, we rebuild the entire d-dimensional (s,k)-divided range tree.

During the walk in T, we have to update all associated structures we encounter. The update in the associated structure of the root r of T is performed by using the same algorithm recursively for $(d-1)$-dimensional (s,k)-divided range trees. If this associated structure is a 1-dimensional structure, we use the update algorithm for (s,k)-divided binary trees.

First suppose that no rebuilding operation is necessary in the associated structure of the root r of T. (Except for perhaps a rebuilding in a bottom-part. Note that a bottom-part in this associated structure is a d'-dimensional BB[α]-range tree for some $1 \leq d' < d$.) Then the other associated structures along the search path in T are updated in the standard way, and no rebuilding operations are carried out (again, except for perhaps a rebuilding in a bottom-part). Of course, we also adjust the extra information—the mark bits and the extra pointers—in each associated structure.

Otherwise, if a rebuilding operation is necessary, a d'-dimensional (s,i)-divided range subtree of the associated structure of r is rebuilt, for some $1 \leq d' < d$ and $1 < i \leq k$. We repeat this rebuilding in all associated structures of T. More precisely, let T_i' be the upper part of the new d'-dimensional (s,i)-divided range subtree, i.e., the subtree without the bottom-parts. Then we copy T_i' $O(n/m)$ times. In each associated structure of T we replace the old subtree by a copy of T_i', and we complete each copy by traversing it and adding the bottom-parts containing the points, and setting the extra pointers and mark bits.

Lemma 5.3.4 *In a d-dimensional (s,k)-divided range tree, representing a set of n points, insertions and deletions can be performed in amor-*

tized time

$$O\left(n^{1/(k+s-1)} \times (\log n)^{d-1+(s-1)/(k+s-1)}\right).$$

Proof. Let $U(n, d, s, k)$ be the amortized update time of a d-dimensional (s, k)-divided range tree for a set of n points. Then for $k = 1$, $U(n, d, s, 1) = O((\log n)^d)$. For $d = 1$ and $k \geq 2$, the following recurrence relation follows easily from the update algorithm for (s, k)-divided binary trees:

$$U(n, 1, s, k) \leq O(\log(n/m)) + U(m, 1, s, k-1) + O(n/m).$$

Let $k \geq 2$, $d \geq 2$, and $m = \lceil n^{(k+s-2)/(k+s-1)}/(\log n)^{(s-1)/(k+s-1)} \rceil$. It takes $O(\log(n/m))$ time to find the appropriate d-dimensional $(s, k-1)$-divided range tree T_i. The update of this T_i takes, amortized, $U(m, d, s, k-1)$ time. If one such tree T_i represents either $m/2$ or $2m$ points, we rebuild the entire d-dimensional (s, k)-divided range tree. Since this happens at most once every $\Omega(m)$ updates, this rebuilding adds $O(n(\log n)^{d-1}/m)$ to the amortized update time.

Now consider the update of the associated structures of T. As we saw in the update algorithm, if no rebuilding operation is necessary in the associated structure of the root r of T, the other associated structures along the search path in T are updated in the standard way. To adjust the extra information—the mark bits and the extra pointers—only $O(1)$ time is needed for each node on the path to the point to be inserted or deleted. In this case, we spend, amortized, $O((\log n)^d)$ time in the associated structures of T.

Now assume that a rebuilding operation is necessary. Then a d'-dimensional (s, i)-divided range subtree of the associated structure of r is rebuilt, for some $1 \leq d' < d$ and $1 < i \leq k$. We repeat this rebuilding in all associated structures of T, as indicated in the update algorithm. In this case, we spend an amount of time that is bounded by the following expression, which is explained below (we use the notation of the proof of Lemma 5.3.2):

$$O(n/m) \times R(n_i, d', s, i) + O(\log(n/m)) \times O(n_i(\log n_i)^{d'-1}),$$

where n_i is the number of points represented by the (s, i)-divided range tree. (The n_i's are given in Lemma 5.3.1.) Here, the first term is the amount of time needed to build the upper part T_i' and to copy it

$O(n/m)$ times. The second term is the total amount of time needed to add the bottom-parts to the copies of T_i': At each level of T, all new bottom-parts together represent n_i points, and all these bottom-parts together can never be larger than a d'-dimensional BB[α]-range tree for a set of n_i points. This explains the above expression. Since this rebuilding happens at most once every $\Omega(n_{i-1})$ updates—an $(s, i-1)$-divided range tree must get out of balance—this case adds

$$O\left((n/m) \times R(n_i, d', s, i)/n_{i-1} + \log(n/m) \times n_i (\log n_i)^{d'-1}/n_{i-1}\right)$$

$$= O\left(n^{1/(k+s-1)} \times (\log n)^{d'+(s-1)/(k+s-1)}\right)$$

$$= O\left(n^{1/(k+s-1)} \times (\log n)^{d-1+(s-1)/(k+s-1)}\right)$$

to the amortized update time. This can happen for $d-1$ values of d' and $k-1$ values of i. Since d and k are constants, rebuilding of the associated structures of T adds

$$O\left(n^{1/(k+s-1)} \times (\log n)^{d-1+(s-1)/(k+s-1)}\right)$$

to the amortized update time.

We have proved that

$$\begin{aligned} U(n, d, s, k) \;\leq\; & U(m, d, s, k-1) + O(n(\log n)^{d-1}/m) + O((\log n)^d) \\ & + O\left(n^{1/(k+s-1)} \times (\log n)^{d-1+(s-1)/(k+s-1)}\right). \end{aligned}$$

¿From the given recurrence relations, the lemma can be proved by induction. □

Query algorithm: The query algorithm is similar to the one in the two-dimensional case. Let $([x_1 : y_1], \ldots, [x_d : y_d])$ be a query rectangle.

1. Perform a range query in the main tree with $[x_1 : y_1]$, as in the ordinary case, and select the associated structures to be queried.

2. For each associated structure selected in step 1, perform recursively a $(d-1)$-dimensional range query with $([x_2 : y_2], \ldots, [x_d : y_d])$.

3. Queries in one-dimensional structures are performed by following the paths to x_d and y_d and selecting the subtrees in which the answers must lie. For every selected subtree, if it does not contain points, do nothing. Otherwise, report the points it contains by following the extra pointers. (We know from the mark bits whether a subtree contains points.)

Lemma 5.3.5 *In a d-dimensional (s,k)-divided range tree, representing a set of n points, range queries can be performed in $O((\log n)^d + t)$ time, where t is the number of reported answers.*

Proof. From the given algorithm, the bound on the query time can be proved in the same way as in Lemma 4.4.5. □

Theorem 5.3.1 *A d-dimensional k-divided range tree for a set of n points can be built in $O(n(\log n)^{d-1})$ time and takes $O(n(\log n)^{d-1})$ space to store. In this tree, range queries can be solved in $O((\log n)^d + t)$ time, where t is the number of reported answers. Insertions and deletions can be performed in amortized time $O(n^{1/(k+d-1)} \times (\log n)^{(d-1)(k+d)/(k+d-1)})$.*

Proof. The proof follows from Lemmas 5.3.3, 5.3.4 and 5.3.5. Take $s = d$ in Lemma 5.3.4. □

The partition: We partition the d-dimensional (s,k)-divided range tree into two types of parts. The first type only contains tree-parts, whereas the second type only contains bottom-parts.

A d-dimensional $(s,1)$-divided range tree forms one part in the partition—a bottom-part—on its own. A 1-dimensional (s,k)-divided range tree is an (s,k)-divided binary tree, and consists of tree-parts and bottom-parts. Each such tree- or bottom-part forms one part in the partition.

Let $d \geq 2$ and $k \geq 2$. Then we partition the d-dimensional $(s,k-1)$-divided range trees T_i recursively into parts containing only tree-parts, and parts containing only bottom-parts. Each such part forms a part in the final partition.

Next consider the associated structure of the root r of the highest tree-part T of the main tree. We partition this associated structure

into parts containing only tree-parts, and parts containing only bottom-parts. Let T'_r be the upper part of this associated structure, i.e., the tree without the bottom-parts. The upper parts of the other associated structures of nodes in T are copies of T'_r, and these are partitioned in the same way, into parts containing only tree-parts. For each part Π in the partition of T'_r, we store the copies of Π from all other associated structures of T in one part of the final partition. The tree-part T itself is put in that part of the partition that contains the highest tree-parts of the associated structures. Similarly, for each part Π' in the partition of the associated structure of r, that contains only bottom-parts: Put the bottom-parts of all associated structures of T, that are located at the same positions as the bottom-parts in Π', in one part of the final partition.

Lemma 5.3.6 *If a d-dimensional (s,k)-divided range tree, where $k \geq 2$, is partitioned as described above, each part that contains only tree-parts has size*

$$\Theta\left(n^{d/(k+s-1)} \times (\log n)^{d(s-1)/(k+s-1)}\right).$$

Proof. For $k = 2$, the lemma follows from Lemma 5.3.2, since a partition of a d-dimensional $(s,2)$-divided range tree contains only one part that contains only tree-parts, namely the upper part. For $d = 1$, the lemma follows from Lemma 5.3.1. So let $d \geq 2$ and $k \geq 3$, and suppose the lemma is proved for smaller values of d and k. Consider a d-dimensional (s,k)-divided range tree. This range tree contains d-dimensional $(s,k-1)$-divided range trees T_i representing $\Theta(m)$ points, where $m = \lceil n^{(k+s-2)/(k+s-1)}/(\log n)^{(s-1)/(k+s-1)} \rceil$. By the induction hypothesis, each such tree T_i is partitioned into parts—containing only tree-parts—of size

$$\Theta\left(m^{d/(k+s-2)} \times (\log m)^{d(s-1)/(k+s-2)}\right) = \Theta\left(n^{d/(k+s-1)} \times (\log n)^{d(s-1)/(k+s-1)}\right).$$

Each such part forms a part in the final partition. Again by the induction hypothesis, the associated structure of the root of T, which is a $(d-1)$-dimensional (s,k)-divided range tree, is partitioned into parts—that contain only tree-parts—of size

$$\Theta\left(n^{(d-1)/(k+s-1)} \times (\log n)^{(d-1)(s-1)/(k+s-1)}\right).$$

Since we store $\Theta(n/m)$ such parts—one for each associated structure of T—in one part of the final partition, this latter part has size

$$\Theta\left(n^{d/(k+s-1)} \times (\log n)^{d(s-1)/(k+s-1)}\right).$$

The part of the partition that contains the tree-part T has an additional number of $\Theta(n/m)$ nodes. So this part still has size $\Theta(n^{d/(k+s-1)} \times (\log n)^{d(s-1)/(k+s-1)})$. □

Lemma 5.3.7 *If a d-dimensional (s,k)-divided range tree is partitioned as described above, each part that contains only bottom-parts has size*

$$\Theta\left(n^{s/(k+s-1)} \times (\log n)^{d-1-(s-1)(k-1)/(k+s-1)}\right).$$

Proof. For $k = 1$, the lemma is obvious, since an $(s,1)$-divided range tree forms a part in the partition—a bottom-part—on its own. For $d = 1$, the lemma follows from Lemma 5.3.1. Let $d \geq 2$ and $k \geq 2$, and suppose the lemma is proved for smaller values of d and k. Consider a d-dimensional (s,k)-divided range tree. This tree contains d-dimensional $(s,k-1)$-divided range trees T_i representing $\Theta(m)$ points. By the induction hypothesis, each such tree T_i is partitioned into parts—containing only bottom-parts—of size

$$\Theta\left(m^{s/(k+s-2)} \times (\log m)^{d-1-(s-1)(k-2)/(k+s-2)}\right)$$

$$= \Theta\left(n^{s/(k+s-1)} \times (\log n)^{d-1-(s-1)(k-1)/(k+s-1)}\right).$$

Each such part forms a part in the final partition. Again by the induction hypothesis, the associated structure of the root of T is partitioned into parts—that contain only bottom-parts—of size

$$\Theta\left(n^{s/(k+s-1)} \times (\log n)^{d-2-(s-1)(k-1)/(k+s-1)}\right).$$

Consider such a part Π'. We add to Π', the bottom-parts of all associated structures of T, that are located at the same positions as the bottom-parts in Π'. These bottom-parts of the associated structures of a fixed level of T, together contain the same points as Π', and therefore their total size—for this level—is proportional to the size of Π'. Hence,

since T consists of $\Theta(\log(n/m))$ levels, the part of the final partition containing these bottom-parts has size

$$\Theta(\log(n/m)) \times \Theta\left(n^{s/(k+s-1)} \times (\log n)^{d-2-(s-1)(k-1)/(k+s-1)}\right)$$
$$= \Theta\left(n^{s/(k+s-1)} \times (\log n)^{d-1-(s-1)(k-1)/(k+s-1)}\right).$$

□

Lemma 5.3.8 *Let $s \geq 2$ and $k \geq 1$. Each update in a d-dimensional (s,k)-divided range tree, partitioned as described above, passes, amortized, through at most*

$$\left(\begin{array}{c} k+d-1 \\ d \end{array}\right) + o(1), \quad (n \to \infty)$$

parts of the partition.

Proof. Let $f(n,d,s,k)$ be the amortized number of parts through which an update passes in a d-dimensional (s,k)-divided range tree, representing a set of n points. Then for $k=1$, we have $f(n,d,s,1)=1$.

Let $d=1$ and $k \geq 2$. If no rebuilding operation is necessary (except for possible rotations in a bottom-part), an update in an (s,k)-divided binary tree passes through exactly $k-1$ tree-parts, and one bottom-part. Hence in this case, the update passes through k parts of the partition. Note that if a rotation is carried out in a bottom-part, no extra tree- or bottom-parts are needed. If an (s,i)-divided subtree is rebuilt, for some $1 < i \leq k$, the number of tree-parts that are involved is bounded by the size of the upper part of this (s,i)-divided subtree divided by the size of a tree-part. This upper part has size $R(n_i,1,s,i)$, where we use the notation of Lemma 5.3.2, and where n_i is the number of points represented by the (s,i)-divided subtree. So by Lemmas 5.3.1 and 5.3.2, the number of tree-parts that are involved in this rebuilding, is bounded by

$$O\left(n^{(i-2)/(k+s-1)}/(\log n)^{(s-1)(i-2)/(k+s-1)}\right).$$

Similarly, the number of bottom-parts that are involved is bounded by the size of the (s,i)-divided subtree divided by the size of a bottom-part. By Lemma 5.3.1, this number is bounded by

$$O\left(n^{(i-1)/(k+s-1)} \times (\log n)^{(s-1)(i-1)/(k+s-1)}\right).$$

Since an (s,i)-divided subtree is rebuilt at most once every

$$\Omega\left(n^{(i+s-2)/(k+s-1)}/(\log n)^{(s-1)(k-i+1)/(k+s-1)}\right)$$

updates—an $(s,i-1)$-divided subtree must get out of balance—this rebuilding adds $o(1)$ to the amortized number of parts that are visited. (Here we have used that $s \geq 2$.) This can happen for $k-1$ values of i. Since k is a constant, rebuilding of the tree adds $o(1)$ to the amortized number of visited parts. We have proved that $f(n,1,s,k) = k + o(1)$.

Let $d \geq 2$ and $k \geq 2$. An update passes, amortized, through $f(m,d,s,k-1)$ parts of the partition in one d-dimensional $(s,k-1)$-divided range tree T_i, where $m = \lceil n^{(k+s-2)/(k+s-1)}/(\log n)^{(s-1)/(k+s-1)} \rceil$. The entire range tree is rebuilt at most once every $\Omega(m)$ updates—as soon as a d-dimensional $(s,k-1)$-divided range tree T_i gets out of balance. In this rebuilding, we visit a number of parts in the partition containing only tree-parts, that is bounded by $R(n,d,s,k)$ divided by the size of a part of the partition that contains only tree-parts. (Again, we use the notation of Lemma 5.3.2.) Similarly, we visit a number of parts in the partition that contain only bottom-parts, that is bounded by the size of the entire range tree divided by the size of a part of the partition that contains only bottom-parts. From this it follows (after some calculations) that the rebuilding of the entire range tree adds $o(1)$ to the amortized number of visited parts. Finally, the update of the associated structures of nodes in T passes, amortized, through at most $f(n,d-1,s,k)$ parts of the partition, because the tree-parts and bottom-parts of the associated structures that are visited, are stored in the same parts of the partition as the tree-parts and bottom-parts of the associated structure of the root of T that are visited in the update. It follows that

$$f(n,d,s,k) = f(m,d,s,k-1) + f(n,d-1,s,k) + o(1).$$

¿From this recurrence relation the proof can be completed by induction. □

Lemma 5.3.9 *Each query in a d-dimensional (s,k)-divided range tree, partitioned as described above, passes through at most*

$$\sum_{i=0}^{d} \binom{d+k-1-i}{k-1} \binom{k-1}{i} + 2t - 1$$

parts of the partition, where t is the number of answers to the query.

Proof. Let $g(d,k)$ denote the maximal number of parts of the partition through which a query passes in a d-dimensional (s,k)-divided range tree, and let $h(d,k)$ denote this number for a query with the first interval being half-infinite. We do not count in $g(d,k)$ and $h(d,k)$ the number of parts that we have to visit for reporting the answers. Just as in the previous lemma, the number of visited parts does not depend on n and s (if n is sufficiently large).

In the same way as in the two-dimensional case, we may have to visit $2t-1$ parts for reporting the t answers. Therefore, the number of parts through which the complete query passes is bounded above by $g(d,k)+2t-1$. So it remains to prove that $g(d,k)$ is equal to the summation in the statement of the lemma.

We have $g(1,k)=2k-1$ for $k\geq 1$; $g(d,1)=1$ for $d\geq 1$; and

$$g(d,k) = 2\,h(d,k-1) + g(d-1,k) \text{ for } d\geq 2 \text{ and } k\geq 2. \tag{5.2}$$

Also, $h(1,k)=k$ for $k\geq 1$; $h(d,1)=1$ for $d\geq 1$; and

$$h(d,k) = h(d,k-1) + g(d-1,k) \text{ for } d\geq 2 \text{ and } k\geq 2. \tag{5.3}$$

Subtract twice Equation (5.3) from Equation (5.2). This gives $2\,h(d,k) = g(d,k)+g(d-1,k)$. Substitute this latter equation, with k replaced by $k-1$, in Equation (5.2). Then we get

$$g(d,k) = g(d,k-1)+g(d-1,k)+g(d-1,k-1) \text{ for } d\geq 2 \text{ and } k\geq 3. \tag{5.4}$$

In fact, Equation (5.4) also holds for $d\geq 2$ and $k=2$, as can easily be verified from Equations (5.2) and (5.3).

We transform this recurrence relation into a more symmetric one. (In this way, we decrease the amount of work needed to solve the recurrence relation.) Define $f(0,k)=1$ for $k\geq 0$, and $f(d,k)=g(d,k+1)$ for $d\geq 1$ and $k\geq 0$. Then

$$\begin{aligned} f(0,k) &= 1, \text{ if } k\geq 0,\\ f(d,0) &= 1, \text{ if } d\geq 0,\\ f(d,k) &= f(d,k-1)+f(d-1,k)+f(d-1,k-1), \text{ if } d\geq 1 \text{ and } k\geq 1. \end{aligned}$$

Using the theory of generating functions (see e.g. Graham, Knuth and Patashnik [22]), or using counting techniques due to Monier [34], it can be shown that

$$f(d,k) = \sum_{i=0}^{d} \left(\begin{array}{c} d+k-i \\ k \end{array} \right) \left(\begin{array}{c} k \\ i \end{array} \right).$$

Since $g(d,k) = f(d,k-1)$, the proof is complete. □

Remark. The recurrence relation for $f(d,k)$ and its solution in the form of the above summation do not occur in standard books, such as [22, 48]. So it seems that no closed form exists. On page 32 of [48], the following formula appears:

$$\left(\begin{array}{c} n+p \\ m \end{array} \right) = \sum_{j \geq 0} \left(\begin{array}{c} p \\ j \end{array} \right) \left(\begin{array}{c} m-j \\ n \end{array} \right).$$

This formula—which would imply a closed solution to our summation—is, however, incorrect. The correct form is

$$\left(\begin{array}{c} n+p \\ m \end{array} \right) = \sum_{j \geq 0} \left(\begin{array}{c} p \\ j \end{array} \right) \left(\begin{array}{c} n \\ m-j \end{array} \right).$$

By combining Lemmas 5.3.6, 5.3.7, 5.3.8 and 5.3.9 we get the final result. (Take $s = d$ in these lemmas.)

Theorem 5.3.2 *Let $d \geq 2$ and $k \geq 1$. A d-dimensional k-divided range tree, representing a set of n points, can be partitioned into parts of size*

$$\Theta\left(n^{d/(k+d-1)} \times (\log n)^{d(d-1)/(k+d-1)}\right),$$

such that the amortized number of parts through which an update passes is at most

$$\left(\begin{array}{c} k+d-1 \\ d \end{array} \right) + o(1) = \frac{1}{d!}k^d + O\left(k^{d-1}\right),$$

and the number of parts through which a query passes is at most

$$\sum_{i=0}^{d} \left(\begin{array}{c} d+k-1-i \\ k-1 \end{array} \right) \left(\begin{array}{c} k-1 \\ i \end{array} \right) + 2t - 1 = \frac{2^d}{d!}k^d + 2t + O\left(k^{d-1}\right),$$

where t is the number of answers to the query. The asymptotic approximations are valid for fixed d and large k. That is, the constants in the terms $O(k^{d-1})$ depend on d, but not on k.

Remark. All results of this section are valid if $d = s = 1$, except for Lemma 5.3.8. If we apply the technique of this section with $d = s = 1$, we get a partition of a binary tree into parts of size $\Theta(n^{1/k})$, such that an update passes, amortized, through at most ck parts, for some constant $c > 1$. The structure of this partitioned tree is similar to a B-tree (see [4, 17]), which can be seen, in some sense, as a clever implementation of a $(1, k)$-divided binary tree.

Chapter 6

The lower bounds

In the previous chapters, several partition schemes were given for range trees, obtaining different trade-offs between the size of the parts and the number of parts through which queries and updates pass. In the present chapter we study lower bounds for partitions. To be more precise, suppose we have a partition of a range tree into parts of size at most $f(n)$. Then we want a worst-case lower bound—in terms of $f(n)$—on the number of parts through which an update passes. Similarly, given a partition such that each update passes through at most $h(n)$ parts, we want a lower bound—in terms of $h(n)$—for the maximal size of any part. We do not consider queries here. With some work, however, all lower bounds of this chapter can be extended to the case of queries.

We prove the lower bounds for a more general form of partitions than we used so far. That is, all lower bounds also apply for partitions where parts have different size. In particular, we do not require that a partition into parts of size at most $f(n)$ contains $O(S(n)/f(n))$ parts, as we did until now.

Also, we prove the lower bounds for arbitrary range trees. We do not require trees to be balanced. See Section 2.3, where we introduced this general class of range trees. Therefore, the bounds also apply if we for example would use AVL-trees as underlying structure; choosing different balance conditions will not help in improving the bounds. Furthermore, the lower bounds apply to any individual range tree, not just to some smartly created ones. This implies that the bounds are not only worst-case lower bounds, but also amortized lower bounds.

We consider two types of partitions of range trees. The first type are the restricted partitions. Recall that a partition of a d-dimensional range tree, where $d > 1$, is called restricted, if only the main tree is partitioned, whereas associated structures are not subdivided. In such a restricted partition, a node of the main tree and its associated structure are contained in the same part. The second type of partitions we consider, are those in which also associated structures are divided into parts.

In this chapter, we define the *size* of a part in a partition as the number of nodes it contains. Of course, the size of a part together with the information stored in its nodes—such as pointers, search and balance information—is proportional to our notion of size.

For completeness, we recall our notion of the set of points that are *represented* by a node in a range tree. Consider a range tree, representing the set V. Let w be a node in this structure (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 all points of V that are in the subtree of w. Then node w represents the set V_w.

6.1 Lower bounds for binary search trees

In order to give an introduction to the ideas that are used in the proofs of the lower bounds, we first give a lower bound for partitions of one-dimensional range trees. Note that a one-dimensional range tree is just a binary tree. We need the following two lemmas.

Lemma 6.1.1 *For any non-negative integer k, we have*

$$(k/e)^k \leq k! \leq k^k,$$

where e is the basis of the natural logarithm.

Proof. The proof follows from a straightforward calculation. □

The following lemma will also be used later in the chapter.

Lemma 6.1.2 *Let T be a binary tree, having at least n leaves. Let V be a subset of the leaves, of cardinality n. Let $m \geq 1$ be a real number.*

Then the number of nodes in T, that represent at least m points of V, is at least $n/m - 1$.

Proof. The proof is by induction on n. If $1 \leq n < \lceil m \rceil$, then there are no nodes representing at least m points of V, so the number of such nodes is 0, which is at least $n/m - 1$. If $n = \lceil m \rceil$, then the root of T represents at least m points of V. So the total number of nodes in T, that represent at least m points of V, is at least 1, which is at least $n/m - 1$. Now let $n > \lceil m \rceil$, and suppose the lemma is proved for smaller values of n. Let v be a node of T that represents the entire set V, such that the left son of v represents n_1 points of V, where $1 \leq n_1 \leq n - 1$. (v need not be the root of T, since it is possible that the left or right son of the root represents the entire set V.) By the induction hypothesis, the number of nodes in the left subtree of v, that represent at least m points of V, is at least $n_1/m - 1$. Similarly, the right subtree of v contains at least $(n - n_1)/m - 1$ nodes, that represent at least m points of V. Finally, node v itself represents at least m points of V. It follows that the total number of nodes in T, that represent at least m points of V, is at least $(n_1/m - 1) + ((n - n_1)/m - 1) + 1 = n/m - 1$. This proves the lemma. □

If we take for T a balanced binary tree, we see that this bound is tight (except for constant factors). Lemma 6.1.2 enables us to prove our first lower bound. Clearly, for a binary tree from a class of $O(\log n)$-maintainable trees—e.g. a BB[α]-tree—there exists a partition into parts of constant size, such that each update passes through $O(\log n)$ parts: Put each node in a separate part. Therefore, it is sufficient to consider partitions of binary trees in which an update passes through at most $h(n)$ parts, for some function $h(n) \leq \log n$.

Theorem 6.1.1 *Let $h(n)$ be an integer function, such that $1 \leq h(n) \leq \log n$. Let T be a binary leaf search tree representing n points. Suppose the tree T is partitioned into parts, such that each update passes through at most $h(n)$ parts. Then there is a part of size at least*

$$\left(\frac{n}{h(n)!}\right)^{1/h(n)} - 1 \quad \sim \quad \frac{e}{h(n)}\, n^{1/h(n)} - 1.$$

Proof. We write $k = h(n)$. Let $m_i = \frac{n}{i!}(\frac{k!}{n})^{i/k}$ for $0 \leq i \leq k$. We first show that the m_i's are at least 1. Clearly, $m_0 = n \geq 1$.

Let $0 < i \leq k/2$. Then, by using Lemma 6.1.1,

$$m_i = n^{1-i/k}\,\frac{(k!)^{i/k}}{i!} \geq n^{1-i/k}\,\left(\frac{k}{e\,i}\right)^i \geq \sqrt{n}\,\left(\frac{2}{e}\right)^i.$$

Since $i \leq k/2 \leq (\log n)/2$, it follows that $\sqrt{n} \geq 2^i$, and hence $m_i \geq (4/e)^i \geq 1$.

Let $k/2 < i \leq k-1$. Again using Lemma 6.1.1, we get

$$\frac{m_i}{m_{i+1}} = (i+1)\left(\frac{n}{k!}\right)^{1/k} \geq \frac{i+1}{k}\,n^{1/k} \geq \frac{1}{2}\,n^{1/k} \geq 1.$$

Since $m_k = 1$, it follows that for $k/2 < i \leq k-1$:

$$m_i \geq m_{i+1} \geq m_{i+2} \geq \ldots \geq m_k = 1.$$

We have proved that $m_i \geq 1$ for $0 \leq i \leq k$.

Let P_i be the following property:

> v_i is a node in T, that represents at least m_i points. $\Pi_0, \Pi_1, \ldots, \Pi_i$ is a sequence of $i+1$ different parts of the partition. Each update in T, that passes through v_i, passes through $\Pi_0, \Pi_1, \ldots, \Pi_i$.

Now execute the following algorithm:

> v_0 := the root of T;
> Π_0 := the part of the partition that contains v_0;
> $i := 0$;
> QED := **false**;
> { property P_i holds }
> **while** $i \neq k \wedge$ **not** QED
> **do** { property P_i holds }
> Let V be the set of all nodes in the subtree of v_i that represent at least m_{i+1} points;
> **if** $V \subset \bigcup_{j=0}^{i} \Pi_j$
> **then** QED := **true**

else $v_{i+1} :=$ a node in $V \setminus \bigcup_{j=0}^{i} \Pi_j$;
$\Pi_{i+1} :=$ the part of the partition that contains v_{i+1};
{ property P_{i+1} holds }
$i := i + 1$
{ property P_i holds }
fi
od.

First note that this algorithm terminates. Also, it is not difficult to see that the algorithm correctly maintains property P_i.

Suppose that after the algorithm is completed, QED has the value **false**. Then we must have $i = k$. Also property P_k holds. So we have a node v_k that represents at least $m_k = 1$ point, and we have a sequence $\Pi_0, \Pi_1, \ldots, \Pi_k$ of $k+1$ different parts of the partition, such that each update through node v_k passes through these $k+1$ parts. But this is a contradiction, because each update in the tree passes through at most k parts.

Therefore, after the algorithm is completed, QED has the value **true**. Hence there is an i, $0 \leq i \leq k-1$, such that P_i holds, and all nodes in the subtree of v_i, that represent at least m_{i+1} points, are contained in $\bigcup_{j=0}^{i} \Pi_j$. By Lemma 6.1.2 — which may be applied since $m_{i+1} \geq 1$ — there are at least $m_i/m_{i+1} - 1$ such nodes. It follows that $|\bigcup_{j=0}^{i} \Pi_j| = \sum_{j=0}^{i} |\Pi_j| \geq m_i/m_{i+1} - 1$. Hence there is a part in the partition of size at least

$$\frac{1}{i+1} \sum_{j=0}^{i} |\Pi_j| \geq \frac{1}{i+1} \left(\frac{m_i}{m_{i+1}} - 1 \right) = \left(\frac{n}{k!} \right)^{1/k} - \frac{1}{i+1} \geq \left(\frac{n}{k!} \right)^{1/k} - 1.$$

The approximation $(\frac{n}{k!})^{1/k} \sim \frac{e}{k} n^{1/k}$ follows from Stirling's formula. This proves the theorem. □

Consider again the proof of Theorem 6.1.1. We started with a sequence $m_0, \ldots, m_k$ of real numbers, such that $m_0 = n$, and $m_i \geq 1$ for all i. Then we showed that the partition contains a part of size at least $\frac{1}{i+1}(m_i/m_{i+1} - 1)$ for some $0 \leq i \leq k-1$. Since i can take any value between 0 and $k-1$, it follows that the partition contains a part of size

at least

$$\min\left\{\frac{m_0}{m_1}-1, \frac{1}{2}\left(\frac{m_1}{m_2}-1\right), \frac{1}{3}\left(\frac{m_2}{m_3}-1\right), \ldots, \frac{1}{k}\left(\frac{m_{k-1}}{m_k}-1\right)\right\}. \quad (6.1)$$

Clearly the proof remains valid for any such sequence $m_0, m_1, \ldots, m_k$. (Note that the condition $m_i \geq 1$ is important, since otherwise Lemma 6.1.2 may not be applied. Also, the condition $m_0 = n$ is necessary to make property P_0 hold after the initialization of the algorithm.) Therefore it might be possible that for some other choice of the numbers m_i, a better lower bound follows. We show that this is not the case. That is, the value of (6.1) is at most $(\frac{n}{k!})^{1/k}$, for any sequence of real numbers $m_0 = n, m_1 \geq 1, m_2 \geq 1, \ldots, m_k \geq 1$. Take such a sequence $m_0, \ldots, m_k$. There are two possibilities.

(i) There is an i, $0 \leq i \leq k-1$, such that $m_i \leq \frac{n}{i!}(\frac{k!}{n})^{i/k}$ and $m_{i+1} > \frac{n}{(i+1)!}(\frac{k!}{n})^{(i+1)/k}$. Then

$$\frac{1}{i+1}\frac{m_i}{m_{i+1}} \leq \left(\frac{n}{k!}\right)^{1/k},$$

and hence the value of (6.1) is at most $(\frac{n}{k!})^{1/k}$.

(ii) Otherwise, for each i, $0 \leq i \leq k-1$, we have that if $m_i \leq \frac{n}{i!}(\frac{k!}{n})^{i/k}$, then $m_{i+1} \leq \frac{n}{(i+1)!}(\frac{k!}{n})^{(i+1)/k}$. Since $m_0 = n \leq \frac{n}{0!}(\frac{k!}{n})^{0/k}$, it follows that $m_1 \leq \frac{n}{1!}(\frac{k!}{n})^{1/k}$, and hence $m_2 \leq \frac{n}{2!}(\frac{k!}{n})^{2/k}, \ldots, m_{k-1} \leq \frac{n}{(k-1)!}(\frac{k!}{n})^{(k-1)/k}$, $m_k \leq \frac{n}{k!}(\frac{k!}{n})^{k/k} = 1$. Since $m_k \geq 1$, we have $m_k = 1$. We conclude that

$$\frac{1}{k}\frac{m_{k-1}}{m_k} \leq \frac{1}{k}\frac{n}{(k-1)!}\left(\frac{k!}{n}\right)^{(k-1)/k} = \left(\frac{n}{k!}\right)^{1/k},$$

hence the value of (6.1) is bounded above by $(\frac{n}{k!})^{1/k}$.

Remark. At the end of Section 5.3 we saw that there exists a class of balanced binary search trees that can be partitioned into parts of size $O(n^{1/k})$, such that the amortized number of parts through which an updates passes is bounded by $O(k)$. Therefore, the lower bound of Theorem 6.1.1 is tight (except for constant factors) for constant functions $h(n)$.

6.2 Weight estimates for range trees

6.2.1 2-dimensional range trees

Before we can prove our lower bounds, we need some lemmas that count the number of nodes in range trees satisfying some constraints. These counting lemmas are proved by induction on the dimension d. In this subsection we prove the basis for the inductive proofs, the case $d = 2$.

Lemma 6.2.1 *Let $x_0 \geq 1$ be a real number, and let $n \geq 2x_0$ be an integer. Let $f(x)$ be a convex function for $x_0 \leq x \leq n - x_0$. Then for all x, $x_0 \leq x \leq n - x_0$, we have*

$$f(x) + f(n - x) \geq 2\, f(n/2).$$

Proof. Let $x_0 \leq x \leq n - x_0$. Then

$$f(n/2) = f(x/2 + (n - x)/2) \leq f(x)/2 + f(n - x)/2,$$

by the convexity of $f(x)$. □

Lemma 6.2.2 *Let $m \geq 1$ be a real number, and let $U(n)$ be a function satisfying*

$$\begin{aligned} U(n) &\geq 0 \quad \textit{for} \;\; 1 \leq n \leq \lfloor m \rfloor, \\ U(n) &\geq n + \min_{n_1 = 1, \ldots, n-1} [U(n_1) + U(n - n_1)] \quad \textit{for} \;\; n \geq \lfloor m \rfloor + 1. \end{aligned}$$

Then $U(n) \geq n \log(n/m)$ for $n \geq 1$.

Proof. Suppose $1 \leq n \leq \lfloor m \rfloor$. Then $U(n) \geq 0 \geq n \log(n/m)$, since $\log(n/m) \leq 0$. So let $n \geq \lfloor m \rfloor + 1$, and suppose the lemma is proved for smaller values of n. Let n_1 be an integer, such that $1 \leq n_1 \leq n - 1$. (Note that $n \geq 2$, hence such an integer n_1 exists.) By the induction hypothesis, we have

$$U(n_1) + U(n - n_1) \geq n_1 \log(n_1/m) + (n - n_1) \log((n - n_1)/m).$$

Then, applying Lemma 6.2.1, with $f(x) = x \log(x/m)$ and $x_0 = 1$, we get

$$U(n_1) + U(n - n_1) \geq 2\,(n/2) \log(n/2m) = n \log(n/m) - n.$$

Since n_1 was arbitrary, it follows that

$$U(n) \geq n + \min_{1 \leq n_1 \leq n-1} [U(n_1) + U(n - n_1)] \geq n \log(n/m).$$

□

Now we are ready to prove the main results of this subsection.

Lemma 6.2.3 *Let T be a binary tree with n leaves. Let $m \geq 1$ be a real number. For each node v of T, let the weight $wt(v)$ of v be the number of leaves in its subtree. Then*

$$\sum_{v:wt(v) \geq m} wt(v) \geq n \log(n/m).$$

Proof. Let $U(n)$ denote the sum $\sum_{v:wt(v) \geq m} wt(v)$. (Strictly speaking we should define $U(n)$ to be the minimum of the expressions $\sum_{v:wt(v) \geq m} wt(v)$ over all binary trees having n leaves.) If $1 \leq n \leq \lfloor m \rfloor$, then of course $U(n) \geq 0$. So let $n \geq \lfloor m \rfloor + 1$. The root of T has weight n, which is at least m. Let n_1 be the number of leaves in the left subtree of the root of T. Then $1 \leq n_1 \leq n - 1$, and

$$U(n) \geq n + U(n_1) + U(n - n_1) \geq n + \min_{n_1 = 1, \ldots, n-1} [U(n_1) + U(n - n_1)].$$

It follows from Lemma 6.2.2, that $U(n) \geq n \log(n/m)$. □

Corollary 6.2.1 *A two-dimensional range tree, representing n points, has size at least $n \log n$.*

Proof. This follows immediately from the above lemma, by taking T the main tree of the range tree, and $m = 1$. □

The next lemma generalizes Lemma 6.1.2 to the two-dimensional case.

Lemma 6.2.4 *Let T be a two-dimensional range tree, representing at least n points. Let V be a subset of these points, of cardinality n. Let $m \geq 1$ be a real number. Then the total number of nodes in T (in the main tree, or in an associated structure) that represent at least m points of V, is at least $(n/m) \log(n/m)$.*

Proof. Let $W(n)$ be the number of nodes in T, that represent at least m points of V. (Also here we should define $W(n)$ to be the minimum of all these numbers over all range trees and all sets V of cardinality n.) If $1 \leq n \leq \lfloor m \rfloor$, then $W(n) \geq 0$. Let $n \geq \lfloor m \rfloor + 1$. Just as in the proof of Lemma 6.1.2, let v be a node in the main tree that represents the entire set V, such that the left son of v represents n_1 points of V, where $1 \leq n_1 \leq n-1$. By Lemma 6.1.2, the associated structure of v contains at least $n/m - 1$ nodes, that represent at least m points of V. Also node v itself represents at least m points of V. Hence

$$\begin{aligned} W(n) &\geq (n/m - 1) + 1 + W(n_1) + W(n - n_1) \\ &\geq n/m + \min_{n_1 = 1, \ldots, n-1} [W(n_1) + W(n - n_1)]. \end{aligned}$$

It follows that the function $U(n) = m \times W(n)$ satisfies:

$$\begin{aligned} U(n) &\geq 0 \text{ for } 1 \leq n \leq \lfloor m \rfloor, \\ U(n) &\geq n + \min_{n_1 = 1, \ldots, n-1} [U(n_1) + U(n - n_1)] \text{ for } n \geq \lfloor m \rfloor + 1. \end{aligned}$$

Then by Lemma 6.2.2, $U(n) = m \times W(n) \geq n \log(n/m)$, and hence $W(n) \geq (n/m) \log(n/m)$. □

6.2.2 d-dimensional range trees

We generalize the counting lemmas of the preceding subsection to the multi-dimensional case.

Lemma 6.2.5 (Bernoulli's inequality) *Let d be a non-negative integer, and let $h \geq 1$ be a real number. Then*

$$(h-1)^d \geq h^d - d\, h^{d-1}.$$

Proof. Induction on d. □

Lemma 6.2.6 *Let $m \geq 1$ be a real number, and let $d \geq 2$ be an integer. Suppose the function $U(n)$ satisfies*

$$\begin{aligned} U(n) &\geq 0 \quad \textit{for} \quad 1 \leq n \leq \lfloor m \rfloor, \\ U(n) &\geq n \left(\log(n/m)\right)^{d-2} + \min_{n_1 = 1, \ldots, n-1} [U(n_1) + U(n - n_1)] \quad \textit{for} \quad n \geq \lfloor m \rfloor + 1. \end{aligned}$$

Then $U(n) \geq \frac{1}{d}\, n \left(\log(n/m)\right)^{d-1}$ for $n \geq \lfloor m \rfloor + 1$.

Proof. Let n be an integer such that $\lfloor m \rfloor + 1 \leq n \leq 2m$. (Such an integer n exists.) Then $U(n) \geq n\,(\log(n/m))^{d-2} \geq \frac{1}{d}\,n\,(\log(n/m))^{d-1}$, since $0 < \log(n/m) \leq 1 \leq d$. So let $n > 2m$, and suppose the lemma is proved for smaller values of n. Let $1 \leq n_1 \leq n-1$. Since $n > 2m$, we have $n_1 \geq \lfloor m \rfloor + 1$ or $n - n_1 \geq \lfloor m \rfloor + 1$. There are three possible cases.

(i) Suppose $n_1 \geq \lfloor m \rfloor + 1$ and $n - n_1 \geq \lfloor m \rfloor + 1$. Then by the induction hypothesis,

$$
\begin{aligned}
& n \left(\log \frac{n}{m}\right)^{d-2} + U(n_1) + U(n-n_1) \\
&\quad \geq \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\, n_1 \left(\log \frac{n_1}{m}\right)^{d-1} + \frac{1}{d}\,(n-n_1) \left(\log \frac{n-n_1}{m}\right)^{d-1} \\
&\qquad \{\text{apply Lemma 6.2.1 with } \; f(x) = x\,(\log(x/m))^{d-1} \text{ and } x_0 = m\} \\
&\quad \geq \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\, n \left(\log \frac{n}{2m}\right)^{d-1} \\
&\quad = \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\, n \left(\log \frac{n}{m} - 1\right)^{d-1} \qquad \{\text{apply Lemma 6.2.5}\} \\
&\quad \geq \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\, n \left[\left(\log \frac{n}{m}\right)^{d-1} - (d-1) \left(\log \frac{n}{m}\right)^{d-2}\right] \\
&\quad = \; \frac{1}{d}\, n \left(\log \frac{n}{m}\right)^{d-1} + \frac{1}{d}\, n \left(\log \frac{n}{m}\right)^{d-2} \\
&\quad \geq \; \frac{1}{d}\, n \left(\log \frac{n}{m}\right)^{d-1}.
\end{aligned}
$$

(ii) Suppose $n_1 \geq \lfloor m \rfloor + 1$ and $n - n_1 \leq \lfloor m \rfloor$. Then, again by the induction hypothesis,

$$
\begin{aligned}
& n \left(\log \frac{n}{m}\right)^{d-2} + U(n_1) + U(n-n_1) \\
&\quad \geq \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\, n_1 \left(\log \frac{n_1}{m}\right)^{d-1} \qquad \{\text{apply } n_1 \geq n - m\} \\
&\quad \geq \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\,(n-m) \left(\log \frac{n_1}{m}\right)^{d-1} \qquad \{\text{apply } \frac{n_1}{m} \geq \frac{n}{2m} \geq 1\} \\
&\quad \geq \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\,(n-m) \left(\log \frac{n}{2m}\right)^{d-1} \\
&\quad = \; n \left(\log \frac{n}{m}\right)^{d-2} + \frac{1}{d}\,(n-m) \left(\log \frac{n}{m} - 1\right)^{d-1} \qquad \{\text{apply Lemma 6.2.5}\}
\end{aligned}
$$

$$\begin{aligned} &\geq n\left(\log\frac{n}{m}\right)^{d-2} + \frac{1}{d}(n-m)\left[\left(\log\frac{n}{m}\right)^{d-1} - (d-1)\left(\log\frac{n}{m}\right)^{d-2}\right] \\ &= \frac{1}{d}n\left(\log\frac{n}{m}\right)^{d-1} + \left[\frac{n}{d} + m - \frac{m}{d} - \frac{m}{d}\log\frac{n}{m}\right]\left(\log\frac{n}{m}\right)^{d-2} \\ &\geq \frac{1}{d}n\left(\log\frac{n}{m}\right)^{d-1}, \end{aligned}$$

since

$$\frac{n}{d} + m - \frac{m}{d} - \frac{m}{d}\log\frac{n}{m} \geq \frac{n}{d} + m - \frac{m}{d} - \frac{m}{d}\frac{n}{m} = m\left(1 - \frac{1}{d}\right) \geq 0.$$

(iii) Otherwise, $n_1 \leq \lfloor m \rfloor$ and $n - n_1 \geq \lfloor m \rfloor + 1$. Then in the same way as in case (ii), we find

$$n\left(\log\frac{n}{m}\right)^{d-2} + U(n_1) + U(n-n_1) \geq \frac{1}{d}n\left(\log\frac{n}{m}\right)^{d-1}.$$

It follows from (i), (ii) and (iii), that for $n > 2m$

$$\begin{aligned} U(n) &\geq n\left(\log\frac{n}{m}\right)^{d-2} + \min_{n_1=1,\ldots,n-1}[U(n_1) + U(n-n_1)] \\ &\geq \frac{1}{d}n\left(\log\frac{n}{m}\right)^{d-1}. \end{aligned}$$

□

Lemma 6.2.7 *Consider a d-dimensional range tree ($d \geq 2$), representing a set of n points. Let $m \geq 1$ be a real number. For each node v of the main tree, the weight $wt(v)$ of v is defined as the total number of leaves in the associated structure of v. (Here we count the leaves in the main tree of the associated structure, in associated structures of the associated structure, etc.) Then*

$$\sum_{v:v \text{ represents } \geq m \text{ points}} wt(v) \geq \frac{2}{d!}n(\log n)^{d-2}\log(n/m) \quad \textit{if} \quad n \geq \lfloor m \rfloor + 1.$$

This summation runs over all nodes v in the main tree of the d-dimensional range tree, such that v represents at least m points.

Proof. The case $d = 2$ is proved already in Lemma 6.2.3. So let $d \geq 3$, and suppose the lemma is proved for smaller values of d. Let $W(n)$ denote the sum to be estimated. Then $W(n) \geq 0$ for $1 \leq n \leq \lfloor m \rfloor$. Let $n \geq \lfloor m \rfloor + 1$. The root of the main tree represents at least m points. To estimate the weight of the root, we have to count the total number of leaves in its associated structure. By the induction hypothesis (applied for $d-1$ and $m = 1$), this weight is at least $\frac{2}{(d-1)!} n (\log n)^{d-2}$. Let n_1 be the number of points represented by the left son of the root of the main tree. Then $1 \leq n_1 \leq n-1$. Hence

$$\begin{aligned} W(n) &\geq \frac{2}{(d-1)!}\, n\, (\log n)^{d-2} + W(n_1) + W(n-n_1) \\ &\geq \frac{2}{(d-1)!}\, n\, (\log n)^{d-2} + \min_{1 \leq n_1 \leq n-1} [W(n_1) + W(n-n_1)], \end{aligned}$$

for $n \geq \lfloor m \rfloor + 1$. Now let

$$\begin{aligned} H(n) &= 0 \quad \text{for} \quad 1 \leq n \leq \lfloor m \rfloor, \\ H(n) &= \frac{(d-1)!}{2} \frac{(\log(n/m))^{d-2}}{(\log n)^{d-2}} = \frac{(d-1)!}{2} \left(1 - \frac{\log m}{\log n}\right)^{d-2} \quad \text{for} \quad n \geq \lfloor m \rfloor + 1. \end{aligned}$$

If $1 \leq n \leq \lfloor m \rfloor$, then $H(n) \times W(n) \geq 0$. Let $n \geq \lfloor m \rfloor + 1$. Then, since $H(n) \geq 0$ and since $H(n)$ is non-decreasing,

$$\begin{aligned} &H(n) \times W(n) \\ &\quad \geq n\, (\log(n/m))^{d-2} + \min_{1 \leq n_1 \leq n-1} [H(n) \times W(n_1) + H(n) \times W(n-n_1)] \\ &\quad \geq n\, (\log(n/m))^{d-2} + \min_{1 \leq n_1 \leq n-1} [H(n_1) \times W(n_1) + H(n-n_1) \times W(n-n_1)]. \end{aligned}$$

It follows from Lemma 6.2.6, that

$$H(n) \times W(n) \geq \frac{1}{d}\, n\, (\log(n/m))^{d-1} \quad \text{for} \quad n \geq \lfloor m \rfloor + 1,$$

and hence

$$W(n) \geq \frac{2}{d!}\, n\, (\log n)^{d-2} \log(n/m) \quad \text{for} \quad n \geq \lfloor m \rfloor + 1.$$

□

Note that the bound in Lemma 6.2.7 is tight (except for constant factors): Equality is obtained if all binary trees involved are balanced. Now apply this lemma, with $m = 1$. Then we get the following corollary.

Corollary 6.2.2 *A d-dimensional range tree ($d \geq 2$), representing n points, has size at least $\frac{2}{d!} n (\log n)^{d-1}$.*

Lemma 6.2.8 *Consider a d-dimensional range tree ($d \geq 2$), representing at least n points. Let V be a subset of these points, of cardinality n. Let $m \geq 1$ be a real number, such that $n \geq \lfloor m \rfloor + 1$. Then the total number of nodes in the range tree (in the main tree, or in an associated structure, or in an associated structure of an associated structure, etc.), that represent at least m points of V, is at least*

$$\frac{2}{d!} \frac{n}{m} \left(\log(n/m)\right)^{d-1}.$$

Proof. For $d = 2$, the claim follows from Lemma 6.2.4. Let $d \geq 3$, and suppose the lemma is proved for smaller values of d. Let $W(n)$ be the total number of nodes in the range tree, that represent at least m points of V. If $1 \leq n \leq \lfloor m \rfloor$, then $W(n) \geq 0$. Let $n \geq \lfloor m \rfloor + 1$. Let v be a node in the main tree, that represents the entire set V, such that the left son of v represents n_1 points of V, where $1 \leq n_1 \leq n - 1$. (v need not be the root of the main tree, since it is possible that the left or right son of the root represents the entire set V.) By the induction hypothesis, the associated structure of v contains at least $\frac{2}{(d-1)!}(n/m) \left(\log(n/m)\right)^{d-2}$ nodes, that represent at least m points of V. Hence for $n \geq \lfloor m \rfloor + 1$,

$$W(n) \geq \frac{2}{(d-1)!} \frac{n}{m} \left(\log(n/m)\right)^{d-2} + \min_{1 \leq n_1 \leq n-1} [W(n_1) + W(n - n_1)].$$

Then it follows from Lemma 6.2.6, that

$$W(n) \geq \frac{2}{d!} \frac{n}{m} \left(\log(n/m)\right)^{d-1}, \quad \text{for} \quad n \geq \lfloor m \rfloor + 1.$$

□

By taking all binary trees balanced, we see that the bound of this lemma is tight, except for constant factors.

Lemma 6.2.9 *Let $n \geq 1$ be an integer, and let $\beta \geq 4$ be a real number. Let $a_1 = 0$ and $a_{i+1} = a_i + \beta\, i\, 2^{a_i}$ for $i \geq 1$. Let $k = \min\{i \geq 1 | a_{i+1} > \log n\}$, and let $\alpha = \beta(1 + 2^{1+\beta})$. Then $k \geq \frac{2}{3}\log^* n + \frac{1}{3} - \frac{2}{3}\log^* \alpha$.*

Proof. We first prove that

$$a_i \geq \log(1 + \beta i) \text{ for } i \geq 2. \tag{6.2}$$

For $i = 2$, we have $a_2 = \beta \geq \log(1 + 2\beta)$. So suppose that $a_i \geq \log(1+\beta i)$ for some $i \geq 2$. Then $a_{i+1} = a_i + \beta i 2^{a_i} \geq \beta i 2^{a_i} \geq \beta i(1+\beta i) \geq \beta(i+1) \geq \log(1 + \beta(i+1))$. This proves (6.2).

Now let $i \geq 2$. Then $a_{i+1} = a_i + \beta i 2^{a_i} \leq 2^{a_i} + \beta i 2^{a_i} = (1 + \beta i)2^{a_i}$. It follows from (6.2) that

$$\log a_{i+1} \leq 2a_i \text{ for } i \geq 2. \tag{6.3}$$

We have to prove that $k \geq \frac{2}{3}\log^* n + \frac{1}{3} - \frac{2}{3}\log^* \alpha$. Assume that $k < \frac{2}{3}\log^* n + \frac{1}{3} - \frac{2}{3}\log^* \alpha$. We show that

$$(\log)^{(1+3i)} n \leq a_{k+1-2i} \text{ for } i = 0, 1, \ldots, \lfloor (k-1)/2 \rfloor. \tag{6.4}$$

(Note that $(\log)^{(1+3i)} n$ exists for $0 \leq i \leq \lfloor (k-1)/2 \rfloor$, since by our assumption on k, it holds that $1 + 3i \leq 1 + 3\lfloor (k-1)/2 \rfloor \leq \log^* n$.)

By definition of k, (6.4) holds for $i = 0$. So let $0 \leq i \leq \lfloor (k-3)/2 \rfloor$, and suppose that $(\log)^{(1+3i)} n \leq a_{k+1-2i}$. Then by applying (6.3), we get

$$(\log)^{(2+3i)} n \leq \log a_{k+1-2i} \leq 2\, a_{k-2i}.$$

Hence

$$\begin{aligned} (\log)^{(3+3i)} n &\leq 1 + \log a_{k-2i} \qquad \{\text{apply (6.3)}\} \\ &\leq 1 + 2\, a_{k-2i-1} \\ &\leq 3\, a_{k-2i-1}. \end{aligned}$$

Therefore

$$(\log)^{(1+3(i+1))} n \leq \log 3 + \log a_{k-2i-1} \leq a_{k-2i-1},$$

since $a_{k-2i-1} \geq a_2 = \beta \geq 4$. This proves (6.4).

Now take $i = \lfloor (k-1)/2 \rfloor$ in (6.4). Then

$$(\log)^{(1+3\lfloor (k-1)/2 \rfloor)} n \leq a_3 = \beta + 2\beta 2^{\beta} = \alpha.$$

Taking $\log^* \alpha$ times logarithms, we get

$$(\log)^{(\log^* \alpha + 1 + 3\lfloor (k-1)/2 \rfloor)} n \leq (\log)^{(\log^* \alpha)} \alpha \leq 1.$$

(Note that $(\log)^{(\log^* \alpha + 1 + 3\lfloor (k-1)/2 \rfloor)} n$ exists, since $\log^* \alpha + 1 + 3\lfloor (k-1)/2 \rfloor \leq \log^* n$ by our assumption on k.) However, since we assumed that $k < \frac{2}{3}\log^* n + \frac{1}{3} - \frac{2}{3}\log^* \alpha$, we have $\log^* \alpha + 1 + 3\lfloor (k-1)/2 \rfloor < \log^* n$, and hence

$$(\log)^{(\log^* \alpha + 1 + 3\lfloor (k-1)/2 \rfloor)} n > 1.$$

So we have a contradiction. □

6.3 Lower bounds for restricted partitions

In this section we prove two tight lower bounds for restricted partitions of range trees. Recall that in a restricted partition, a node of the main tree and its associated structure are contained in the same part. It follows from Corollary 6.2.2, that in such a partition of a d-dimensional range tree, there is a part of size at least $\frac{2}{(d-1)!} n (\log n)^{d-2}$.

Theorem 6.3.1 *Consider a d-dimensional range tree ($d \geq 2$), representing n points, where $(d-2)\log\log n \leq \frac{1}{2}\log n$. Let c be a constant, $c \geq \frac{2}{(d-1)!}$. Suppose the range tree is partitioned—in the restricted sense—into parts of size at most $c\, n\, (\log n)^{d-2}$. Then there is an update that passes through at least $\frac{2}{3}\log^* n + \frac{1}{3} - \frac{2}{3}\log^* \alpha$ parts, where $\alpha = c\, d!(1 + 2^{1+cd!})$.*

Proof. Let T be the main tree. Let $a_1 = 0$ and $a_{i+1} = a_i + c\, d!\, i\, 2^{a_i}$ for $i \geq 1$. Let $k = \min\{i \geq 1 | a_{i+1} > \log n\}$. We construct a sequence $v_1, v_2, \ldots, v_k$ of nodes in T, as follows. (For each such node v_i, let Π_i be the part of the partition that contains v_i.) Let v_1 be the root of T. Then v_1 represents at least $n/2^{a_1}$ points. Now let $1 \leq i < k$, and suppose $v_1, \ldots, v_i$ are chosen, in i different parts, such that v_i represents at least $n/2^{a_i}$ points. Consider all nodes in the subtree of T with root

v_i, that represent at least $n/2^{a_{i+1}}$ points. These nodes, together with their associated structures, have size at least $\sum_{v:v \text{ represents } \geq m \text{ points}}(1 + wt(v))$. Here $wt(v)$ is the total number of leaves in the associated structure of node v, $m = n/2^{a_{i+1}}$, and the summation runs over all nodes v in the subtree of T with root v_i, that represent at least m points. By Lemma 6.2.7, this sum is

$$\begin{aligned}
&> \frac{2}{d!}\left\lceil\frac{n}{2^{a_i}}\right\rceil\left(\log\left\lceil\frac{n}{2^{a_i}}\right\rceil\right)^{d-2}\log\frac{\lceil n/2^{a_i}\rceil}{n/2^{a_{i+1}}}\\
&\geq \frac{2}{d!}\,\frac{n}{2^{a_i}}\left(\log\frac{n}{2^{a_i}}\right)^{d-2} c\,d!\,i\,2^{a_i}\\
&= 2\,c\,i\,n\left(\log\frac{n}{2^{a_i}}\right)^{d-2} \quad \{\text{apply } \log n \geq a_{i+1} \geq 2^{a_i}\}\\
&\geq 2\,c\,i\,n\left(\log\frac{n}{\log n}\right)^{d-2}\\
&= 2\,c\,i\,n\,(\log n - \log\log n)^{d-2}\\
&= 2\,c\,i\,n\,(\log\log n)^{d-2}\left(\frac{\log n}{\log\log n} - 1\right)^{d-2} \quad \{\text{apply Lemma 6.2.5}\}\\
&\geq 2\,c\,i\,n\,(\log\log n)^{d-2}\left[\left(\frac{\log n}{\log\log n}\right)^{d-2} - (d-2)\left(\frac{\log n}{\log\log n}\right)^{d-3}\right]\\
&\geq 2\,c\,i\,n\,(\log\log n)^{d-2}\times\frac{1}{2}\left(\frac{\log n}{\log\log n}\right)^{d-2}\\
&= c\,i\,n\,(\log n)^{d-2}.
\end{aligned}$$

Here the last inequality follows from the assumption that $(d-2)\log\log n \leq \frac{1}{2}\log n$. (Note that since $1 \leq i < k$, we have $m = n/2^{a_{i+1}} \geq 1$, and

$$\lfloor m\rfloor + 1 \leq m + 1 = \frac{n}{2^{a_{i+1}}} + 1 \leq \frac{n}{2^{a_2}} + 1 = \frac{n}{2^{c\,d!}} + 1 \leq \frac{n}{16} + 1 \leq n.$$

Hence Lemma 6.2.7 can be applied.)

Now since $|\bigcup_{j=1}^{i}\Pi_j| \leq c\,i\,n\,(\log n)^{d-2}$, it follows that there is a node v_{i+1} in the subtree of T with root v_i, that represents at least $n/2^{a_{i+1}}$ points, and that is not contained in $\bigcup_{j=1}^{i}\Pi_j$.

This procedure gives us nodes $v_1, \ldots, v_k$ in k different parts, such that v_{i+1} is in the subtree of v_i, for $i = 1, 2, \ldots, k-1$. An update in

the range tree, that passes through node v_k, passes through at least k parts of the partition. It follows from Lemma 6.2.9, that $k \geq \frac{2}{3}\log^* n + \frac{1}{3} - \frac{2}{3}\log^* \alpha$. □

Remark. In Theorem 5.1.3, it is shown, that there exists an efficiently maintainable class of d-dimensional range trees, that can be partitioned—in the restricted sense—into parts of size $O(n(\log n)^{d-2})$, such that each update passes, amortized, through at most $\log^* n + O(1)$ parts. Since the lower bound of Theorem 6.3.1 is valid for any individual range tree, not just for some range tree, this theorem also gives a lower bound on the amortized number of parts through which an update passes. Therefore, the lower bound is tight, except for constant factors.

In the next theorem we consider the opposite point of view: We give a lower bound on the maximal size of any part, if each update passes through at most $h(n)$ parts, for some integer function $h(n)$.

We saw already that in a restricted partition of a d-dimensional range tree, there is a part—the part containing the associated structure of the root of the main tree—of size $\Omega(n(\log n)^{d-2})$. Since there exists a class of efficiently maintainable d-dimensional range trees that can be partitioned—in the restricted sense—into parts of size $O(n(\log n)^{d-2})$, such that each update passes, amortized, through at most $\log^* n + O(1)$ parts, it is sufficient to consider restricted partitions where updates visit at most $h(n)$ parts, for some function $h(n) \leq \log^* n$.

Theorem 6.3.2 *Let $h(n)$ be an integer function, such that $1 \leq h(n) \leq \log^* n$. Let T be a d-dimensional range tree ($d \geq 2$), representing n points, and suppose that $(\log)^{h(n)} n \geq 8$ and $(d-1)\log\log n \leq \frac{1}{2}\log n$. Suppose T is partitioned, in the restricted sense, into parts such that each update passes through at most $h(n)$ parts. Then there is a part of size at least*

$$\frac{1}{2}\frac{1}{d!}\, n\, (\log n)^{d-2}\, (\log)^{h(n)} n.$$

Proof. We write $k = h(n)$. Let $m_i = (i+1)\, n\, (\log)^k n / (\log)^{k-i} n$ for $0 \leq i \leq k$. Note that the iterated logarithms exist since $1 \leq k \leq \log^* n$. Then

$$m_k = (k+1)\, (\log)^k n \geq 16,$$

and for $0 \leq i \leq k-1$,

$$\frac{m_i}{m_{i+1}} = \frac{i+1}{i+2}\,\frac{(\log)^{k-i-1}n}{(\log)^{k-i}n} \geq \frac{1}{2}\,\frac{(\log)^{k-i-1}n}{(\log)^{k-i}n} \geq 2,$$

since for $N = (\log)^{k-i}n$, we have $2^N/N \geq 4$. (Note that $N \geq (\log)^k n \geq 8$.) It follows that all m_i's are at least 1, and that $m_i \geq 2m_{i+1} \geq m_{i+1}+1 \geq \lfloor m_{i+1} \rfloor + 1$.

Let P_i be the following property:

> v_i is a node in the main tree of T, that represents at least m_i points. $\Pi_0, \Pi_1, \ldots, \Pi_i$ is a sequence of $i+1$ different parts of the partition. Each update in T, that passes through v_i, passes through $\Pi_0, \Pi_1, \ldots, \Pi_i$.

Now execute the following algorithm:

> v_0 := the root of the main tree;
> Π_0 := the part of the partition that contains v_0;
> $i := 0$;
> QED := **false**;
> { property P_i holds }
> **while** $i \neq k \wedge$ **not** QED
> **do** { property P_i holds }
> Let V be the set of all nodes in the main tree below v_i, that represent at least m_{i+1} points;
> **if** $V \subset \bigcup_{j=0}^{i} \Pi_j$
> **then** QED := **true**
> **else** v_{i+1} := a node in $V \setminus \bigcup_{j=0}^{i} \Pi_j$;
> Π_{i+1} := the part of the partition that contains v_{i+1};
> { property P_{i+1} holds }
> $i := i+1$
> { property P_i holds }
> **fi**
> **od**.

It is not difficult to see that the algorithm correctly maintains property P_i.

Suppose that after the algorithm is completed, QED has the value **false**. Then we must have $i = k$. Also property P_k holds. So we have a node v_k in the main tree, that represents at least m_k points, and we have a sequence $\Pi_0, \Pi_1, \ldots, \Pi_k$ of $k+1$ different parts of the partition, such that each update through node v_k passes through these $k+1$ parts. But this is a contradiction, because each update passes through at most k parts.

Therefore, after the algorithm is completed, QED has the value **true**. Hence there is an i, $0 \leq i \leq k-1$, such that P_i holds, and all nodes in the main tree below v_i that represent at least m_{i+1} points, are contained in $\bigcup_{j=0}^{i} \Pi_j$. By Lemma 6.2.7, all these nodes, together with their associated structures—and hence $\bigcup_{j=0}^{i} \Pi_j$—have size at least

$$\frac{2}{d!}\, m_i\, (\log m_i)^{d-2}\, \log(m_i/m_{i+1}).$$

(Since $m_{i+1} \geq 1$ and $m_i \geq \lfloor m_{i+1} \rfloor + 1$, Lemma 6.2.7 may be applied. Note that we apply Lemma 6.2.7 to the tree having v_i as its root, which is a d-dimensional range tree, representing at least m_i points.) It follows that there is a part in our partition of size at least

$$\frac{1}{i+1}\, |\bigcup_{j=0}^{i} \Pi_j| \geq \frac{1}{i+1}\, \frac{2}{d!}\, m_i\, (\log m_i)^{d-2}\, \log(m_i/m_{i+1}).$$

It remains to prove that this latter expression is at least

$$\frac{1}{2}\, \frac{1}{d!}\, n\, (\log n)^{d-2}\, (\log)^k n.$$

We have

$$\begin{aligned}
\log m_i &= \log\left((i+1)\, n\, \frac{(\log)^k n}{(\log)^{k-i} n}\right) \\
&\geq \log\left(\frac{n}{(\log)^{k-i} n}\right) \\
&\geq \log\left(\frac{n}{\log n}\right) \\
&= \log n - \log\log n.
\end{aligned}$$

It follows that

$$\begin{aligned}(\log m_i)^{d-2} &\geq (\log n - \log\log n)^{d-2} \\ &\geq (\log n)^{d-2} - (d-2)(\log n)^{d-3}\log\log n \\ &\geq \frac{1}{2}(\log n)^{d-2},\end{aligned}$$

since we assumed that $\frac{1}{2}\log n \geq (d-1)\log\log n \geq (d-2)\log\log n$. Furthermore,

$$\begin{aligned}\log(m_i/m_{i+1}) &= \log\left(\frac{i+1}{i+2}\,\frac{(\log)^{k-i-1}n}{(\log)^{k-i}n}\right) \\ &\geq \log\left(\frac{1}{2}\,\frac{(\log)^{k-i-1}n}{(\log)^{k-i}n}\right) \\ &= (\log)^{k-i}n - (\log)^{k-i+1}n - 1 \\ &\geq \frac{1}{2}(\log)^{k-i}n,\end{aligned}$$

since for $N = (\log)^{k-i}n \geq (\log)^k n \geq 8$, we have $N - \log N - 1 \geq N/2$.

Hence

$$\begin{aligned}&\frac{1}{i+1}\,\frac{2}{d!}\,m_i\,(\log m_i)^{d-2}\,\log(m_i/m_{i+1}) \\ &\quad\geq \frac{1}{i+1}\,\frac{2}{d!}\,m_i\,\frac{1}{2}\,(\log n)^{d-2}\,\frac{1}{2}(\log)^{k-i}n \\ &\quad= \frac{1}{2}\,\frac{1}{d!}\,n\,(\log n)^{d-2}\,(\log)^k n.\end{aligned}$$

This proves the theorem. □

Remark. We saw in Theorem 5.1.5, that there exists a class of efficiently maintainable d-dimensional range trees, that can be partitioned—in the restricted sense—into parts of size $O(n(\log n)^{d-2}(\log)^k n)$, such that each update passes, amortized, through at most $k + o(1)$ parts. If the amortized number of parts through which an update passes is at most k, there must be a single update that passes through at most k parts. Therefore, Theorem 6.3.2 also applies for amortized bounds. Hence, the lower bound in Theorem 6.3.2 is tight for constant functions $h(n)$.

6.4 Lower bounds for general partitions

We give two lower bounds for general partitions of range trees. The first bound is proved is by induction, whereas the second bound is proved in a similar way as the lower bound in Theorem 6.1.1. The second bound gives a better result. We include the first bound, however, to illustrate the proof-technique.

In order to be able to give an inductive proof, we prove a more general result.

Theorem 6.4.1 *Let k be a positive integer. Consider a d-dimensional range tree, that is partitioned into parts, such that the following holds. There is a subset V of the points that are represented by the range tree, where $|V| = n \geq 2^k$. Each update that visits a leaf that contains a point of V, passes through at most k parts of the partition. Then there is a part of size at least*

$$\frac{1}{d!}\left(\frac{1}{2}\right)^{k-2}\left(\frac{1}{k}\right)^{d-1} n^{1/k}(\log n)^{d-1}.$$

Proof. Suppose $k = 1$. Then all nodes that represent at least one point of V, are contained in the same part of the partition. By Lemma 6.2.8, with $m = 1$, this part has size at least $\frac{2}{d!}n(\log n)^{d-1}$. Let $k > 1$, and suppose the theorem is proved for $k-1$. Consider a d-dimensional range tree, that satisfies the assumptions of the theorem (for value k). Let Π be the part of the partition that contains the root of the main tree. There are two possible cases.

(i) There is a node y in the range tree, that represents at least $n^{(k-1)/k}$ points of V, that is not contained in part Π. (y may be a node of the main tree, or of an associated structure, or of an associated structure of an associated structure, etc.) Then we merge part Π and the part containing y together into a new part. Let V' be the intersection of V and the set of points that are represented by y. This gives us a d-dimensional range tree, that is partitioned into parts such that the following holds. There is a subset V' of the points that are represented by the range tree, where $|V'| \geq n^{(k-1)/k} \geq 2^{k-1}$. Each update that visits a leaf that contains a point of V', passes through at most $k-1$

parts of this new partition. By the induction hypothesis, there is a part in this new partition, of size at least

$$\frac{1}{d!}\left(\frac{1}{2}\right)^{k-3}\left(\frac{1}{k-1}\right)^{d-1}\left(n^{(k-1)/k}\right)^{1/(k-1)}\left(\log\left(n^{(k-1)/k}\right)\right)^{d-1}$$
$$=\frac{1}{d!}\left(\frac{1}{2}\right)^{k-3}\left(\frac{1}{k}\right)^{d-1}n^{1/k}\left(\log n\right)^{d-1}.$$

It follows that in our original partition, there is a part of size at least

$$\frac{1}{d!}\left(\frac{1}{2}\right)^{k-2}\left(\frac{1}{k}\right)^{d-1}n^{1/k}\left(\log n\right)^{d-1}.$$

(ii) Otherwise, all nodes y in the range tree, that represent at least $n^{(k-1)/k}$ points of V, are contained in part Π. By Lemma 6.2.8, there are at least

$$\frac{2}{d!}\frac{n}{n^{(k-1)/k}}\left(\log\left(\frac{n}{n^{(k-1)/k}}\right)\right)^{d-1}=\frac{2}{d!}\left(\frac{1}{k}\right)^{d-1}n^{1/k}\left(\log n\right)^{d-1}$$

such nodes y. (Note that $n^{(k-1)/k}\geq 1$, and since $n\geq 2^k$, we have $n\geq 2\,n^{(k-1)/k}\geq\lfloor n^{(k-1)/k}\rfloor+1$. Hence Lemma 6.2.8 can be applied.) It follows that part Π has size at least

$$\frac{2}{d!}\left(\frac{1}{k}\right)^{d-1}n^{1/k}\left(\log n\right)^{d-1}\geq\frac{1}{d!}\left(\frac{1}{2}\right)^{k-2}\left(\frac{1}{k}\right)^{d-1}n^{1/k}\left(\log n\right)^{d-1}.$$

This finishes the proof. □

Corollary 6.4.1 *Let $h(n)$ be an integer function, such that $1\leq h(n)\leq\log n$. Consider a d-dimensional range tree ($d\geq 2$), representing n points. Suppose the range tree is partitioned such that each update passes through at most $h(n)$ parts. Then there is a part of size at least*

$$\frac{1}{d!}\left(\frac{1}{2}\right)^{h(n)-2}\left(\frac{1}{h(n)}\right)^{d-1}n^{1/h(n)}\left(\log n\right)^{d-1}.$$

Proof. This follows from Theorem 6.4.1, by taking V the set of all points that are represented by the range tree, and $k=h(n)$. Note that Theorem 6.4.1 remains valid if k depends on n. □

We now prove the other lower bound that improves the bound of Corollary 6.4.1.

Theorem 6.4.2 *Let $h(n)$ be an integer function, such that $1 \leq h(n) \leq \log n$. Let T be a d-dimensional range tree ($d \geq 2$), representing n points. Suppose this range tree is partitioned into parts, such that each update passes through at most $h(n)$ parts. Then there is a part of size at least*

$$\frac{2}{d!} \left(\frac{1}{h(n)} \right)^d n^{1/h(n)} (\log n)^{d-1}.$$

Proof. We write $k = h(n)$. Let $m_i = n^{1-i/k}$ for $0 \leq i \leq k$. Let P_i be the following property:

> V_i is a subset of the set of points represented by T. V_i has cardinality at least m_i. $\Pi_0, \Pi_1, \ldots, \Pi_i$ is a sequence of $i+1$ different parts of the partition. Each update in T, that visits a leaf that contains a point of V_i, passes through $\Pi_0, \Pi_1, \ldots, \Pi_i$.

Now execute the following algorithm:

$i := 0$;
$V_0 :=$ the set of all points that are represented by the range tree;
$\Pi_0 :=$ the part of the partition that contains the root of the main tree;
QED := **false**;
{ property P_i holds }
while $i \neq k \wedge$ **not** QED
do { property P_i holds }
    Let W be the set of all nodes that represent at least m_{i+1} points of V_i;
    if $W \subset \bigcup_{j=0}^{i} \Pi_j$
    then QED := **true**
    else $w :=$ a node in $W \setminus \bigcup_{j=0}^{i} \Pi_j$;
        $V_{i+1} :=$ the set of all points in V_i, that are represented by w;
        $\Pi_{i+1} :=$ the part of the partition that contains w;
        { property P_{i+1} holds }
        $i := i + 1$
        { property P_i holds }
    fi
od.

Note that the algorithm correctly maintains property P_i. In this algorithm, the V_i's are sets of points that are represented by the data structure, whereas the set W is a set of nodes. These are nodes in the main tree, or in an associated structure, or in an associated structure of an associated structure, etc.

Suppose that after the algorithm is completed, QED has the value **false**. Then we must have $i = k$. Also property P_k holds. So we have a set V_k of points of cardinality at least $m_k = 1$, and a sequence $\Pi_0, \Pi_1, \ldots, \Pi_k$ of $k+1$ different parts of the partition, such that each update that visits a leaf that contains a point of V_k, passes through these $k+1$ parts. This is a contradiction, because each update passes through at most k parts of the partition.

Therefore, after the algorithm is completed, QED has the value **true**. Hence there is an i, $0 \leq i \leq k-1$, such that P_i holds, and all nodes that represent at least m_{i+1} points of V_i, are contained in $\bigcup_{j=0}^{i} \Pi_j$. Since $|V_i| \geq m_i$, it follows from Lemma 6.2.8 that there are at least $\frac{2}{d!}(m_i/m_{i+1})\,(\log(m_i/m_{i+1}))^{d-1}$ such nodes. (Note that $m_{i+1} \geq 1$, and that $\lfloor m_{i+1} \rfloor + 1 \leq 2m_{i+1} \leq m_i$, since $k \leq \log n$. Hence Lemma 6.2.8 may be applied.) Hence

$$|\bigcup_{j=0}^{i} \Pi_j| \geq \frac{2}{d!}\,\frac{m_i}{m_{i+1}}\,\left(\log \frac{m_i}{m_{i+1}}\right)^{d-1}.$$

This proves that there is a part of size at least

$$\begin{aligned}
\frac{1}{i+1}\,|\bigcup_{j=0}^{i} \Pi_j| &\geq \frac{1}{i+1}\,\frac{2}{d!}\,\frac{m_i}{m_{i+1}}\,\left(\log \frac{m_i}{m_{i+1}}\right)^{d-1} \\
&= \frac{1}{i+1}\,\frac{2}{d!}\,n^{1/k}\left(\frac{1}{k}\right)^{d-1}(\log n)^{d-1} \\
&\geq \frac{2}{d!}\left(\frac{1}{k}\right)^{d} n^{1/k}\,(\log n)^{d-1}.
\end{aligned}$$

This finishes the proof. □

Consider again the proof of Theorem 6.4.2. We started with a sequence $m_0, \ldots, m_k$ of real numbers such that $m_0 = n$, $m_i \geq 1$ and

$m_i \geq \lfloor m_{i+1} \rfloor + 1$ for all i. Then it was shown that the partition contains a part of size at least

$$\frac{2}{d!} \times \frac{1}{i+1} \frac{m_i}{m_{i+1}} \left(\log \frac{m_i}{m_{i+1}} \right)^{d-1},$$

for some i, $0 \leq i \leq k-1$. Since i can take any value between 0 and $k-1$, there is a part of size at least (we omit the factor $\frac{2}{d!}$)

$$\min \left\{ \frac{m_0}{m_1} \left(\log \frac{m_0}{m_1} \right)^{d-1}, \frac{1}{2} \frac{m_1}{m_2} \left(\log \frac{m_1}{m_2} \right)^{d-1}, \ldots, \frac{1}{k} \frac{m_{k-1}}{m_k} \left(\log \frac{m_{k-1}}{m_k} \right)^{d-1} \right\}. \tag{6.5}$$

Just as in Section 6.1, it might be possible to improve the lower bound in Theorem 6.4.2 by taking another sequence $m_0, \ldots, m_k$. We shall show that in this way the lower bound can only be improved by a constant factor, where this constant depends on d, but not on n and not on k. More precisely, we shall prove that for any sequence $m_0, \ldots, m_k$ of positive real numbers, where $m_0 = n$, $m_i \geq 1$ and $m_i \geq m_{i+1}$ for all i, the value of (6.5) is at most

$$e(1 + \log e)^{d-1} \left(\frac{1}{k} \right)^d n^{1/k} (\log n)^{d-1}.$$

Take such a sequence $m_0, \ldots, m_k$. There are two possibilities.

(i) There is an i, $0 \leq i \leq k-1$, such that $m_i \leq \frac{n}{i!} (\frac{k!}{n})^{i/k}$ and $m_{i+1} > \frac{n}{(i+1)!} (\frac{k!}{n})^{(i+1)/k}$. Then, by using Lemma 6.1.1,

$$\frac{m_i}{m_{i+1}} \leq (i+1) \left(\frac{n}{k!} \right)^{1/k} \leq (i+1) \frac{e}{k} n^{1/k},$$

and hence (note that $\log(m_i/m_{i+1}) \geq 0$, since $m_i \geq m_{i+1}$)

$$\begin{aligned} \frac{1}{i+1} \frac{m_i}{m_{i+1}} \left(\log \frac{m_i}{m_{i+1}} \right)^{d-1} &\leq \frac{e}{k} n^{1/k} \left(\log \left(\frac{i+1}{k} e\, n^{1/k} \right) \right)^{d-1} \\ &\leq \frac{e}{k} n^{1/k} \left(\log \left(e\, n^{1/k} \right) \right)^{d-1}. \end{aligned}$$

Hence the value of (6.5) is at most $\frac{e}{k} n^{1/k} (\log(e\, n^{1/k}))^{d-1}$.

(ii) Otherwise, for each i, $0 \leq i \leq k-1$, we have that if $m_i \leq \frac{n}{i!}(\frac{k!}{n})^{i/k}$, then $m_{i+1} \leq \frac{n}{(i+1)!}(\frac{k!}{n})^{(i+1)/k}$. In the same way as in Section 6.1, it follows that $m_{k-1} \leq \frac{n}{(k-1)!}(\frac{k!}{n})^{(k-1)/k} = k\,(\frac{n}{k!})^{1/k} \leq e\,n^{1/k}$ and that $m_k = 1$. Hence, since $\log m_{k-1} \geq 0$,

$$\frac{1}{k}\,\frac{m_{k-1}}{m_k}\left(\log\frac{m_{k-1}}{m_k}\right)^{d-1} \leq \frac{e}{k}\,n^{1/k}\left(\log\left(e\,n^{1/k}\right)\right)^{d-1}.$$

Again we conclude that the value of (6.5) is at most $\frac{e}{k}\,n^{1/k}(\log(e\,n^{1/k}))^{d-1}$.

Now since

$$\begin{aligned}\left(\log\left(e\,n^{1/k}\right)\right)^{d-1} &= \left(\log n^{1/k} + \log e\right)^{d-1} \\ &= \left(\log n^{1/k}\right)^{d-1}\left(1 + \frac{\log e}{\log n^{1/k}}\right)^{d-1} \\ &\leq \left(\log n^{1/k}\right)^{d-1}(1+\log e)^{d-1},\end{aligned}$$

—where the inequality follows from the fact that $\log n^{1/k} \geq 1$ or, equivalently, $k \leq \log n$—it follows that the value of (6.5) is at most

$$\frac{e}{k}\,n^{1/k}\left(\log\left(e\,n^{1/k}\right)\right)^{d-1} \leq e(1+\log e)^{d-1}\left(\frac{1}{k}\right)^{d} n^{1/k}(\log n)^{d-1},$$

which proves our claim.

Chapter 7

Summary and concluding remarks

We have studied the problem of partitioning range trees, such that queries and updates pass through only a small number of parts. This enables us to store range trees in secondary memory and to query and maintain them efficiently. This is useful in large scale applications, where the data structure is too large to be stored in main memory.

Because the reader might be overwhelmed by the many theorems of this part of the thesis, we give in this chapter a summary of the most important results.

Recall that a balanced range tree storing a set of n points in d-dimensional space, has size $\Theta(n(\log n)^{d-1})$. A partition of such a range tree 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 size of 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 $h(n)$.

We have considered two types of partitions. The first type are the restricted partitions. These have been studied in Sections 4.1, 5.1 and

6.3. For such partitions, we have proved lower bounds that match with the best upper bounds. The best results are:

1. An $(O(n(\log n)^{d-2}), 4\log^* n + O(1), \log^* n + O(1))$-partition, see Theorems 4.1.4 and 5.1.3. This partition is optimal, see Theorem 6.3.1.

2. An $(O(n(\log n)^{d-2}(\log)^k n), 2k-1, k+o(1))$-partition, see Theorems 4.1.6 and 5.1.5. Here, k is a fixed parameter. This partition is optimal, see Theorem 6.3.2.

General partitions have been studied in Sections 4.3, 4.4, 5.3 and 6.4. In Theorem 5.3.2, the most general result is given, which is a partition into parts of size

$$\Theta\left(n^{d/(k+d-1)} \times (\log n)^{d(d-1)/(k+d-1)}\right),$$

such that the amortized number of parts through which an update passes is at most

$$\left(\begin{array}{c} k+d-1 \\ d \end{array}\right) + o(1) = \frac{1}{d!}k^d + O\left(k^{d-1}\right),$$

and the number of parts through which a query passes is at most

$$\sum_{i=0}^{d} \left(\begin{array}{c} d+k-1-i \\ k-1 \end{array}\right)\left(\begin{array}{c} k-1 \\ i \end{array}\right) + 2t - 1 = \frac{2^d}{d!}k^d + 2t + O\left(k^{d-1}\right),$$

where t is the number of answers to the query. Here, k is a fixed parameter. The asymptotic terms $O(k^{d-1})$ are valid for fixed d and $k \to \infty$.

The best lower bound for general partitions is given in Theorem 6.4.2, which states that if we partition a range tree into parts, such that each update passes through at most $h(n)$ parts, there must be a part of size at least

$$\frac{2}{d!}\left(\frac{1}{h(n)}\right)^d n^{1/h(n)} (\log n)^{d-1}.$$

In Sections 4.2 and 5.2 we have shown, that it is useful to change range trees, to get new data structures for the range searching problem,

for which more efficient (restricted) partitions exist. These new structures have the same performances as ordinary balanced range trees. In the two-dimensional case, this leads to a data structure—of size $O(n \log n)$—for which an $(O(n), 3, 2 + o(1))$-partition exists. See Theorem 4.2.2. The general result is given in Theorem 5.2.2: A partition of a d-dimensional structure of size $O(n(\log n)^{d-1})$, into parts of size $O(n)$, such that an update passes, amortized, through at most

$$\left\lfloor \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^d + \frac{3}{2} \right\rfloor + o(1), (n \to \infty)$$

parts, and a query passes through at most

$$1 + 2^d (\log n)^{\lceil (d-1)/2 \rceil}$$

parts. (This latter bound is very pessimistic. See the remark after Theorem 4.2.2.)

Note that the lower bounds do not apply for these new structures, since they do not have the form of a range tree. (Several of the associated structures are omitted.)

In all partitions, we have used asymptotic estimates to express the size of the parts, whereas in the lower bounds we have also given the constant factors. Of course, it is possible to compute the constant factors in the estimates of the size of the parts. The details, however, become very tedious. Furthermore, such computations will not give additional insight in the nature of the partition schemes.

Note that in most lower bounds, the constant factor $1/d!$ appears. Therefore, one might think that for large values of d, these bounds are not very useful. Monier [34], however, has shown that the constant factors that occur in the complexity of algorithms that use Bentley's multi-dimensional divide-and-conquer technique—and range trees indeed use this technique—are proportional to $1/(d-1)!$.

We finish this chapter with some open problems and directions for future research.

For the general partitions, the upper and lower bounds are still reasonably far apart. In the two-dimensional case, the best result is

a partition into parts of size $\Theta((n \log n)^{2/(k+1)})$, such that an update passes, amortized, through at most $k(k+1)/2 + o(1)$ parts, and a query passes through at most $2k^2 - 2k + 2t$ parts, if t is the number of answers. (See Theorem 4.4.2.) Compare this to the best lower bound—see Theorem 6.4.2—which says that if a two-dimensional range tree is partitioned such that each update passes through at most $k(k+1)/2$ parts, there must be a part of size $\Omega(n^{2/(k(k+1))} \times \log n)$. It would be interesting to close the gap between these bounds. Another interesting problem is to decrease in the just mentioned partition the term $2t$ in the number of visited parts for a query. In this partition, all reported answers can be situated in different parts. Maybe it is possible to distribute the points over parts, of size say $f(n)$, such that only $O(t/f(n))$ parts are needed to report t answers. In fact, Corollary 4.3.1 shows that for $k = 2$, this is indeed possible.

We mentioned already that the techniques for restricted partitions of Sections 4.1 and 5.1, also apply to many more data structures having the form of an augmented binary tree, with some reasonable properties of the query and update algorithms. Examples are segment trees (see [47]), structures solving order decomposable set problems (see Section 2.5), and structures for adding range restrictions to searching problems (see [5, 66]). An interesting direction for research is to identify the basic properties such structures should have, in order that the techniques apply. In this way, it might be possible to design partitions for a (maybe very general) class of data structures.

A more general problem is to design partition techniques for other data structures, or for special classes of data structures. For many data structures that are based on tree structures, the techniques presented here will be applicable. So data structures that are not based on trees seem especially interesting.

We saw one application of the partitioning problem: Solutions can be applied for maintaining the structure in secondary memory in case it is too large to be stored in main memory. The partitions as we considered them can also be applied to the reconstruction problem of Part III. There, we can maintain a copy of the data structure—which itself is stored in main memory—in secondary memory. In this way we can reconstruct the structure in case the information in main memory is destroyed. Another application is in the area of parallel algorithms.

Here we partition a data structure into parts, and we distribute the parts among a number of processors. Clearly, this leads again to the partitioning problem. In this application, however, the partition should satisfy other constraints, since we want the amount of parallelism to be as large as possible. So one could study this type of partitioning problems for different kinds of data structures. In [24], the idea of using a partitioned range tree on a parallel computer is already applied. Finally, one could search for other applications of partitioned data structures.

Bibliographic comments

The partitioning problem as studied here was posed to the author in 1986 by Mark Overmars. Chapters 3-5 are based on joint work with Mark Overmars, Mark de Berg and Marc van Kreveld, see [44, 45]. Sections 4.4 and 5.3 are more polished versions of the corresponding sections in [45]. Chapter 6 is based on work with Mark Overmars, see [44, 50, 53]. The proofs of Theorems 6.1.1, 6.3.2 and 6.4.2 have been simplified by Peter van Emde Boas.

Part III

The reconstruction problem for dynamic data structures

Chapter 8

The reconstruction problem

8.1 Introduction

In this part we study the reconstruction problem for dynamic data structures, which is a special instance of the general problem of maintaining multiple representations of data structures. In the reconstruction problem, we have to design for a given searching problem, a dynamic data structure solving this searching problem, together with a *shadow administration* from which the data structure can be reconstructed in case of calamity. This shadow administration is stored in secondary memory, whereas the data structure itself is stored in main memory.

In this way, we have a multiple representation of the data. There is one data structure—stored in main memory—that stores the data, and on which queries and updates are performed. In secondary memory the data is represented by a shadow administration on which only updates are performed. We study how to organize this shadow administration for several types of searching problems, what type of information has to be stored in the shadow administration such that the data structure can be reconstructed fast, and how to update it efficiently.

Clearly, we can solve the reconstruction problem by maintaining in secondary memory a copy of the data structure. Therefore, the techniques of Part II can be used here. We will see, however, that there are much more efficient techniques to solve the reconstruction

problem. In fact, we have added more degrees of freedom, compared to the problem in Part II: The shadow administration should be a structure from which the original data structure can be reconstructed; it does not have to be an exact copy of this structure. Moreover, no queries have to be performed on it. Therefore, we can design a shadow administration that can be maintained—in secondary memory—more efficiently than the copy of the data structure. In this way, we get for example a very efficiently maintainable shadow administration for a two-dimensional range tree. (See Theorem 9.3.3.)

The reconstruction problem first appeared in a paper by Torenvliet and van Emde Boas [61]. In this paper, the reconstruction and optimization of trie hashing functions are investigated. No other papers concerning the reconstruction problem have appeared.

In the next section, we introduce a general framework in which we describe solutions to the reconstruction problem. We use the Random Access Machine as our model of main memory. For secondary memory, we take the Indexed Sequential Model, as described in Section 2.6. In Section 8.3, we give some basic solutions to the reconstruction problem.

In Chapter 9, we give some general techniques that apply to large classes of searching problems.

In Chapter 10, we consider a particular searching problem: The union-find problem. We design an efficient main memory data structure, that has a worst-case single operation complexity that is lower than the best previously known complexity. This new structure is designed in such a way that a copy of it can efficiently be maintained in secondary memory.

In Chapter 11 we apply the ideas of deferred data structuring—due to Karp, Motwani and Raghavan [28, 35]—to the reconstruction problem. We first show that static deferred data structures can often be dynamized using well-known techniques. Then we use the dynamic deferred structures to get a new approach for solving the reconstruction problem.

Note that Chapters 10 and 11 contain results that are also interesting in other areas besides the reconstruction problem.

In Chapter 12, we give a summary of the most important results.

8.2 The general framework

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

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

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

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

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

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

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

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

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

In most cases, the copy *CSH* of the shadow administration is stored in secondary memory in consecutive blocks, always starting at the same block. This block is called block 0. We assume that the system knows the address of block 0; it is not destroyed in a system crash. Then, the number of disk accesses in the reconstruction procedure is equal to one. (In Section 9.3, we have a situation where *CSH* does not always start at the same block, and where the number of disk accesses for reconstruction is greater than 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.

We have introduced a *multiple representation* of the data: The set of objects for which we want to solve the given searching problem is represented by several cooperating structures, each having its own task. The data structure *DS* has to maintain the set of objects, in order that queries can be performed on it. Hence, on *DS*, all operations (i.e., queries, insertions and deletions) are performed. On the structures *SH*, *CSH* and *INF*, only insertions and deletions are carried out. The structure *INF* is used to update *SH* and *CSH*. The structure *SH* is used to update *CSH*. Finally, the task of *CSH* is to maintain information to reconstruct the other structures.

In this part of the thesis, we use the following notations to denote the complexity of the structures. For the data structure *DS*, we use the usual notations $P(n)$, $S(n)$, $Q(n)$, $I(n)$, $D(n)$ and $U(n)$ without subscripts. See Section 2.1. The complexity of the additional structures *SH*, *CSH* and *INF* is denoted by:

- $S'(n)$: the total amount of space required by the additional struc-

tures.

- $P_s(n)$, $P_t(n)$ and $P_c(n)$: the number of disk accesses (seeks), the transport time and the computing time, respectively, needed to build the additional structures.
- $I_s(n)$, $I_t(n)$ and $I_c(n)$: the number of disk accesses, the transport time and the computing time, respectively, needed to insert an object into the additional structures.
- $D_s(n)$, $D_t(n)$ and $D_c(n)$: the number of disk accesses, the transport time and the computing time, respectively, needed to delete an object from the additional structures.
- If the insert and delete complexity measures are equal, we denote the common update complexity by $U_s(n)$, $U_t(n)$ and $U_c(n)$.
- $R_s(n)$, $R_t(n)$ and $R_c(n)$: the number of disk accesses, the transport time and the computing time, respectively, needed to reconstruct the structures *DS*, *SH* and *INF* from the structure *CSH* that is stored in secondary memory.

Note that $P_t(n) = \Theta(S_{CSH}(n))$, $R_t(n) = \Theta(S_{CSH}(n))$ $P_c(n) = \Omega(S_{CSH}(n))$ $R_c(n) = \Omega(S_{CSH}(n))$, $I_c(n) = \Omega(I_t(n))$ and $D_c(n) = \Omega(D_t(n))$. If the copy *CSH* is stored in consecutive blocks, then $P_s(n) = 1$.

We assume that all these complexity measures are smooth and non-decreasing. We also assume that $S'(n)/n$, $P_c(n)/n$ and $R_c(n)/n$ are non-decreasing.

8.3 Some basic solutions

8.3.1 A low storage shadow administration

Let *DS* be a dynamic data structure, representing a set V of n objects. We assume that the set V is a subset from some ordered universe. Clearly, if we keep in secondary memory the objects of the set V, we have enough information to reconstruct the data structure *DS*.

Let $V = \{p_1 < p_2 < \ldots < p_n\}$ be the ordered set of objects. Divide secondary memory in blocks, such that each block can contain b objects.

We partition V into subsets $V_1 = \{p_1, \ldots, p_{b/2}\}$, $V_2 = \{p_{b/2+1}, \ldots, p_b\}$, etc.

The shadow administration: The structure *SH* consists of a linked list, containing the objects of V in sorted order. Each node in this list contains a pointer to its successor. The structure *CSH* is a copy of *SH*. We store *CSH* in secondary memory in consecutive blocks, starting at block 0: Each sublist containing a V_i is stored in one block. Note that a pointer in *CSH* "points" to a successor in *main* memory. So in secondary memory, the pointers in *CSH* have no meaning. With each indivisible piece of information of *CSH*, we store the address of the corresponding piece of *SH*. Finally, we store in main memory at the end of each sublist that contains a V_i, the address in secondary memory of the block containing this sublist, and we maintain in main memory the address of the block at the end of the file.

The structure *INF* is a balanced leaf search tree, containing in its leaves the elements of V in sorted order. Each leaf of this tree—storing, say, object p—contains a pointer to object p in the list *SH*.

The insert algorithm: Suppose object p is to be inserted into the set V. Then we insert p into the tree *INF*. This gives us the position in *SH* where p has to be inserted. Next p is inserted in the list *SH*. Let V_i be the subset of V into which p is inserted. There are two possibilities.

(i) After the insertion, the set V_i contains less than b objects. In this case, we replace the block in secondary memory, containing the old V_i, by a block containing the updated V_i, together with their addresses in main memory. (We know the address of this block, by walking to the end of the sublist that contains V_i.) Also, we add in *INF* the information about the position of p in *SH*. If p is at the end of its sublist, we store in this sublist—in main memory—the address of the block in secondary memory that contains the copy of this sublist. If p is at the beginning of its sublist, we replace in secondary memory the predecessor block of V_i—the address of which we find by searching in *INF* the predecessor q of p, and by following the pointer to q in the list *SH*—by a block storing the same information, except that the last object of the sublist contains a pointer—which is an address in main memory—to p.

(ii) After the insertion, V_i contains b objects. Then we split V_i in two subsets V_{i1} and V_{i2}, both of cardinality $b/2$, such that the objects in V_{i1} are less than those in V_{i2}. We store the part of the list containing V_{i1} in the block containing the old V_i. The part of the list containing V_{i2} is stored in a new block at the end of the file. (We know the address of the end of the file.) Then, the addresses in secondary memory of the blocks containing V_{i1} and V_{i2} are inserted at the end of the (main memory) sublists V_{i1} and V_{i2}. We also maintain in main memory the address of the new block at the end of the file. Again if p is the first element of V_{i1}, we replace the predecessor block of V_{i1} by a block storing the same information, except that the last object of the sublist contains a pointer to p.

The delete algorithm: Suppose object p is to be deleted from the set V. Then we delete p from the tree *INF*. This gives us the position in *SH* where p has to be deleted. Next we delete p from the list *SH*. Let i be the index of the subset from which p is deleted. Again there are two possibilities.

(i) After the deletion, V_i contains more than $b/4$ objects. Then we proceed in a similar way as we did in case (i) of the insert algorithm.

(ii) After the deletion, V_i contains $\leq b/4$ elements. First suppose that V_i does not contain the smallest elements of V. Then there is a predecessor sublist V_{i-1}. In this case we merge V_{i-1} and V_i into a new subset V_i, and the old V_{i-1} is discarded. If the resulting V_i contains at least b objects, we split it in two equal sized sets V_{i-1} and V_i. The part of the list containing V_{i-1} resp. V_i is stored in secondary memory in the block containing the old V_{i-1} resp. V_i. (Using the tree *INF*, we can find the address of the block containing the old V_{i-1}.) If the resulting V_i contains less than b elements, we store the list containing V_i in secondary memory in the block containing the old V_i. In order to avoid gaps in secondary memory, the block at the end of the file is moved to the block containing the old V_{i-1}. Of course, information about the new addresses of the moved blocks is inserted in the list *SH*, and we store in main memory the address of the new block that is at the end of the file. Note that if p was the first element of the sublist, its predecessor—say q—was in V_{i-1}. Hence no extra disk access is necessary for giving q a pointer to its new successor. If V_i contains the

smallest elements of V, we merge V_i and V_{i+1} and proceed in a similar way.

The reconstruction algorithm: To reconstruct the structures, we transport *CSH* to main memory. We use the addresses stored in *CSH* to store the information in the same positions as it was in the destroyed list *SH*. Then *CSH* can take over the role of the destroyed *SH*, and we build the data structure *DS* and the tree *INF* from the objects in the ordered list.

Theorem 8.3.1 *Let DS be a data structure that can be built from an ordered set of n objects in $P_o(n)$ time. There exists a shadow administration for DS, with performances:*

1. $S'(n) = O(n)$.
2. $U_s(n) \leq 3$, $U_t(n) = O(b)$ *and* $U_c(n) = O(\log n + b)$, *where b is the number of points that can be stored in one block.*
3. $R_s(n) = 1$, $R_t(n) = O(n)$ *and* $R_c(n) = O(n + P_o(n))$.

Proof. It is clear that the total amount of space required by the additional structures *SH*, *CSH* and *INF* is bounded by $O(n)$.

The insert algorithm takes $O(\log n)$ computing time for the insertion of p into the structures *INF* and *SH*. It takes $O(\log n + b)$ computing time to find the addresses of the blocks in secondary memory that have to be updated. The update of the structure in secondary memory takes at most 3 disk accesses, an amount of $O(b)$ computing time, and at most 3 blocks of data transport. Note that in most cases only one disk access and one block of data transport are necessary. The complexity of a deletion follows in the same way.

The reconstruction algorithm takes one disk access, $O(n)$ transport time and $O(n + P_o(n))$ computing time. Here, the number of disk accesses is equal to one, because the copy *CSH* is stored in consecutive blocks, always starting at block 0. Note that we assume that the system knows the address of block 0; this address will not be destroyed. □

This technique is especially efficient if the preprocessing algorithm of the data structure *DS* consists of two phases. In the first phase we

order the objects in $\Theta(n \log n)$ time. Then, in the second phase, the actual building of the structure is done. For such data structures, we save $\Theta(n \log n)$ computing time in the reconstruction algorithm.

As an example, consider the case where *DS* is a range tree with a slack parameter, as introduced by Mehlhorn [33].

Definition 8.3.1 Let k be a positive integer, and let V be a set of n points in the plane. A *range tree with slack parameter* k, representing the set V, consists of the following. There is a BB[α]-tree T that contains in its leaves the elements of V, ordered according to their x-coordinates. Each node v of T, that has a distance to the root of T which is a multiple of k, contains (a pointer to) a BB[α]-tree, that stores in its leaves the subset of V represented by node v, ordered according to their y-coordinates.

Note that the BB[α]-range tree as introduced in Section 2.3 is in fact a range tree with slack parameter one. The complexity of these more general range trees is given in the following theorem.

Theorem 8.3.2 *A range tree with slack parameter k, representing a set V of n points in the plane, has performances:*

1. $S(n) = O((n \log n)/k)$.
2. $P(n) = O(n \log n) + O((n \log n)/k) = O(n \log n)$. *Here the first term is the time required to order the points of V according to their y-coordinates, whereas the second term is the actual building time of the structure.*
3. *The amortized update time is bounded by $O((\log n)^2)$.*
4. *Orthogonal range queries can be solved in time $O((\log n)^2 2^k/k+t)$, where t is the number of reported answers.*

Proof. The tree T has size $O(n)$. There are $O((\log n)/k)$ levels in T, the nodes of which contain associated structures. For each of these levels, the associated structures on that level together have size $O(n)$. It follows that the entire data structure has size $O((n \log n)/k)$. The proof of the building time follows in a similar way as in Section 2.3. A range

tree with a slack parameter is a BB[α]-range tree where several associated structures are omitted. Therefore, the amortized update time for this type of range trees cannot be larger than that of a normal range tree. Hence, the bound on the update time follows from Section 2.3. The proof of the query time can be found in [33]. □

It follows from this theorem, that by maintaining in secondary memory the points of the set V ordered according to their y-coordinates, we save $O(n \log n)$ computing time in reconstructing the data structure. Therefore, we take for the range tree the shadow administration given above.

Theorem 8.3.3 *For a range tree with slack parameter k, there exists a shadow administration with performances:*

1. $S'(n) = O(n)$.
2. $U_s(n) \leq 3$, $U_t(n) = O(b)$ *and* $U_c(n) = O(\log n + b)$, *where b is the number of points that can be stored in one block.*
3. $R_s(n) = 1$, $R_t(n) = O(n)$ *and* $R_c(n) = O((n \log n)/k)$.

Proof. This follows from Theorems 8.3.2 and 8.3.1. □

Take for example the slack parameter $k = \lceil \log \log n \rceil$. Then the range tree has size $S(n) = O((n \log n)/\log \log n)$, whereas the shadow administration has size only $O(n)$. So the size of the additional structures is asymptotically less than that of the data structure itself. Also, reconstruction of the range tree takes $O((n \log n)/\log \log n)$ computing time, which is asymptotically less than the preprocessing time of the range tree.

8.3.2 Other basic solutions

Let *DS* be a dynamic data structure of size $S(n)$. We can solve the reconstruction problem by maintaining in secondary memory a copy of *DS*. In that case we do not need the structures *SH* and *INF*, and *CSH* is a copy of *DS*. We partition the structure *DS* into a number of parts. Then, *CSH* is stored in secondary memory by putting the copy of each

part of the partition in one block. With each piece of information, we store in secondary memory the address of the corresponding piece in main memory. In main memory, we record for each part of the partition the address of the block in secondary memory containing its copy. After an update of the data structure DS, the copy CSH is updated by replacing all parts of the partition in which some information has changed. Obviously, the complexity of an update heavily depends on the way the structure DS is partitioned.

Reconstruction of the data structure DS takes one disk access and $O(S(n))$ transport and computing time: We only have to transport the structure CSH to main memory and put the information in the correct positions.

This way of solving the reconstruction problem is studied further in Chapter 10, where a copy of a Union-Find data structure is stored in secondary memory. The techniques of Part II, where we investigated the problem of maintaining a range tree in secondary memory, also apply here.

Another basic solution is the following. Let DS be a dynamic data structure of size $S(n)$, having an update time $U(n)$. Let CSH consist of a copy of DS. Again, we do not need the structures SH and INF. We store the copy CSH in secondary memory in a number of consecutive blocks, starting at block 0. With each piece of information, we store in secondary memory the address of the corresponding piece in main memory. Let n be the initial number of objects that are represented by DS.

The update algorithm: An insertion or deletion of an object p is performed in the main memory structure DS. Then we store p, together with information whether it has to be inserted or deleted, in secondary memory in a block of constant size at the end of the file. So the structure CSH itself is not affected.

After $S(n)/U(n)$ updates have been performed in this way, we transport a copy of the up-to-date data structure DS to secondary memory, where it replaces the old information. (The old copy CSH and the sequence of $S(n)/U(n)$ updates are discarded.) Then we proceed in the same way, now with a sequence of $S(m)/U(m)$ updates, where m is the

number of objects at this moment.

The reconstruction algorithm: To reconstruct the data structure DS, we transport the information to main memory, and we store the structure CSH in the correct positions. Then we perform the sequence of at most $S(n)/U(n)$ updates to make the structure up-to-date.

Theorem 8.3.4 *Let DS be a dynamic data structure of size $S(n)$, having an update time $U(n)$. There exists a shadow administration for DS, with performances:*

1. $S'(n) = O(S(n))$.
2. $U_s(n) = 1$, *and the amortized values of* $U_t(n)$ *and* $U_c(n)$ *are both bounded by* $O(U(n))$.
3. $R_s(n) = 1$, $R_t(n) = O(S(n))$ *and* $R_c(n) = O(S(n))$.

Proof. We first prove that $S(n) \leq n \times U(n)$: The data structure DS can be built by performing n insertions into an initially empty structure. In this way we spend an amount of time that is bounded above by $U(1) + U(2) + \cdots + U(n) \leq n \times U(n)$. (This inequality holds because we assume our complexity measures to be non-decreasing.) During these insertions we have built a structure of size $S(n)$, and, hence, we have spent at least $S(n)$ time. Therefore, $S(n) \leq n \times U(n)$.

The given update procedure of the shadow administration takes one disk access, and an amortized transport and computing time that are bounded by

$$O\left(1 + \frac{S(m)}{S(n)/U(n)}\right).$$

Clearly, $m \leq n + S(n)/U(n)$. We saw that $S(n) \leq n \times U(n)$. Therefore, $m \leq 2n$. Since we assume our complexity measures to be smooth, we have $S(m)/S(n) = O(1)$. Hence, the amortized transport and computing time of the shadow administration are both bounded by $O(U(n))$.

The size of the shadow administration is bounded by $O(S(n))$ for the structure CSH, and $O(S(n)/U(n))$ for the sequence of updates. So in total, we need $O(S(n))$ space in secondary memory.

The reconstruction algorithm takes one disk access, $O(S(n))$ transport time and an amount of computing time. To store the information in the correct positions takes $O(S(n))$ computing time. The final updates take an amount of computing time that is bounded by $(S(n)/U(n)) \times U(n')$, where n' is the maximal number of objects that are represented by DS during these updates. In the same way as above, we have $n' \leq 2n$. Since our complexity measures are smooth, the final updates take $O(S(n))$ computing time. Hence the amount of computing time in the complete reconstruction algorithm is bounded by $O(S(n))$. □

In this theorem, the update time bounds are amortized. It is possible to turn these bounds into worst-case bounds. Then, the number of disk accesses for an update increases to 2. We do not prove this here, because in Section 9.3 we give a general worst-case technique that gives an even better result.

Chapter 9

General approaches

9.1 Order decomposable set problems

For the data structures solving order decomposable set problems, as defined in Section 2.5, efficient shadow administrations can be constructed.

Let PR be an $M(n)$-order decomposable set problem, and let V be a set of cardinality n, for which we want to solve PR. As in Section 2.5, we assume that the answer $PR(V)$ takes $O(M(n))$ space to store. (In the examples we consider, this is indeed the case. The assumption is, however, not crucial. See [41, 42].)

Let $f(n)$ be a smooth integer function, such that $1 \leq f(n) \leq n$. Partition the ordered set $V = \{p_1 < p_2 < \ldots < p_n\}$ into subsets $V_1 = \{p_1, \ldots, p_{f(n)}\}$, $V_2 = \{p_{f(n)+1}, \ldots, p_{2f(n)}\}$, etc. Let DS be the dynamic data structure of Section 2.5 that maintains the answer $PR(V)$. Recall that we store each set V_i in a binary search tree T_i. The roots of the T_i's are stored in the leaves of a binary search tree T. Each node v of T stores the answer to PR for the subset of V represented by v.

Consider this structure DS. Clearly, if we have all trees T_i and all answers $PR(V_i)$, we can build the rest of the data structure DS very fast: We only have to merge the answers to obtain the tree T with the partial answers in its nodes. This leads to the following shadow administration.

The shadow administration: The structure CSH consists of the

trees T_i, and the answers $PR(V_i)$ to the set problem PR for the subsets V_i. Since the shadow administration consists only of parts of the data structure DS itself, we do not need the structures SH and INF. We reserve in secondary memory consecutive blocks, starting at block 0, such that each block can contain one answer $PR(V_i)$ and one tree T_i, together with information about their positions in main memory, for a set V_i of cardinality at most $2f(n)$. Then we store in each such block an answer $PR(V_i)$ and the corresponding tree T_i. In the data structure DS itself, we store in each leaf of the tree T—the leaves contain the roots of the trees T_i—the address of the block in secondary memory that contains the corresponding structures T_i and $PR(V_i)$.

The update algorithm: If after an update the data structure is not rebuilt, only one tree T_i and one answer $PR(V_i)$ will have changed and, hence, have to be transported to secondary memory. Note that we know from the update of the data structure, which T_i and which $PR(V_i)$ are changed. Also we know the position in secondary memory where these changed structures have to be written. Otherwise, if the data structure is rebuilt, we just transport the entire new shadow administration to secondary memory.

The reconstruction algorithm: To reconstruct the data structure, we transport the shadow administration to main memory, and we put all information in the correct positions. Then we rebuild the tree T that stores the partial answers in its nodes, by merging the partial answers in a bottom-up fashion: For each node v, we copy the answers stored in its two sons, and we merge them to obtain the answer for v.

This leads to the following theorem. See Section 2.5 for the notations.

Theorem 9.1.1 *Let $f(n)$ be a smooth integer function, $1 \leq f(n) \leq n$. For the data structure DS, solving an $M(n)$-order decomposable set problem, there exists a shadow administration, with performances:*

1. $S'(n) = O(n + (n/f(n)) \times M(f(n)))$.
2. $U_s(n) = 1$ *and the amortized values of* $U_c(n)$ *and* $U_t(n)$ *are both bounded by* $O(f(n)+M(f(n))+n/f(n)+(n/(f(n))^2)\times M(f(n)))$.

3. $R_s(n) = 1$, $R_t(n) = O(n + (n/f(n)) \times M(f(n)))$ *and* $R_c(n) = O(n + M'(n))$.

Proof. The trees T_i together have size $O(n)$. Because of our assumption, the answers $PR(V_i)$ together have size $O((n/f(n)) \times M(f(n)))$. This proves the bound on $S'(n)$.

If the data structure is not rebuilt, only one tree T_i and one answer $PR(V_i)$ are transported to secondary memory in an update. These two structures T_i and $PR(V_i)$ are stored in one block. So in that case, the update of the shadow administration takes one disk access and $O(f(n)+M(f(n)))$ transport and computing time. If the data structure is rebuilt, the entire file in secondary memory is also rebuilt. This takes one disk access and $O(S'(n))$ transport and computing time. This happens, however, at most once every $\Omega(f(n))$ updates. This leads to the bounds on the amortized update time.

Consider the reconstruction algorithm. It takes one disk access and $O(S'(n))$ transport and computing time to transport the shadow administration to main memory, and to put all information in the correct positions. The rebuilding of the tree T together with its partial answers takes $O(M'(n))$ computing time. Since $(n/f(n)) \times M(f(n)) \leq M'(n)$, the reconstruction computing time is bounded by $O(S'(n) + M'(n)) = O(n + M'(n))$. □

Now consider an $O(n)$-order decomposable set problem. Let $f(n) = \lceil n/\log n \rceil$. Then, by Corollary 2.5.1, there exists a dynamic data structure for this problem, with performances $S(n) = O(n \log\log n)$, $P(n) = O(n \log n)$, $Q(n) = O(1)$, and $U(n) = O(n)$, where the latter bound is amortized. Theorem 9.1.1 leads to

Theorem 9.1.2 *For the data structure solving an $O(n)$-order decomposable set problem, there exists a shadow administration with performances:*

1. $S'(n) = O(n)$.
2. $U_s(n) = 1$ *and the amortized values of* $U_c(n)$ *and* $U_t(n)$ *are both bounded by* $O(n/\log n)$.
3. $R_s(n) = 1$, $R_t(n) = O(n)$ *and* $R_c(n) = O(n \log\log n)$.

So again we have an example where the size of the shadow administration is asymptotically less than that of the data structure itself. Also, the reconstruction computing time is asymptotically less than the building time of the data structure.

An example of an $O(n)$-order decomposable set problems is the 3-dimensional convex hull problem: Two 3-dimensional convex hulls that are separated by a plane can be merged in linear time. See Preparata and Hong [46]. References to other examples have been given in Section 2.5.

9.2 Decomposable searching problems

As mentioned already in Section 2.4, there exist several techniques to dynamize static data structures that solve decomposable searching problems. Many of these techniques can be generalized to shadow administrations. We illustrate this by Bentley's logarithmic method, that was given in Section 2.4. In [56], other generalized techniques can be found.

Let *DS* be a static data structure for the decomposable searching problem *PR*. Let *SH*, *CSH* and *INF* form a shadow administration for *DS*. Note that we assume that $S(n)/n$, $S'(n)/n$, $P_c(n)/n$ and $R_c(n)/n$ are non-decreasing. Also we assume that the structure *SH* is not partitioned into parts. The reason for this latter assumption will be clear later.

The logarithmic method: Let V be a set of n objects, for which we want to solve the problem *PR*. As in Section 2.4, we write n in the binary number system, i.e., $n = \sum_{i \geq 0} a_i 2^i$, where $a_i \in \{0, 1\}$. Then we partition the set V into subsets V_0, V_1, V_2, etc., such that either V_i is empty or $|V_i| = 2^i$.

Our semi-dynamic data structure DS' consists of static structures DS_i of type *DS*, one for each non-empty set V_i. The insert algorithm and the complexity of DS' are given in Section 2.4.

The shadow administration we make for this semi-dynamic data structure DS' looks as follows. For each non-empty set V_i we make a shadow administration SH_i, CSH_i and INF_i. The structures CSH_i are

stored in secondary memory in consecutive blocks, starting at block 0, in decreasing size. That is, the file in secondary memory contains (in this order) the copy CSH_i representing the largest non-empty set V_i, then the copy $CSH_{i'}$ representing the second largest non-empty set $V_{i'}$, etc. At the end of the file, the copy $CSH_{i''}$ representing the smallest non-empty set $V_{i''}$ is stored. As usual, we store in secondary memory with each piece of information the address of the corresponding piece in main memory. Also we store in main memory for each structure SH_i the address of the beginning of its copy CSH_i.

The insert algorithm of the shadow administration: Suppose we insert object p into the set V. Let i be the minimal index for which $a_i = 0$. We discard the structures SH_j and INF_j for $j = 0, 1, \ldots, i-1$. Let $V_i := V_0 \cup V_1 \cup \ldots \cup V_{i-1} \cup \{p\}$; $V_0 := V_1 := \ldots := V_{i-1} := \emptyset$. Note that $|V_i| = 2^i$. We build additional structures SH_i and INF_i for this new set V_i. Then we transport a copy CSH_i of the resulting structure SH_i, together with the main memory addresses, to secondary memory. This copy is stored in the blocks containing the copies for the old sets V_0, V_1, ..., V_{i-1}—which are stored at the end of the file—and in some new blocks at the end of the file. (Since we transport a copy of the entire structure SH_i to secondary memory, there is no reason to partition it into parts. This explains why we assumed that the shadow administration SH is not partitioned.)

The reconstruction algorithm: To reconstruct the structures, we transport the copies CSH_i to main memory, and we store the information in the correct positions. Then each copy CSH_i takes over the role of the destroyed SH_i, and we reconstruct the structures DS_i and INF_i.

The following theorem gives the complexity of the shadow administration for DS'. In this theorem, $S'(n)$, $P_c(n)$ and $R_c(n)$ denote the complexity of the structures SH, CSH and INF.

Theorem 9.2.1 *For the semi-dynamic data structure DS', solving a decomposable searching problem, there exists a shadow administration, with performances:*

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

2. *An insertion takes one disk access; an amortized transport time of* $O(S'(n)/n)$ *if* $S'(n)/n^{1+\epsilon}$ *is non-decreasing for some* $\epsilon > 0$, *and* $O((S'(n)/n) \times \log n)$ *otherwise; and an amortized computing time of* $O(P_c(n)/n)$ *if* $P_c(n)/n^{1+\epsilon}$ *is non-decreasing for some* $\epsilon > 0$, *and* $O((P_c(n)/n) \times \log n)$ *otherwise.*

3. *Reconstruction takes one disk access,* $O(S'(n))$ *transport time, and* $O(R_c(n))$ *computing time.*

Proof. The storage required by the additional structures is bounded by

$$O\left(\sum_{i\geq 0} a_i\, S'(2^i)\right) = O\left(\sum_{i\geq 0} a_i\, 2^i\, S'(n)/n\right) = O(S'(n)),$$

since we assumed that $S'(n)/n$ is non-decreasing. Reconstruction takes one disk access, $O(S'(n))$ transport time, and $\sum_{i\geq 0} a_i R_c(2^i) = O(R_c(n))$ computing time. So we are left with the insertion time. The given insert algorithm takes one disk access, $O(S'(2^i))$ transport time and $O(P_c(2^i))$ computing time, for some integer i, $0 \leq i \leq \log n$. We derive an upper bound on the amortized insert complexity.

Suppose we start with an empty set V, and consider a sequence of n insertions. Let n_i be the number of times that additional structures for a set of cardinality 2^i are built. Each object is built at most once in a structure representing 2^i objects. (With an insertion, an object in set V_j, $0 \leq j < i$, moves to V_i, which has a higher index). Hence $2^i n_i \leq n$. So the total transport time required for these n insertions is bounded by

$$O\left(\sum_{i=0}^{\log n} n_i\, S'(2^i)\right) = O\left(n \sum_{i=0}^{\log n} S'(2^i)/2^i\right).$$

We have proved that the amortized transport time for an insertion is bounded by $O(\sum_{i=0}^{\log n} S'(2^i)/2^i)$. ¿From this it is easy to prove the bounds claimed in the theorem. In the same way, it can be shown that the amortized computing time for an insertion is bounded by $O(\sum_{i=0}^{\log n} P_c(2^i)/2^i)$. This completes the proof. □

As an illustration of the logarithmic method, let PR be the orthogonal range searching problem in the plane, which is indeed decomposable. In Theorem 8.3.2, we saw that there is a data structure DS for this problem—a range tree with slack parameter k—with performances $P(n) = O(n \log n)$, $S(n) = O((n \log n)/k)$ and $Q(n) = O((\log n)^2 2^k/k)$. Now apply Theorem 2.4.1. Then we get a semi-dynamic data structure DS', with performances $S(n) = O((n \log n)/k)$, $Q(n) = O((\log n)^3 2^k/k)$, and $I(n) = O((\log n)^2)$, where the latter bound is amortized.

The shadow administration SH for the data structure DS consists of an array that contains the points represented by DS ordered according to their y-coordinates. The copy CSH of the structure SH is stored in secondary memory as one part. We do not use a structure INF.

The complexity of these additional structures is given by $S'(n) = O(n)$, $P_c(n) = O(n \log n)$ and $R_c(n) = O((n \log n)/k)$, where the bound for $R_c(n)$ follows from Theorem 8.3.2.

Now apply Theorem 9.2.1. Note that if we build a shadow administration of size 2^i after an insertion, we merge sorted arrays of size $2^0, 2^1, \ldots, 2^{i-1}$. This merging can be done in $O(2^i)$ time. Therefore, we may apply Theorem 9.2.1 with $P_c(n) = O(n)$. This leads to the following theorem.

Theorem 9.2.2 *For the semi-dynamic data structure DS', solving the orthogonal range searching problem in the plane, there exists a shadow administration, with performances:*

1. *The storage is bounded by $O(n)$.*
2. *An insertion takes one disk access, and an amortized transport and computing time of $O(\log n)$.*
3. *Reconstruction takes one disk access, $O(n)$ transport time and $O((n \log n)/k)$ computing time.*

Note that in this example, the data structure DS' is less efficient than the dynamic version of the range tree (see Theorem 8.3.2): The query time increases by a factor $\log n$. Also, no deletions are possible. The usefulness of the logarithmic method, however, is in the complexity of the shadow administration: An insertion requires only one disk

access, and an amortized transport and computing time of $O(\log n)$. Compare this to Theorem 8.3.3, where an insertion can cost 3 disk accesses, but only $O(1)$ transport time.

9.3 A general technique

9.3.1 Introduction

Let *DS* be a dynamic data structure of size $S(n)$, having an update time $U(n)$. In Subsection 8.3.2, we gave a technique, in which we transport with an update the object, together with information whether it concerns an insertion or a deletion, to secondary memory. After $S(n)/U(n)$ updates, we transport a copy of the up-to-date data structure to secondary memory. In this way, the shadow administration is updated at the cost of one disk access and an amortized amount of $O(U(n))$ transport and computing time.

Let $C(n)$ denote the amount of data that is changed in *DS* in an update. Then clearly, $C(n) \leq U(n)$. In this section, we reduce the update transport time of the shadow administration to $O(C(n))$. The idea is to transport with each update the (at most) $C(n)$ changed entries in the data structure *DS*, together with their addresses, to secondary memory. These changed entries and their positions are stored in one block at the end of the file. To reconstruct the data structure *DS*, we transport the file from secondary memory to main memory, and we perform the most recent updates. Since we know the positions of the entries that have to be changed in these updates, together with the updated entries themselves, each update can be performed in $O(C(n))$ time.

This is the main idea of the general technique of this section. In order to get a more general result, we apply this idea to the shadow administration *SH*, instead of to *DS*.

Let *DS* be a dynamic data structure and let *SH* and *INF* be some corresponding shadow administration. We denote the size of *SH* by $S_{SH}(n)$, the size of *INF* by $S_{INF}(n)$, the total update computing time of *SH* and *INF* by $U_c(n)$, and the computing time needed to reconstruct

the structures *DS* and *INF* from *SH* by $R_c(n)$. Let $C(n)$ be the amount of data that is changed in an update in *SH*. We assume that all these complexity measures are smooth and non-decreasing.

We show how to implement these structures, such that the entire shadow administration can be updated in two disk accesses, $O(U_c(n))$ computing time and $O(C(n))$ transport time. These bounds are worst-case bounds. Also, the total size of the additional structures is bounded by $O(S_{SH}(n) + S_{INF}(n))$, and reconstruction takes three disk accesses, $O(S_{SH}(n))$ transport time, and $O(R_c(n))$ computing time. This result is obtained in two steps. We first give an amortized solution. Then we turn the amortized bounds into worst-case bounds.

We need the following lemma.

Lemma 9.3.1 *The complexity measures introduced above satisfy*

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

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

9.3.2 An amortized solution

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

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

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

The reconstruction algorithm: To reconstruct the structures, we transport *CSH* and *UF* to main memory, where we store *CSH* in the correct locations using the addresses. Then pointers in *CSH* indeed "point" to the correct objects. Next we carry out the at most $S_{SH}(m)/(2C(m))$ updates using the list *UF*. (This list gives us the addresses of the entries in *CSH* that have to be changed, and the new contents of these entries.) After these updates, the resulting structure *CSH* contains the up-to-date shadow administration. Hence it can take over the role of *SH*. Finally, we reconstruct from *SH* the structures *DS* and *INF*. Then all information is reconstructed, and we can proceed with answering queries and performing updates.

Theorem 9.3.1 *Let SH and INF be a shadow administration for the dynamic data structure DS, with complexity $S_{SH}(n)$, $S_{INF}(n)$, $U_c(n)$, $R_c(n)$ and $C(n)$. We can implement these structures such that the resulting shadow administration*

1. *has size $O(S_{SH}(n) + S_{INF}(n))$,*

2. *can be updated in one disk access, an amortized computing time of* $O(U_c(n))$*, and an amortized transport time of* $O(C(n))$.

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

Proof. First note that all information in secondary memory is stored in consecutive blocks, always starting at block 0. In particular, there are no gaps. Therefore, the amount of space used in secondary memory is proportional to the total size of the structures *CSH* and *UF*. The size of *CSH*, together with the corresponding addresses in main memory, is equal to $O(S_{SH}(m))$, where m is the number of objects at the beginning of the sequence of updates. During this sequence, n—the current number of objects—satisfies $n \leq m + S_{SH}(m)/(2C(m))$. It follows from Lemma 9.3.1, that $n \leq 3m/2$. Similarly, $n \geq m/2$, and hence $n = \Theta(m)$. Since our complexity measures are assumed to be smooth, we have $C(n) = \Theta(C(m))$. Hence in each update we add $O(C(n)) = O(C(m))$ data to the list *UF*. It follows that the size of *UF* is bounded by $(S_{SH}(m)/(2C(m))) \times O(C(m)) = O(S_{SH}(m))$. Therefore, the total amount of space used in secondary memory is bounded by $O(S_{SH}(m)) = O(S_{SH}(n))$. The amount of space used in main memory by the shadow administration is bounded by $S_{SH}(n) + S_{INF}(n)$. This proves the bound on the space complexity.

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

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

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

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

Here we have used Lemma 9.3.1.

In the reconstruction algorithm, it takes one disk access and $O(S_{SH}(n))$ transport time and computing time to transport *CSH* and *UF* to main memory and to store *CSH* in the correct positions. Each update from the list *UF* takes $O(C(m))$ computing time. It follows that all updates from *UF* together take an amount of computing time that is bounded by $O((S_{SH}(m)/C(m)) \times C(m)) = O(S_{SH}(m))$. Finally, it takes $R_c(n)$ computing time to reconstruct the structures *DS* and *INF* from the up-to-date structure *CSH*. Hence, the entire reconstruction algorithm takes one disk access, $O(S_{SH}(n))$ transport time and $O(S_{SH}(n)+R_c(n))$ computing time. □

9.3.3 A worst-case solution

In this subsection, we assume that the update computing time $U_c(n)$ of the structures *SH* and *INF*, and the amount of data $C(n)$ that is changed in *SH* are worst-case bounds. We turn the amortized bounds of the preceding subsection into worst-case bounds. The idea is to spread out the transport of the copy of *SH* over a number of updates. The technique is related to the global rebuilding technique given in Overmars [42].

The structures: Let m be the number of objects that are initially represented by the data structure *DS*. We store in main memory, the structure *DS* and two copies of each of the corresponding additional structures *SH* and *INF*. We denote these copies by SH_1, INF_1, SH_2 and INF_2. In secondary memory we store—in consecutive blocks, starting at the block "next to" block 0—a copy CSH_1 of the structure SH_1. Of course, this copy CSH_1 contains with each piece of information the address of the corresponding piece in main memory. In block 0, we store the addresses of the first and the last block of the segment that contains CSH_1. Initially, all structures are up-to-date. We initialize in main memory an empty list L.

To initialize the process, we perform an initial stage, that consists of the first $S_{SH}(m)/(2C(m))$ updates. This stage is split in two parts.

Part 1 of the initial stage: This part consists of the first $S_{SH}(m)/(4C(m))$ updates. These updates are performed on the structures *DS*, SH_1 and

INF_1. With each update, we store the object, together with information whether it concerns an insertion or a deletion, at the end of the (main memory) list L.

Furthermore, with each update, we do the following. First, we collect the changed entries in SH_1, together with their addresses. These changed entries become a part of the update file UF_1. Next consider the structure SH_2. This structure has size $S_{SH}(m)$. Now with this update we collect a part of SH_2, together with their addresses, of size

$$\frac{S_{SH}(m)}{S_{SH}(m)/(4C(m))} = O(C(m)).$$

This part becomes a part of the copy CSH_2. We transport to secondary memory, the part of UF_1 and the part of CSH_2. These parts are stored in two consecutive blocks at the end of the file. We also replace block 0 by a new block 0 containing the addresses of the first and the last block of the file. Note that during these updates, the structures SH_2 and INF_2 cannot be affected. Also, after this part of the initial stage, the entire copy CSH_2 has been transported to secondary memory.

After Part 1: After the first part of this initial stage, main memory contains an up-to-date data structure DS, up-to-date structures SH_1 and INF_1, a list L of the updates performed so far, and structures SH_2 and INF_2 that store the objects that were present $S_{SH}(m)/(4C(m))$ updates ago. Secondary memory contains structures CSH_1 and CSH_2 that store the objects that were present $S_{SH}(m)/(4C(m))$ updates ago, and a list UF_1 of the changes of the $S_{SH}(m)/(4C(m))$ most recently performed updates. The structure CSH_1 is stored in consecutive blocks. The structures UF_1 and CSH_2 are also stored in consecutive blocks, but mixed up together.

Part 2 of the initial stage: This part consists of the final $S_{SH}(m)/(4C(m))$ updates. We perform these updates on the structures DS, SH_1 and INF_1. In order to make the structures SH_2 and INF_2 up-to-date, we perform with each update, two updates from the list L. Then we remove these two updates from L, and we add the actual update at the end of it. (Note that the updates have to be performed in chronological order, since one object can be inserted and deleted several times.)

Also, with each update, we do the following. Again we collect the changed entries in SH_1, together with their addresses. These changed

entries become a part of UF_1. We also collect the changed entries—that are caused by two updates—in SH_2, together with their addresses. These changes become a part of the update file UF_2. We transport to secondary memory, the part of UF_1 and the part of UF_2. Again, these parts are stored in two consecutive blocks at the end of the file. Finally, we update block 0.

After Part 2: After the second part of this initial stage, main memory contains up-to-date structures DS, SH_1, INF_1, SH_2 and INF_2, and an empty list L. Secondary memory contains structures CSH_1 and CSH_2 that store the objects that were present before the initial stage, and lists UF_1 and UF_2 that contain the changes that were made in SH_1 and SH_2 during the initial stage. The structure CSH_1 is stored in consecutive blocks. The structure CSH_2 and the first half of UF_1 are stored mixed up in consecutive blocks. Also, the structure UF_2 and the second half of UF_1 are stored in consecutive blocks, again mixed up.

The reconstruction algorithm: During the initial stage, reconstruction can be done as follows. We transport all information—that is stored in consecutive blocks, starting at the block next to block 0—to main memory. (During the initial stage, we do not need block 0.) We discard the structures CSH_2 and UF_2. We store CSH_1 in the correct positions in main memory, and we perform the most recent updates, using the list UF_1. Then CSH_1 is up-to-date, and it takes over the role of the destroyed SH_1. Next, we reconstruct the information as it was before the initial stage. That is, we make a copy SH_2 of SH_1, and we reconstruct the structures DS, INF_1 and INF_2. Finally, we transport a copy of SH_1, together with the addresses of the pieces of information, to secondary memory. We store this copy again in consecutive blocks, starting at the block next to block 0. In block 0, we store the addresses of the first and the last block of the segment that store the copy of SH_1. Then all necessary information is reconstructed, and we are in the same situation as before the initial stage. Now we can proceed answering queries and performing updates.

After the initial stage: After this initial stage, we "discard" in secondary memory the structure CSH_1, that is stored in consecutive

blocks at the beginning of the file. We do this by transporting to secondary memory a new block 0, that stores the addresses of the new first block of the file—which is the first block that stores information from CSH_2—and the last block of the file. (The address of the last block does not change.) Then, the structure UF_1 has no use anymore. We do not, however, discard this update file UF_1, that is stored mixed up with CSH_2 and UF_2. ¿From now on we write UF_1' for this structure, in order to distinguish it from the new structure UF_1 that is made in the sequel. Note that at this moment the structures CSH_2 and UF_2 contain enough information to reconstruct the other structures.

Now the periodic process of updating can start. The process is similar to the initial stage.

Before the regular stage: At the start we have in main memory up-to-date structures DS, SH_1, INF_1, SH_2 and INF_2, and an empty list L. Secondary memory contains a structure CSH_2 that stores the objects that were present $S_{SH}(m)/(2C(m))$ updates ago, and lists UF_1' and UF_2 that contain the changes that were made in SH_1 and SH_2 during the most recent $S_{SH}(m)/(2C(m))$ updates. The structure CSH_2 and the first half of UF_1' are stored mixed up in consecutive blocks. Also, the structure UF_2 and the second half of UF_1' are stored mixed up in consecutive blocks.

Let m_0 be the number of objects that are represented by DS at this moment. Consider a sequence of $S_{SH}(m_0)/(2C(m_0))$ updates. Again, we split this sequence in two stages.

Part 1 of the regular stage: The first $S_{SH}(m_0)/(4C(m_0))$ updates are performed as follows. Each update is carried out on the structures DS, SH_2 and INF_2. With each update, we store the object, together with information whether it concerns an insertion or a deletion, at the end of the list L.

Also, with each update we collect the changed entries in SH_2, together with their addresses. These changed entries become a part of UF_2. We collect a part of SH_1, together with their addresses, of size

$$\frac{S_{SH}(m_0)}{S_{SH}(m_0)/(4C(m_0))} = O(C(m_0)).$$

This part becomes a part of the copy CSH_1. We transport to secondary

memory, the part of UF_2 and the part of CSH_1. These parts are stored in two consecutive blocks at the end of the file. We also replace block 0 by a new block 0 containing the addresses of the first and the last block of the file. The entire copy CSH_1 is transported to secondary memory, during the first part of the regular stage.

After Part 1: After the first part of the regular stage, main memory contains up-to-date structures DS, SH_2 and INF_2, a list L of the updates of Part 1, and structures SH_1 and INF_1 that store the objects that were present at the beginning of Part 1. Secondary memory contains a structure CSH_2 that stores the objects that were present $S_{SH}(m)/(2C(m)) + S_{SH}(m_0)/(4C(m_0))$ updates ago, a structure CSH_1 storing the objects that were present $S_{SH}(m_0)/(4C(m_0))$ updates ago, a list UF'_1 that stores information that is irrelevant now, and a list UF_2 of the changes of the $S_{SH}(m)/(2C(m)) + S_{SH}(m_0)/(4C(m_0))$ most recent updates. The structure CSH_2 and the first half of UF'_1 are stored mixed up in consecutive blocks. The changes of the first $S_{SH}(m)/(2C(m))$ updates in UF_2 and the second half of UF'_1 are stored mixed up in consecutive blocks. Finally, the structure CSH_1 and the rest of UF_2 are stored mixed up in consecutive blocks.

Part 2 of the regular stage: Part 2 consists of the final $S_{SH}(m_0)/(4C(m_0))$ updates. These updates are performed in the same way as in Part 2 of the initial stage. (Interchange the indices 1 and 2.) The changes in SH_1 are stored in secondary memory in the update file UF_1.

After Part 2: After this second part of the regular stage, main memory contains up-to-date structures DS, SH_1, INF_1, SH_2 and INF_2, and an empty list L. Secondary memory contains a structure CSH_2 that stores the objects that were present $S_{SH}(m)/(2C(m))+S_{SH}(m_0)/(2C(m_0))$ updates ago, a corresponding list UF_2 that stores the changes of these updates, a structure CSH_1 storing the objects that were present $S_{SH}(m_0)/(2C(m_0))$ updates ago, a corresponding list UF_1 that stores the changes of these updates, and finally an old list UF'_1. The structures CSH_2, UF'_1 and the first piece of UF_2 are stored mixed up in consecutive blocks. The structure CSH_1 and the second piece of UF_2 are stored mixed up in consecutive blocks. Also, the structure UF_1 and the final piece of UF_2 are stored mixed up in consecutive blocks.

The reconstruction algorithm: During this regular stage, re-

construction can be done from the structures CSH_2 and UF_2, in a similar way as during the initial stage. We first transport block 0 to main memory. Then we know the addresses of the segment of blocks in which the information is stored. We now transport this information to main memory, and we reconstruct the information as it was before the initial stage. After reconstruction, we transport a copy CSH_1 of SH_1, together with the addresses of the pieces of information, to secondary memory. We store this copy again in consecutive blocks, starting at the block next to block 0. In block 0, we store the addresses of the first and the last block of the segment that stores the copy CSH_1.

After the regular stage: After the regular stage, we "discard" in secondary memory the structures UF'_1, CSH_2 and the first piece of UF_2. Again, this is done by transporting to secondary memory a new block 0, that stores the addresses of the new first block of the file—which is the first block that stores information from CSH_1—and the last block of the file. (The address of this last block has not changed.) Note that all "discarded" structures are stored in consecutive blocks, at the front of the file, so this does not lead to gaps in secondary memory.

We end with in main memory up-to-date structures DS, SH_1, INF_1, SH_2 and INF_2, and an empty list L. Secondary memory contains a structure CSH_1, that contains the objects that were present $S_{SH}(m_0)/(2C(m_0))$ updates ago, and lists UF_1 and UF_2 that contain the changes that were made in SH_1 and SH_2 during the most recent $S_{SH}(m_0)/(2C(m_0))$ updates. The structure CSH_1 and the first half of UF_2 are stored mixed up in consecutive blocks. Also, the structure UF_1 and the rest of UF_2 are stored mixed up in consecutive blocks.

It follows that we are in the same situation as before Part 1 of the regular stage. Therefore, we can proceed performing updates in the same way, now with a sequence of length $S_{SH}(m')/(2C(m'))$, where m' is the number of objects at this moment. (Of course, we have to interchange the indices 1 and 2.)

Since we only add information at the end of the file in secondary memory, and since we only remove information from the front of this file, all structures in secondary memory are stored in consecutive blocks. There are no gaps. Of course, if the structures are "moved too far to the

right", we can add information at the front, and remove information from the end of the file. Note that we do not really remove information in secondary memory, since the information remains stored in the blocks. We can, however, use these blocks again, since they contain information that is not needed for reconstruction. Block 0 contains the necessary information where the current shadow administration is stored. It follows that the amount of space we use in secondary memory is proportional to the total size of all blocks that store the current shadow administration.

Theorem 9.3.2 *Let SH and INF be a shadow administration for the dynamic data structure DS, with complexity $S_{SH}(n)$, $S_{INF}(n)$, $U_c(n)$, $R_c(n)$ and $C(n)$. We can implement these structures such that the resulting shadow administration*

1. *has size $O(S_{SH}(n) + S_{INF}(n))$,*
2. *can be updated in two disk accesses, $O(U_c(n))$ computing time and $O(C(n))$ transport time in the worst case.*

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

Proof. The theorem can be proved by carefully checking the given algorithms. In the same way as in Subsection 9.3.2, it can be shown that the integers n, m and m_0 satisfy $n = \Theta(m) = \Theta(m_0)$. Note that two disk accesses are required for an update: One for updating block 0, and one for transporting the two blocks to the end of the file. Also, three disk accesses are required for reconstruction: One disk access for block 0, one for the shadow administration, and one for transporting the new shadow administration back to secondary memory. □

We illustrate this theorem with an example. In Section 10.4, we will see another example.

Consider a range tree with slack parameter k. See Definition 8.3.1. As we saw in Theorem 8.3.2, in such a range tree, representing n points in the plane, range queries can be solved in $O((\log n)^2 2^k/k +$

t) time, if t is the number of reported answers. The structure has size $O((n \log n)/k)$, and can be built in $O(n \log n + (n \log n)/k)$ time. Here the first term is the time to sort the n points according to their y-coordinates, whereas the second term is the actual building time. Therefore, just as in Subsection 8.3.1, let the structure SH consist of a list that contains the n points ordered according to their y-coordinates. The structure INF consists of a balanced binary tree, that stores the points in its leaves, also ordered according to their y-coordinates. Each leaf of this tree contains a pointer to the corresponding point in the list. The complexity of this shadow administration is given by $S_{SH}(n) = O(n)$, $S_{INF}(n) = O(n)$, $U_c(n) = O(\log n)$ and $R_c(n) = O((n \log n)/k)$. Since an update changes only a constant amount of data in the sorted list SH, we have $C(n) = O(1)$. Now apply Theorem 9.3.2 to get:

Theorem 9.3.3 *For a range tree with slack parameter k, there exists a shadow administration*

1. *of size $O(n)$.*
2. *that can be updated in two disk accesses, $O(\log n)$ computing time and $O(1)$ transport time in the worst case.*
3. *from which the range tree can be reconstructed in three disk accesses, $O(n)$ transport time and $O((n \log n)/k)$ computing time.*

Compare this result to Theorems 8.3.3 and 9.2.2. In Theorem 8.3.3, an update can take 3 disk accesses, whereas in Theorem 9.2.2, only insertions are possible, at the cost of one disk access and an amortized amount of $O(\log n)$ transport and computing time.

Chapter 10

A union-find data structure

10.1 Introduction

Until now we have seen several techniques that apply to arbitrary searching problems or to classes of searching problems that satisfy some constraints. In this chapter we consider a specific problem, the Union-Find Problem, and we design an efficient main memory data structure for it. This structure is designed in such a way that a copy of it can efficiently be maintained in secondary memory.

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.

We require that the operations are carried out on-line, i.e. each operation has to be completed before the next is known.

The union-find problem has many applications, and many algorithms use the problem in some way as a subroutine. Examples are algorithms for computing minimum spanning trees, solving an off-line

minimum problem, computing depths in trees and determining the equivalence of finite automata. (See [2].)

The problem has received considerable attention. Tarjan showed in [57] that a sequence of m *UNION* and *FIND* operations can be carried out in total time $O(m\,\alpha(m+n,n)+n)$ using $O(n)$ space, where α is a functional inverse of Ackermann's function, which is a very, very slow growing function. Furthermore, he introduced in [58] a machine model—the Pointer Machine, see also Section 2.6—on which all known important algorithms solving the union-find problem can be implemented. Tarjan showed that on a Pointer Machine, any algorithm for the union-find problem needs $\Omega(m\,\alpha(m+n,n)+n)$ time for performing m *UNION* and *FIND* operations. (See also [60].)

In this chapter we are interested in the single-operation time complexity of the union-find problem. Until recently, only algorithms were known having single-operation complexity $\Omega(\log n)$. That is, there is always either a *UNION* or a *FIND* operation that needs $\Omega(\log n)$ time. In [12], Blum gives 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. Blum also gives the following very general class $\mathcal{B}$ of data structures—containing all implementations on Tarjan's Pointer Machine:

The class $\mathcal{B}$: Data structure 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 the 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 different subgraphs.

3. To perform an operation *FIND*(x), the algorithm obtains the node v that contains x. The algorithm follows paths in the graph, until it reaches the node that contains the name of the corresponding set.

4. To perform a *UNION* or a *FIND* operation, the algorithm may insert or delete any edge, as long as rule 2 is satisfied.

For structures in class $\mathcal{B}$, the following theorem holds. For a proof, the reader is referred to [12].

Theorem 10.1.1 (Blum) *Let DS be any data structure from the class $\mathcal{B}$. Suppose that every UNION operation can be performed in $O(k)$ time. Then there is a FIND operation that needs time*

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

As a corollary of this theorem we see that for each data structure in class $\mathcal{B}$, there is always either a *UNION* or a *FIND* operation that takes $\Omega(\log n/\log\log n)$ time.

We will first give a variant of Blum's structure, having the same complexity. That is, we give a structure of size $O(n)$, in which each *UNION* resp. *FIND* operation takes $O(k+\log_k n)$ resp. $O(\log_k n)$ time. Next, we adapt this structure such that each *UNION* operation can be carried out in $O(k)$ time, whereas the size of the structure and the time for a *FIND* operation remain the same. This structure is in Blum's class $\mathcal{B}$. Hence it follows from Theorem 10.1.1 that this structure is optimal—in class $\mathcal{B}$—if $k = \Omega((\log n)^\epsilon)$ for some $\epsilon > 0$.

The improved data structure consists of a number of trees—each set is stored in one such tree—and has the property that for a *UNION* operation we only have to visit the roots of two trees, together with their—at most k—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.

10.2 A variant of Blum's structure

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

Definition 10.2.1 Let k be an integer, $2 \leq k \leq n$. A tree T is called a *UF*(k)*-tree*, if

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

As mentioned already, we store each set A in the partition of V in a separate *UF*(k)-tree. 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. Finally, the root of the tree contains a pointer to each of its sons, and a pointer to an (arbitrary) leaf. Note that the root contains at most $k+1$ pointers. A *UF*(k)-tree storing a set of cardinality one, has two nodes, a root and one leaf.

The Find-algorithm: To perform an operation *FIND*(x), we get at constant cost the leaf containing element x. Then we follow father-pointers until we reach the root of the tree, where we read the name of the set containing 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 containing 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.

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 two pointers to r and s; a pointer to a leaf of the new tree (we can take the corresponding pointer stored in r); 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.

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. We find in the tree of A a node v such that the subtree of v has the same height as the tree of B. This node v can be found by following the pointer from r to a leaf, and then by walking up in the tree. Then we change the father-pointers from all sons of s into pointers to v. (This guarantees that all leaves remain at the same level.) We discard the root s, together with all its information. Finally, 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.

Theorem 10.2.1 *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 $O(k + \log_k n)$ time,*
2. *each FIND takes $O(\log_k n)$ time,*
3. *the data structure has size $O(n)$.*

Proof. The time needed to perform a *FIND* operation is bounded above by the height of a $UF(k)$-tree. It follows from Definition 10.2.1 that if level i in a $UF(k)$-tree contains any nodes, it contains at least $k^{\lfloor i/2 \rfloor}$ of them. (Here the root is at level 0.) Since such a tree has at most n leaves, its height is at most $1 + 2\lceil \log_k n \rceil$. Hence a *FIND* operation takes $O(\log_k n)$ time in the worst case.

It is easy to see that the given *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 relevant information for deciding this is stored in the roots. Case 1 resp. 2 of the *UNION*-algorithm takes $O(k)$ resp. $O(1)$ time in the worst case. In Case 3, it takes $O(\log_k n)$ time to find the node v, whereas the rest of this case can be carried out in $O(k)$ 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 $O(n)$. □

10.3 An improved data structure

We saw in Section 10.2, that it takes $O(k + \log_k n)$ time to perform a *UNION* operation on $UF(k)$-trees. In this time bound, the $O(\log_k n)$ term is due to the fact that in Case 3, we have to search the node v, the subtree of which has the same height as the tree containing B. Clearly, if we take instead of v an arbitrary son of the root of the tree of A, the time for each *UNION* operation will be bounded by $O(k)$. It remains to prove then, however, that the heights of the trees do not increase.

Definition 10.3.1 Let k be an integer, $2 \leq k \leq n$. A tree T is called an $IUF(k)$-*tree* (where the I stands for *improved*), if

1. the root of T has at most k sons,

2. each node in T has either 0 or more than k grandsons.

Again, we store each set A in the partition of V in a separate $IUF(k)$-tree. 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. Also each non-root node contains a pointer to its father, and the root contains pointers to all its sons. (Now we do not need a pointer from the root to a leaf.)

Note that Definition 10.3.1 does not imply anymore that the heights of these trees are bounded above by $1 + 2\lceil \log_k n \rceil$, since the leaves do not have to be positioned at the same level. The trees that are made by the *UNION*-algorithm to be described below, however, do have heights bounded by $1 + 2\lceil \log_k n \rceil$.

The *FIND*-algorithm for $IUF(k)$-trees is the same as for $UF(k)$-trees.

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. As before, 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 $\leq k$. This case is handled in the same way as Case 1 of Section 10.2.

Case 2′. The trees containing A and B have equal height, and the total number of sons of r and s is $> k$. This case is handled as Case 2 of Section 10.2.

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.

First note that the given algorithm correctly maintains $IUF(k)$-trees. Furthermore, each *UNION* operation takes $O(k)$ time, since in Case 3′, the node v can be found in constant time. Also, the size of the data structure is still bounded by $O(n)$.

It remains to prove that the heights of the $IUF(k)$-trees, that are made by the given *UNION*-algorithm, are bounded by $O(\log_k n)$. It suffices to prove that an $IUF(k)$-tree has the same height as the corresponding $UF(k)$-tree that stores the same set.

Suppose we are given a collection of n disjoint sets $S_1, S_2, \ldots, S_n$, each containing one single element, and consider a sequence of *UNION* and *FIND* operations. Let *UF* be the data structure, consisting of $UF(k)$-trees, if we perform these operations according to the algorithm of Section 10.2. Furthermore, let *IUF* be the data structure consisting of $IUF(k)$-trees, where the operations are carried out as described in the current section.

So if $\bigcup_{i \in I} A_i$ is a partition of the n elements at some moment in the sequence of operations, there are two data structures. First, there is a structure $UF = \{T_i | i \in I\}$, where each T_i is a $UF(k)$-tree storing the set A_i. Also, there is a structure $IUF = \{T_i' | i \in I\}$, where each T_i' is an $IUF(k)$-tree storing A_i. Now each *UNION* operation is performed—in parallel—on both *UF* and *IUF*. (The purpose for doing this is to prove the upper bound on the heights of $IUF(k)$-trees.)

Lemma 10.3.1 *At each moment, the trees T_i and T_i' have the same height, and the roots of these trees have the same number of sons. To perform a UNION operation, if we are in Case j for the structure UF, we are in Case j' for IUF, for $j = 1, 2, 3$.*

Proof. Initially, each tree T_i and T_i' contains two nodes, one root and one leaf. Hence at the beginning the statement is true.

Then the lemma can easily be proved, using the following observation. Whether we are in Case 1, 2 or 3 of the *UNION*-algorithm for the structure *UF*, depends only on the heights of the trees and on the number of sons of the corresponding roots. The same holds for the structure *IUF*. By checking the *UNION*-algorithms, it follows that a *UNION* operation leaves the statement in the lemma invariant. □

Theorem 10.3.1 *Let k and n be integers, such that $2 \leq k \leq n$. Using $IUF(k)$-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 have seen already that a *UNION* operation takes $O(k)$ time, and that the size of the data structure is bounded by $O(n)$. It follows from Lemma 10.3.1 and the proof of Theorem 10.2.1, that the height of an $IUF(k)$-tree, made by the given *UNION*-algorithm, is at most $1 + 2\lceil \log_k n \rceil$. Hence each *FIND* operation takes $O(\log_k n)$ time. □

It is clear that the data structure *IUF* is contained in Blum's class $\mathcal{B}$. Therefore, Theorems 10.1.1 and 10.3.1 yield the following corollary.

Corollary 10.3.1 *The data structure of Theorem 10.3.1 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 $\epsilon > 0$.*

10.4 An efficient shadow administration

In this section we show how we can efficiently maintain a copy of the data structure *IUF* of Theorem 10.3.1 in secondary memory.

First, we want to remark that the data structure *IUF* can be implemented on a Pointer Machine. (See Section 2.6.) On a Pointer Machine, no addressing of memory locations is possible. Therefore, to implement this structure together with a copy in secondary memory, as we do now, we need a Random Access Machine as main memory, because we have to maintain in secondary memory the addresses of the entries in main memory in which the information is stored.

We store a copy of the data structure *IUF* 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, as usual.

Since the root of an $IUF(k)$-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 *IUF*, together with all their 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 *IUF* 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 10.4.1 *Among the reserved blocks, there is always at least one free block.*

Initially we have n trees, each of them having one root and one leaf. We store these trees in main memory. Copies of the trees are distributed over the reserved blocks. For each tree, the root and its son, together of course with 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 10.4.1, this stack is never empty. The stack will only be used for updating the structure in secondary memory; it is not used for reconstructing the data structure. Therefore it may be

destroyed in a crash. Note that the amount of space in main memory remains bounded by $O(n)$.

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

A *UNION* operation is first performed on the structure in main memory according to the algorithm of Section 10.3. Then the shadow administration in secondary memory is updated. We take care that at each moment the following holds:

> **Invariant:** For each $IUF(k)$-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.

If we want to reconstruct the data structure *IUF*, 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. This reconstruction algorithm takes one disk access and an amount of $O(n)$ transport time and computing time.

The following theorem summarizes the result.

Theorem 10.4.1 *Let k and n be integers, such that $2 \leq k \leq n$. For the data structure of Theorem 10.3.1, 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.

Proof. The proof follows from the above discussion. □

Remark. In the above shadow administration, a *UNION* operation requires three disk accesses. Since each *UNION* operation takes $O(k)$ time, it is clear that such an operation changes only an amount of

$O(k)$ data in the structure. Therefore, we can apply Theorem 9.3.2, to get a shadow administration with the same complexity as that of the above theorem, except that a *UNION* operation requires only two disk accesses and reconstruction requires three disk accesses. The shadow administration of Theorem 10.4.1, however, is easier to implement.

Chapter 11

Another approach: deferred data structuring

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

This technique of building a data structure is due to Karp, Motwani and Raghavan [28, 35], who call it *deferred data structuring*, although they do not apply this technique to the reconstruction problem. Their motivation to design deferred data structures is to solve a sequence of queries, where the length of the sequence is not known. They only give static deferred data structures. The design of deferred data structures for dynamic data sets in which insertions and deletions are allowed concurrently with queries is stated as an open problem.

In this chapter, we show that it is often possible to design dynamic deferred data structures by using well-known dynamization techniques. The ideas are illustrated by considering dynamic deferred structures for the member searching problem. We show that deferred binary search trees—if properly chosen—can be maintained as in the ordinary case, i.e., by means of rotations. This observation was made, independently, by Ching and Mehlhorn [16]. We also adapt Lueker's partial rebuilding technique, to get another maintenance algorithm for deferred search trees. Finally, we give a trivial solution, based on the ideas of decomposable searching problems and global rebuilding. See Overmars [42] for the notion of global rebuilding.

In Section 11.3, we show how deferred data structures can be used to solve the reconstruction problem.

11.1 The static deferred binary search tree

We first recall the static solution of [28] 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. (In this section, we store the objects in the *nodes* of the tree.) 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 takes linear time [11, 49]. 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. (The notations we introduce here are used in the rest of the chapter.) Each node v in the structure contains a list $L(v)$ of objects, two variables $N(v)$ and $key(v)$, and two pointers. Some of these values may be undefined. The value of $N(v)$ is equal to the number of objects that are stored in the subtree with root v. The meaning of the other variables will be clear from the algorithms below. (Strictly speaking, the variable $N(v)$ is not needed in the static case.)

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

Expand: Let v be a node having an undefined variable $key(v)$. In this case, the list $L(v)$ will contain at least 2 objects, and the value of $N(v)$ will be equal to $|L(v)|$. 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 [28] or Section 11.2. (The proof in Section 11.2 is a generalization of the proof in [28] to the dynamic case.)

Theorem 11.1.1 *A sequence of k member queries in a set of n objects can be solved in total time $O(n \log k)$ if $k \leq n$, and $O((n+k) \log n)$ if $k > n$.*

In [28], it is shown that this theorem gives an optimal result: The number of comparisons needed to perform k member queries in a set of size n is $\Omega((n+k) \times \log \min(n,k))$. In fact, this lower bound even holds in the off-line case, i.e., in case the queries are known in advance.

11.2 Three dynamic solutions

We only consider sequences of at most n queries, insertions and deletions. Clearly, this suffices, since after n operations we will have spent already $\Omega(n \log n)$ time. Therefore, in the n-th operation, we can build a complete data structure—in $O(n \log n)$ time—and continue in the standard non-deferred way.

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

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

If $key(v)$ is undefined, node v contains a list $L(v)$ storing a subset of V—those objects that "belong" in the subtree of v—and the variable $N(v)$ has the value $|L(v)|$, which is at least two.

An insertion algorithm: 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.

Otherwise, 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 this will take $O(|L(w)|)$ 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.

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

A deletion algorithm: A deletion of object x is performed in a similar way. We start searching for x.

First suppose we find a node v with $key(v) = x$. Then we search in the left subtree of v for the maximal object y. We know the path that leads to this maximal object. If we end in a leaf, we interchange x and y, i.e., we set $key(v) := y$, and the *key* value of the leaf is set to x. Then we delete this leaf in the standard way. During the search we decrease the N values in all nodes we encounter by one. If we do not end in a leaf during our search for y, we reach a node w with an undefined *key* value. Then we remove y from the list $L(w)$, we set $key(v) := y$, and we expand node w. Just as in the insertion algorithm, if we again have to delete an object in the subtree of w, the time for this deletion is halved.

If we do not encounter a node v with $key(v) = x$ during our search for x, we might reach a node v with an undefined *key* value. If x is present in the deferred tree, it is stored in the list $L(v)$. So we delete x from this list, and we expand node v.

Note that if x is not present in the tree we will find this out. (In that case, we have to adjust the changed N values.)

We are left with the problem of keeping the deferred tree balanced. There are various types of balanced binary search trees that can be maintained after insertions and deletions. The oldest are the AVL-trees, see Section 2.2. The balance condition for these trees depends on the exact heights of subtrees. Since in our deferred tree several subtrees are not complete during the sequence of operations, their exact heights will not be known. So AVL-trees do not seem appropriate for deferred trees.

We have seen, however, the class of BB[α]-trees, for which 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 $N(v)$.

Balancing by means of rotations: Our deferred search tree will be a BB[α]-tree. That is, for each internal node v for which the value of $key(v)$ is defined, we require that $\alpha \leq N(v_l)/N(v) \leq 1 - \alpha$, where v_l is the left son of v.

Updates are performed as described above. After the insertion or

deletion, we walk back to the root of the deferred tree. Each node we encounter that does not satisfy the balance condition is rebalanced by rotations, as described in [13]. If a node is involved in a rotation that does not "exist", i.e., its *key*-value is undefined, we first expand it. Therefore, the time for rebalancing after one single update can be $\Theta(n)$. However, as was to be expected, future updates—and queries—take advantage from this.

Theorem 11.2.1 *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. By Lemma 2.2.1, there is a constant c such that the root of the deferred tree cannot get out of balance in a sequence of $\leq cn$ updates. (The root was in perfect balance at the moment it was expanded, since we always split the list along the median. Hence, Lemma 2.2.1 can be applied.) 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

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

for some constant c_2.

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

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

The partial dismantling technique: There is another technique to achieve the result of Theorem 11.2.1. It is a generalization of Lueker's partial rebuilding technique. (See Section 2.2.) This generalized technique can also be applied to dynamize other deferred data structures; for example in case the technique that uses rotations does not apply.

We adapt the partial rebuilding technique to deferred data structures. Again the data structure is a deferred BB[α]-tree. Updates are performed as described above. Now, rebalancing is carried out as follows. After the insertion or deletion, we walk back to the root of the deferred tree to find the highest node v that is out of balance. Then we *dismantle* the subtree with root v. That is, we collect all objects that are stored in this subtree, and put them in the list $L(v)$. Furthermore, we set $key(v) :=$ undefined. (The value of $N(v)$ is already equal to $|L(v)|$.) Finally we discard all nodes below v.

Such a dismantling operation takes $O(N(v))$ time. Note that by Lemma 2.2.1, there must have been $\geq (1-2\alpha)N_v - 2$ updates since v was expanded. (Node v was in perfect balance at the moment it was expanded, since we always split the list $L(v)$ along the median. Hence, Lemma 2.2.1 can be applied.)

Let $g(n,k)$ denote the total time to perform a sequence of k member queries, insertions and deletions in a set of initially n objects, using the partial dismantling technique. Then in exactly the same way as in the proof of Theorem 11.2.1, it can be shown that there exist constants c, c_1 and c_2, such that

$$g(n,k) \leq \begin{cases} \max_{0 \leq k_1 \leq k}\{g(n/2,k_1) + g(n/2,k-k_1)\} + c_1 n \text{ if } k \leq cn, \\ c_2 k^2 \text{ if } k \geq cn. \end{cases}$$

It follows that $g(n,k) = O(n \log k + k^2)$, and hence a sequence of $k \leq \sqrt{n}$ queries and updates takes $O(n \log k)$ time.

After $\sqrt{n}$ operations, we have spent already $\Omega(n \log n)$ time. Then, we build—in $O(n \log n)$ time—a binary tree for the objects that are present at this moment, and we proceed with the operations in the standard way.

This gives an alternative proof of Theorem 11.2.1.

A third solution: Finally, we give yet another proof of Theorem 11.2.1. The method is based on the ideas of decomposable searching problems and global rebuilding [42]. Again, this technique can also be applied for other searching problems; for example in case the previous two solutions are not applicable.

We maintain two structures M and I. The main structure M is a static deferred binary search tree in which we store the n objects that are initially present. Each node v in this deferred tree for which the *key*-value is defined, also has a boolean variable $b(v)$, which says whether or not $key(v)$ is present. The structure I is an ordinary—i.e., non-deferred—balanced binary search tree, in which we store all new points. Initially, I is empty.

Suppose we have to insert object x. Then we do a member query in the deferred tree M. If we find x, say in node v, we set $b(v) :=$ **true**. Otherwise, we insert x in I in the standard way.

A deletion of object x is performed as follows. First we do a member query in the deferred tree M. If we find x, say in node v, we set $b(v) :=$ **false**. So we do not delete x, we only "cross it out". If we do not find x in M, we delete it from the tree I in the standard way.

To perform a query x, we first query the deferred tree M. If we find x, say in node v, we infer from $b(v)$ whether or not x is present. If we do not find x, we perform a member query in the tree I.

Suppose we perform a sequence of $k \leq n$ operations in this way. In the tree M we perform k queries. By Theorem 11.1.1, the total time we spend there is bounded by $O(n \log k)$. In the tree I we perform a sequence of at most k queries and updates. Each such operation takes $O(\log k)$ time, since I stores at most k objects. Hence we spend $O(k \log k)$ time in the tree I. It follows that the total time for $k \leq n$ operations is bounded by $O(n \log k + k \log k) = O(n \log k)$. This yields a third proof of Theorem 11.2.1.

11.3 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. By Theorem 9.3.2, this shadow administration can be implemented in $O(n)$ space, such that an update takes $O(\log n)$ computing time, two disk accesses, and $O(1)$ transport time.

Suppose all information in main memory is destroyed. Then we transport the structures from secondary memory to main memory, we make the sorted list up-to-date, and we transport the resulting structure to secondary memory, as described in Subsection 9.3.3. This takes three disk accesses, $O(n)$ transport time and $O(n)$ computing time.

At this moment, main memory contains the objects in sorted order. We immediately proceed with answering queries and performing updates, 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 the preceding sections, we started with an unsorted set of objects.

As an illustration, consider the *dominance counting problem*. Here we are given a set V of n points in the d-dimensional real vector space. For a given query point q in d-dimensional space, we have to report the number of points in V that are dominated by q, i.e., the number of points p in V, such that $p_1 \leq q_1, p_2 \leq q_2, \ldots, p_d \leq q_d$.

It was shown by Bentley [6], that for this problem a (static) data structure exists of size $O(n(\log n)^{d-1})$, that can be built in $O(n(\log n)^{d-1})$

time, and in which dominance queries can be solved in $O((\log n)^d)$ time. By using Lueker's partial rebuilding technique, this structure can be dynamized such that updates can be performed in amortized time $O((\log n)^d)$. (In fact, this structure is almost identical to the range tree of Section 2.3.)

Karp, Motwani and Raghavan showed in [28] that a static deferred version of this structure exists, such that a sequence of $k \leq n$ dominance 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.

It is straightforward to give a dynamic deferred solution for the dominance counting problem. This can be done e.g. by applying the partial dismantling technique to the static structure in [28]. In fact, then the update algorithm for the dynamic deferred structure is almost the same as the one in Section 2.3. One can also apply a dynamization technique for decomposable counting problems that is similar to the third solution of Section 11.2 (see [8, 42]). Because these dynamization techniques are well-known, we leave the details to the reader. The result is expressed in the following theorem.

Theorem 11.3.1 *A sequence of $k \leq n$ dominance counting queries, insertions and deletions in a set of initially n points in d-dimensional space, initially ordered according to one of their coordinates, can be performed in total time $O(n(\log k)^{d-1} + k(\log n)^d)$.*

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

So we have a dynamic deferred data structure for the dominance 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 of points, as described above, in three disk accesses, $O(n)$ transport time and $O(n)$ computing time. Then we immediately proceed with performing operations in the deferred way. Of course, with each update, we also maintain the shadow administration. Note that 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 11.3.1, we can perform $\Theta(n/\log n)$ operations in $O(n(\log n)^{d-1})$ time.

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

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

Chapter 12

Summary and concluding remarks

We have studied the reconstruction problem for dynamic data structures: Given a searching problem, design a dynamic data structure solving this problem, together with a shadow administration from which the data structure can be reconstructed. By storing this shadow administration in secondary memory, we are able to reconstruct the original data structure in case the information in main memory is destroyed.

We have given several techniques that can be used for large classes of searching problems. We give a summary of the most important results.

In Subsection 8.3.1, we have shown that we can maintain an ordered set of n objects in secondary memory, at the cost of 3 disk accesses, $O(\log n + b)$ computing time and $O(b)$ transport time per update. Here, b is the number of objects that can be stored in one block in secondary memory. We applied this technique to a range tree with slack parameter k, which is a data structure of size $O((n \log n)/k)$ that takes $O(n \log n)$ time to build. The result is a shadow administration of size $O(n)$, such that the range tree can be reconstructed at the cost of one disk access, $O(n)$ transport time and $O((n \log n)/k)$ computing time.

In Section 9.1, we have given a general technique for order decomposable set problems. Especially interesting are the $O(n)$-order decomposable set problems. For such problems, the dynamic data structure has size $O(n \log \log n)$ and it can be built in $O(n \log n)$ time. We have given a shadow administration of size $O(n)$, from which the data struc-

ture can be reconstructed in one disk access, $O(n)$ transport time and $O(n \log\log n)$ computing time. For this shadow administration, an update takes one disk access and an amortized amount of $O(n/\log n)$ transport and computing time.

In Section 9.2, we have applied Bentley's logarithmic method to shadow administrations. The result is Theorem 9.2.1:

Suppose we are given a (static) data structure and a corresponding shadow administration for a decomposable searching problem. The complexity of the shadow administration is denoted by $S'(n)$, $P_c(n)$ and $R_c(n)$. The logarithmic method gives a semi-dynamic data structure with a corresponding shadow administration. This shadow administration has size $O(S'(n))$, and the data structure can be reconstructed from it in one disk access, $O(S'(n))$ transport time, and $O(R_c(n))$ computing time. An insertion takes one disk access; an amortized transport time of $O(S'(n)/n)$ if $S'(n)/n^{1+\epsilon}$ is non-decreasing for some $\epsilon > 0$, and $O((S'(n)/n) \times \log n)$ otherwise; and an amortized computing time of $O(P_c(n)/n)$ if $P_c(n)/n^{1+\epsilon}$ is non-decreasing for some $\epsilon > 0$, and $O((P_c(n)/n) \times \log n)$ otherwise.

This result means that a given shadow administration can be dynamized, such that the resulting shadow administration is (asymptotically) of the same size, and has the same reconstruction complexity. This new shadow administration, however, has an easy and efficient insert algorithm.

We have given only one of the many known dynamization techniques. In [56], other techniques are given to design shadow administrations for the data structures solving decomposable searching problems.

In Section 9.3, we have given a general technique to implement any shadow administration in secondary memory. The main result is Theorem 9.3.2: Let *DS* be a dynamic data structure and let *SH* and *INF* be a corresponding shadow administration. Let $U_c(n)$ be the total update computing time of *SH* and *INF*, let $R_c(n)$ be the computing time needed to reconstruct the structures *DS* and *INF* from *SH*, and let $C(n)$ be the amount of data that is changed in an update in *SH*. We have shown that we can implement these structures such that the resulting shadow administration has size $O(S_{SH}(n)+S_{INF}(n))$, and can be updated in two disk accesses, $O(U_c(n))$ computing time and $O(C(n))$ transport time in the worst case. The structures *DS*, *SH* and *INF* can

be reconstructed in three disk accesses, $O(S_{SH}(n))$ transport time and $O(S_{SH}(n) + R_c(n))$ computing time.

So this result gives an efficient implementation of *any* shadow administration. Especially the update algorithm is interesting: it takes only two disk accesses and a small amount of transport time, even in the *worst case*.

In Chapter 10 we have studied the union-find problem. We have designed a structure of size $O(n)$, in which each *UNION* takes $O(k)$ time, and each *FIND* takes $O(\log_k n)$ time. A copy of this data structure can be maintained in secondary memory using $O(n)$ space, such that reconstruction takes one disk access and $O(n)$ transport and computing time. This copy can be maintained after a *UNION* operation at the cost of at most three disk accesses and $O(k)$ transport and computing time. A *FIND* operation does not change the structure and, hence, the copy does not have to be updated after such an operation.

This data structure nicely illustrates how shadow administrations can be implemented. It also shows that the copy in secondary memory is stored in such a way that all pieces of information are mixed up together. For example, the pointers of the main memory structure do not have any meaning in secondary memory. This does not matter, since we only require that the shadow administration contains information from which the original main memory structure can be reconstructed. Note that the structure of Chapter 10 can be implemented on a Pointer Machine. If we store the structure, however, together with a copy in secondary memory, the main memory structure must be implemented on a RAM, since we need to store in secondary memory the addresses of the information in main memory. Furthermore, the structure cannot be applied in the scenario of Part II, where we assumed that the structure does not fit in main memory. The reason is that in this case, the pointers must have a meaning in secondary memory. The maintenance of the correct meaning of these pointers will take a lot of disk accesses.

Finally, in Chapter 11, we have applied known dynamization techniques to static deferred data structures. We have proved in Section 11.2, that a sequence of $k \leq n$ member queries, insertions and deletions in a set of initially n objects can be performed in $O(n \log k)$ time, which is optimal. In fact, we have given three techniques that achieve this result. These techniques can also be applied to dynamize

other static deferred structures. Dynamic deferred data structures lead to another approach to the reconstruction problem. In this approach we maintain in secondary memory the objects represented by the data structure. After a crash, we transport these objects to main memory, and we immediately continue with query answering and performing updates. The data structure is reconstructed "on-the fly" in a deferred way.

We finish this part with some directions for future research.

A first direction is to search for other general solutions, to study other classes of searching problems, and to design other techniques for decomposable searching problems. Also, it would be interesting to have more examples of shadow administrations for specific data structures. For example, in order to apply the general technique of Section 9.3, shadow administrations are needed for which $C(n)$—the amount of data that is changed in an update—is small.

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

A very important problem, that we have not considered at all, is the following *optimization problem*. In the reconstruction problem, we reconstruct the data structure in most cases exactly as it was before the information was destroyed. The optimization problem is to reconstruct the structure in such a way that it is "more balanced" than the destroyed structure was. For example, in case of a range tree, we maintain in secondary memory the points represented by the tree. (See Subsection 8.3.1.) The range tree is reconstructed by building it from these points. Of course, this tree is rebuilt as a perfectly balanced tree. So after reconstruction, the data structure is—in general—more balanced than it was before the information was destroyed. An interesting research direction is to study this optimization problem. Again, general techniques may exist, and special classes of searching problems may admit efficient solutions. An example of a solution to this problem for trie hashing functions is given in Torenvliet and van Emde Boas [61].

Bibliographic comments

The reconstruction problem was first posed by Leen Torenvliet and Peter van Emde Boas in [61]. They suggested to initiate a systematic study for solutions to this problem. The results of Chapters 8 and 9 are based on joint work with Leen Torenvliet, Peter van Emde Boas and Mark Overmars, see [55, 56]. The worst-case solution of Subsection 9.3.3 is new. Chapter 10 is based on [51], and Chapter 11 on [52]. The first technique of Section 11.2 was discovered independently by Yu-Tai Ching and Kurt Mehlhorn, see [16]. Section 11.3 is new.

Part IV

Maintaining dynamic data structures in a network

Chapter 13

The multiple representation problem

13.1 Introduction

In Part III we have studied a special instance of the general problem of maintaining multiple representations of dynamic data structures. In the present part, we consider a related problem, namely the problem of maintaining a number of copies of a data structure in a network of processors.

Assume we have a network of processors, each having its own memory. Each processor holds its own copy of a particular data structure. Changes to the data structure have to be made in all copies. To avoid that each processor spends a lot of time in updating its copy, we dedicate one processor the task of maintaining the data structure and broadcasting the actual changes to the other processors. So we have a multiple representation of the data structure. One data structure that should allow for updates, and a set of other structures that answer queries. Of course, the query data structures must be structured in such a way that they can perform updates, but they get the update in a kind of "preprocessed" form that is easier to handle. The one structure that performs the updates will be called the *central structure*. The other structures that allow for queries are the *client structures*. We study how to organize the central structure for different types of query

problems, how to structure the client structures, and what type of information has to be sent from the central structure to the clients. It will be shown that, after "preprocessing" an update by the central structure, the clients can often perform the update more efficiently. Also, in some situations the client structures can be smaller than the central structure.

This *multiple representation* problem is related to the reconstruction problem. In fact, they are "dual" to each other. In the reconstruction problem there is one memory—main memory—in which queries and updates are performed. After an update in main memory, information is transported to another memory—which is secondary memory. Then in secondary memory, the relevant parts of the structure are updated. In secondary memory, no queries are performed. In the multiple representation problem of the present part, there is one storage medium—the central processor—in which only updates are performed. After this central update, we transport data to other storage media—the clients. Each client then updates its own structure. These client structures are also used for query answering.

An example of a practical instance in which this framework can be applied is a "Star Network". Here the central processor is the main computer; it holds the central data structure, and is connected to all other processors. Often, these other processors, that contain the client structures, are somewhat limited in capacity. Clearly, it is desirable in such situations to utilize the power of the central processor as much as possible.

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

In the next section, we give the general framework we use to describe solutions for this multiple representation problem, and we introduce complexity measures to express the efficiency of solutions. In Section 13.3, we study binary trees as our first example where the client structures store less information than the central structure.

In Chapter 14, we consider general techniques that are applicable

to order decomposable set problems and decomposable searching problems. We also give a technique that applies to any data structure. This general technique is especially efficient for data structures in which an update changes only a small part of the structure.

In Chapter 15 we give a summary of the most interesting results.

13.2 The general framework

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

- There is a network of processors, the *clients*, each having its own memory. Each of these clients contains the same data structure DS—the *client structure*—and uses it to solve queries.
- One of the processors contains a *central structure* DS'.

We assume that all processors are Random Access Machines. Updates have to be performed in all the client structures. Such an update is performed as follows. We first perform the update in the central structure DS'. During this update we (hopefully) obtain information that makes it possible to update the client structures more efficiently than by just directly updating them. Then we send information about the update through the network to the clients, and using this information each client updates its structure DS. We express the complexity of an update of the client structures by the number of words transported to each client, and by the amount of computing time that the client structure needs to perform the update.

Just as in Part III, we have introduced a multiple representation of the data. We have a number of copies of the same data structure DS. Furthermore, there is a data structure DS', that is used to "preprocess" updates, so that the client structures DS can be updated efficiently. On the client structures, queries and preprocessed updates are performed, whereas on the central structure only updates are carried out. We will see that the client structure and the central structure need not be identical. Therefore we use different notations for these structures.

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

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

We assume that $G(n) = \Omega(F(n))$, which is reasonable, since a client receives an amount of $F(n)$ data, and it has to store it somewhere. Note that we express $F(n)$ in terms of words. In Section 13.3, however, we express $F(n)$ in terms of bits. In this chapter, we never use the building time of the client structure DS. Therefore, we do not introduce a notation for it.

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

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

(There is no query time here, because queries are not performed on the central structure.)

The problem investigated in this part of the thesis is the following. We are given a searching problem. The main goal is to design a client structure DS for this searching problem, such that when an update is given in some preprocessed form, this update can be performed efficiently. Ideally, the size of this preprocessed form and the time to

perform the update using this information (i.e., the values of $F(n)$ and $G(n)$) are much smaller than the time needed to perform the update directly in the structure. A second goal is to design a central structure DS' in which the updates can be preprocessed efficiently. We shall emphasize, however, the design of the client structure DS.

13.3 An example: binary search trees

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

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

The update algorithm: Suppose an object p is to be inserted or deleted in the client structures. Then we first insert or delete p in the central structure T'. This gives us a path in T', from the root to an appropriate leaf, along which rotations have been (possibly) carried out. We encode this path by a string $s = (r_1, b_1, r_2, b_2, \ldots, r_k, b_k)$, where k is the length of the path. Starting at the root of the tree, r_1 contains

information whether a left single rotation, a right single rotation, etc. has to be carried out, or that no restructuring operation is necessary; b_1 tells whether the next node on the path lies to the left or to the right of the root; r_2 tells what kind of rotation has to be carried out for the second node of the path, and b_2 says in which direction the path proceeds, and so on. Note that $O(k) = O(\log n)$ bits are sufficient to represent the string s. Now we send to each client structure the object p together with information whether it has to be inserted or deleted, and the string s. Using p and s, the client structures T are updated. Note that we know exactly which path in T we have to walk down, and where on this path restructuring operations have to be carried out. So we do not have to decide in each node—by means of a comparison of p with the value stored in this node—in which direction to proceed. Hence this will save for each client structure $O(\log n)$ comparisons in the update procedure.

The complexity: The complexity of this solution is as follows. The central structure has size $O(n)$, and an update takes $O(\log n)$ time. Each client structure has also size $O(n)$. In this last bound, however, the constant factor will be smaller. Member queries can be solved in the client structures in $O(\log n)$ time. To perform an update, an object p and a bitstring s of length $O(\log n)$ are sent to the client structures, and for each of these structures $O(\log n)$ computing time is needed to update it. Again the constant factor is smaller than in the update time of the central structure.

So at the cost of a slight increase in the amount of data that is transported to the client structures—by sending an additional string of $O(\log n)$ bits—we have decreased the constant factors in the complexity bounds for the client structures, compared to the constants in the bounds of the central structure.

The client structures can be used for solving other searching problems. Examples are the one-dimensional range searching problem, where we are given a range $[a : b]$, and we have to report all objects lying in this range. Such a range query can be answered, without needing balance information at the nodes, in $O(\log n + t)$ time, where t is the number of points in the range. Another example is the one-dimensional

nearest neighbor searching problem. Here we are given an object p, and we have to report the object in the tree that is closest to p. Clearly, such a query can be answered, again without using balance information at the client structures, in $O(\log n)$ time.

Chapter 14

General approaches

14.1 Order decomposable set problems

In Section 2.5, we defined the class of order decomposable set problems. Recall that a set problem $PR : P(T_1) \rightarrow T_2$ is called $M(n)$-order decomposable, if there is an order ORD on T_1, such that for any set $V = \{p_1 < p_2 < \ldots < p_n\}$, ordered according to ORD, and for any i, $1 \leq i < n$, the answer $PR(V)$ can be computed from $PR(\{p_1, \ldots, p_i\})$ and $PR(\{p_{i+1}, \ldots, p_n\})$ in $M(n)$ time.

Let PR be an $M(n)$-order decomposable set problem, and let V be a set of cardinality n for which we want to maintain the answer to PR. In Section 2.5, we gave a dynamic data structure that maintains this answer. The data structure presented there has the property that just a small part of the structure is used for answering a query—the answer to the problem is stored in the root of the tree that contains the elements of V—whereas the rest of the structure is only used to update this answer efficiently.

Therefore, we take for the client structures, the answer $PR(V)$ to the set problem for the entire set V, and we take for the central structure, the fully dynamic data structure. Updates are first performed on the central structure. Then we replace each old client structure by the new answer to the set problem. The result is given in the following theorem. (The notations used are the same as in Section 13.2.)

Theorem 14.1.1 *For an $M(n)$-order decomposable set problem, there*

exists a client structure, that maintains the answer to the set problem, with complexity

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

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

Proof. The proof follows from the above discussion. □

It follows from Theorems 2.5.1 and 14.1.1, that for many values of $M(n)$, the client structures have asymptotically lower complexity than the central structure. For example, for any $O(n)$-order decomposable set problem, the central structure has size $O(n \log\log n)$, whereas the client structures have size only $O(n)$. Examples of such set problems are given in Section 2.5.

14.2 Decomposable searching problems

In this section we consider decomposable searching problems, that were introduced in Section 2.4. Recall that a searching problem $PR : T_1 \times P(T_2) \to T_3$ is called decomposable, if there is a function $\square : T_3 \times T_3 \to T_3$, such that for any partition $V = A \cup B$ of any subset V of T_2, and for any query object x in T_1, we have $PR(x, V) = \square(PR(x, A), PR(x, B))$, where the function $\square$ can be computed in constant time.

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

The multiple representation: To maintain a multiple representation for PR we proceed in the following way. Let the client structure

consist of the structure DS, together with a list of objects. The central structure consists of a copy of the structure DS. In order to avoid confusion, we denote this central structure by DS'.

Initially, the list of objects in the client structure is empty, and the structures DS and DS' are up-to-date. Let n be the initial number of objects.

The insert algorithm: Consider an insertion of an object p. First we insert p in the central structure DS'. If p is already present, then nothing has to be done. (In this case the client structures do not have to know that anything happened.) If p is a new object, we send it to the clients, and each client adds it to its list. After $Q(n)$ objects are inserted in this way—hence each client structure contains a list of $Q(n)$ objects—a copy of the central structure—which is up-to-date—is sent to the clients. Each old client structure is then replaced by this new structure, and the list of objects is initialized again as an empty list. If m is the number of objects that are present after these $Q(n)$ insertions, we repeat this procedure, now with a sequence of $Q(m)$ insertions.

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

Theorem 14.2.1 *Let DS be a data structure for a decomposable searching problem PR, of size $S(n)$ and query time $Q(n)$. There exists a client structure solving PR, with performances:*

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

Proof. The client structure consists of a copy of the data structure DS as it is at the beginning of a sequence of insertions, together with a list

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

The client structure in this theorem is, of course, not very efficient. The given dynamization technique, however, is a first step towards a more powerful technique that will be worked out later in this section.

In the above theorem, the insert complexity for the client structures is an amortized complexity. We show now how these bounds can be turned into worst-case bounds. The idea is to spread out the transportation of the large data structure over a number of insertions. In the sequel we assume that if object p is to be inserted, it is not present yet. (As we saw already, if the object is present, the client structures do not have to know that anything happened.)

The multiple representation: The client structure consists of a data structure DS, and two lists of objects. The central structure consists of a copy of the structure DS—which we denote again by DS'—

and one list of objects.

The insert algorithm: Let k be the initial number of objects. Then the client structure contains an up-to-date data structure DS and the two lists are empty. The central structure contains an up-to-date structure DS' and an empty list.

The initial stage: During $Q(k)$ insertions, we add the new objects to one of the lists of the client structures. (Each time we add it to the same list.) Furthermore, all these insertions are performed in the central structure DS'.

Hence after these $Q(k)$ insertions, the client structure consists of a data structure DS, representing the k objects that were initially present, a list of the $Q(k)$ most recently inserted objects, and an empty list. The central structure consists of an up-to-date structure DS', and an empty list.

Now the periodic process of insertions can start. Let $n = k + Q(k)$, i.e., n is the number of objects that are currently present. Consider a sequence of $Q(n)$ insertions.

Part 1 of the regular stage: During the first $Q(n)/2$ insertions, we add the new objects to the initially empty lists of the client structures, and we send the central structure DS' to the clients: Each update we send a part of DS' of size $O(S(n)/Q(n))$. Then, after these $Q(n)/2$ insertions, each client structure contains a data structure DS', and a list of the $Q(n)/2$ most recently inserted objects. Now we replace the old client structure DS by the structure DS', and we set the old list of $Q(k)$ inserted objects to the empty list. (We denote the new client structure again by DS.) In the central structure we add the $Q(n)/2$ new objects to the list. Note that the central structure DS' cannot be affected during these insertions.

Part 2 of the regular stage: The final $Q(n)/2$ insertions are performed as follows. The new objects are added to the non-empty list of the client structure. In the central structure, we perform in each update the current one, and one update from the list of updates. (Note that the order in which we perform the updates in the central structure does not matter, since all updates are insertions. If, however, deletions were also possible, the updates had to be carried out in chronological order. See Subsection 14.3.4.) Afterwards the list of the central structure is

set to the empty list.

After Part 2: After the entire sequence of $Q(n)$ updates, the client structure contains a data structure DS storing the n objects that were present $Q(n)$ insertions ago, a list of the $Q(n)$ most recently inserted objects, and an empty list. The central structure consists of an up-to-date structure DS' and an empty list. Hence we are in the same situation as $Q(n)$ updates ago, and we can continue in a similar manner.

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

Theorem 14.2.2 *Let DS be a data structure for a decomposable searching problem PR with worst-case complexity $S(n)$, $I(n)$ and $Q(n)$. There exists a client structure solving PR, with performances:*

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

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

Proof. It follows from the above discussion that in each insertion we send an amount of $O(S(n)/Q(n)) + O(1) = O(S(n)/Q(n))$ data, and for each client structure we have to spend $O(S(n)/Q(n)) + O(1) = O(S(n)/Q(n))$ time to receive and write this data. Hence both $F(n)$ and $G(n)$ are bounded by $O(S(n)/Q(n))$ in the worst case. Also, the size and the query time of the client structure are bounded by $O(S(n))$ and $O(Q(n))$. Clearly, the performances for the central structure are increased by at most a constant factor. $\square$

There are more powerful techniques to get efficient solutions for decomposable searching problems. We can for example consider sequences of more than $Q(n)$ insertions. Then the most recently inserted

objects are stored in a small data structure, to ensure that the query time remains bounded by $O(Q(n))$. In this way the values of $F(n)$ and $G(n)$ can be decreased. This idea is worked out below.

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

The multiple representation: Let $f(n)$ be an integer function, such that $Q(n) < f(n) < n$. The client structure consists of two data structures DS_1 and DS_2, and a list of objects. The central structure contains copies of the structures DS_1 and DS_2, which we denote by DS'_1 and DS'_2.

Initially, the structures DS_1 and DS'_1, and the lists in the client structures, are empty. The structures DS_2 and DS'_2 store the n objects that are present at this moment.

The insert algorithm: Consider a sequence of $f(n)-1$ insertions. We insert the new objects in the central data structure DS'_1. In the client structure we add the new objects to the list. Every $Q(n)$-th insertion, the central structure DS'_1 as it is that moment is sent to the client structure, where it replaces the structure DS_1 (of course we denote this new structure by DS_1), and the list of objects is set to the empty list. Hence during these $f(n)-1$ insertions, the client structure consists of a list of at most $Q(n)$ objects, and of two data structures DS_1 and DS_2, where the structure DS_1 represents at most $f(n)$ objects. At each moment, the objects represented by these three structures form a partition of all the objects that are present at that moment.

In the $f(n)$-th insertion, we build a new structure DS'_2 storing all objects that are present at this moment, and send it to the clients, where it becomes the new DS_2. Also, the structures DS_1, DS'_1 and all lists are made empty. If m is the number of present objects at this moment we repeat this procedure, now with a sequence of $f(m)$ insertions.

It is easy to see that, using this technique, the size and the query time of the client structure remain bounded by $O(S(n))$ and $O(Q(n))$.

Furthermore, the amortized values of $F(n)$ and $G(n)$ are both bounded by $O(S(f(n))/Q(n) + S(n)/f(n))$.

We generalize this solution as follows.

The generalized multiple representation: Let k be a positive integer, and let $f_i(n)$ be integer functions, for $i = 0, 1, \ldots, k$, such that $Q(n) = f_0(n) < f_1(n) < f_2(n) < \ldots < f_{k-1}(n) < f_k(n) = n$. Then the client structure contains a collection of data structures DS_i, $i = 1, 2, \ldots, k$, and a list of at most $Q(n)$ objects. The central structure contains copies of the structures DS_i, which are denoted by DS'_i, $i = 1, 2, \ldots, k$. Each DS_i and each DS'_i will represent at most $f_i(n)$ objects. Initially, all structures $DS_1, \ldots, DS_{k-1}, DS'_1, \ldots, DS'_{k-1}$ and all lists are empty. The structures DS_k and DS'_k store the n objects that are present at this moment.

The insert algorithm: Consider a sequence of $f_{k-1}(n)$ insertions. In the j-th insertion, do the following. If there is an i, $0 \leq i \leq k-1$, such that $j \equiv 0 \bmod f_i(n)$, determine the maximal such i. Then build a new structure DS'_{i+1}, storing all objects that were present in the old central structures $DS'_1, \ldots, DS'_{i+1}$, and add it to the central structure. Also, the old central structures $DS'_1, \ldots, DS'_i$ are made empty. Next, send this new structure DS'_{i+1} to the clients, where it replaces the old DS_{i+1}. (We denote this new client structure again by DS_{i+1}.) Finally, all client structures $DS_1, \ldots, DS_i$ and the lists are made empty. If there is no i such that $j \equiv 0 \bmod f_i(n)$, add the new object to the list of the client structures, and insert the new object in the central structure DS'_1.

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

After these $f_{k-1}(n)$ insertions, all structures $DS_1, \ldots, DS_{k-1}, DS'_1, \ldots, DS'_{k-1}$ and all lists, are empty again, and the structures DS_k and DS'_k store the objects that are present at this moment. (Note that in the $f_{k-1}(n)$-th insertion, the maximal value of i in the above update procedure is $k-1$.) So we can proceed in the same way, now with a sequence of $f_{k-1}(m)$ insertions, where m is the current number of objects.

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

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

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

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

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

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

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

Theorem 14.2.3 *Let DS be a data structure for a decomposable searching problem PR with complexity $S(n)$ and $Q(n)$. Then for each positive integer k there exists a client structure solving PR, with performances:*

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

We illustrate this result with an example. In the nearest neighbor searching problem, we are given a set V of n points in the plane, and a query point p, and we are asked to find a point in V that is closest to p

with respect to the euclidean distance. Clearly, this problem is decomposable. There exists a data structure for this problem of size $O(n)$ such that queries can be solved in $O(\log n)$ time, see e.g. Kirkpatrick [29].

If we apply for example the logarithmic method to this structure, see Section 2.4, we get a semi-dynamic structure of size $O(n)$, having a query-time of $O((\log n)^2)$, in which points can be inserted in amortized $O((\log n)^2)$ time.

Applying Theorem 14.2.3 to Kirkpatrick's structure, however, we obtain:

Theorem 14.2.4 *Let k be a positive integer. For the nearest neighbor searching problem in the plane, there exists a client structure, with performances:*

1. *The size of the client structure is bounded by $O(n)$.*
2. *$F(n) = O(k \times (n/\log n)^{1/k})$ in the worst case, for an insertion.*
3. *$G(n) = O(k \times (n/\log n)^{1/k})$ in the worst case, for an insertion.*
4. *The query time of the client structure is bounded by $O(k \times \log n)$.*

It is clear that the techniques presented in this section only allow insertions to be carried out. In some cases, however, deletions are also possible. For example, deletions can be handled if we restrict ourselves to a subclass of the decomposable searching problems, the *decomposable counting problems*. See Section 2.4 for the definition and for a sketch of a dynamic data structure solving these problems.

For these decomposable counting problems, the following analogue of Theorem 14.2.3 can be proved.

Theorem 14.2.5 *Let DS be a data structure for a decomposable counting problem PR with complexity $S(n)$ and $Q(n)$. Then for each positive integer k there exists a fully dynamic client structure solving PR, with performances:*

1. *The size of the client structure is bounded by $O(S(n))$.*
2. *$F(n) = O(k \times (S(n)/n) \times (n/Q(n))^{1/k})$ in the worst case.*
3. *$G(n) = O(k \times (S(n)/n) \times (n/Q(n))^{1/k})$ in the worst case.*
4. *The query time of the client structure is bounded by $O(k \times Q(n))$.*

14.3 A general technique

14.3.1 Introduction

We present a general technique that is similar to the general technique of Section 9.3.

Consider again our strategy with respect to the member searching problem of Section 13.3. In this solution, in each update we send a string of $O(\log n)$ bits to the client structures, where the string contains an encoding of the path to the node where the update is carried out, together with information about what kind of rotations have to be performed. In order to update the client structure, we follow the path, insert or delete the object, and perform the rotations. Clearly, this procedure takes $O(\log n)$ time. If we consider, however, how many nodes in the tree are changed in this update, we see that $O(1)$ of them are changed due to the insertion or deletion, and the rest of them are changed due to rotations. Therefore, if $O(1)$ rotations are carried out, only $O(1)$ nodes of the tree are changed. (Note that a client structure does not contain balance information.) So if we could avoid to walk down the path, it could be possible to update the client structure in only $O(1)$ time.

The solution is to send to the client structures the inserted or deleted object, together with the positions in the tree where changes—and what kind of changes—have to be carried out. Since there are binary trees that can be maintained in logarithmic time with only $O(1)$ rotations in the worst case (see Theorem 2.2.3), this will give us a solution where the client structures can be maintained in constant time.

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

Let PR be a searching problem, and let DS resp. DS' be the corresponding client structure resp. central structure. The performances of DS are denoted by $S(n)$ and $Q(n)$, and those of DS' by $S'(n)$, $P'(n)$

and $U'(n)$ (see Section 13.2 for these notations). We assume that DS is a *substructure* of DS'. That is, DS is a part of DS', containing enough information such that queries can be solved fast. For example, if DS' is a balanced binary tree, then we can take DS, the tree without the balance information at the nodes. Updates are performed as before. That is, first the central structure DS' is updated, then information is sent to the client structures, and finally the client structures DS are updated. Let $C(n)$ denote the amount of data that is changed in the client structure DS in an update. We assume that all these complexity measures are non-decreasing and smooth.

We transform this multiple representation into another one, such that each transformed client structure has size $O(S(n))$, update complexity $F(n) = O(C(n))$ and $G(n) = O(C(n))$, and in which queries can be solved in $O(Q(n))$ time. In each update, we only send the changes of the client structure DS. In order to avoid searching for the positions in the client structure where the changes have to be carried out, we also send these positions. Therefore, we implement the data structures as arrays. (The processors are Random Access Machines, the memories of which are modeled as arrays. Hence we can indeed implement the data structures as arrays.) We take care that each part of DS is stored in the same position in all processors. If such a part has to be changed, we send the index in the array where this part is stored, together with the updated part. Then, in each client structure, we can find in constant time the position where the change has to be carried out. Note that data structures contain pointers, which we consider to be indices of array entries. By storing parts of DS in each processor in the same positions, these pointers indeed "point" to the correct objects.

The implementation will be described more precisely in the next subsection. We finish this subsection with the following lemma.

Lemma 14.3.1 *The complexity measures introduced above satisfy:*

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

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

14.3.2 A fixed size solution

Let N be an integer that denotes the maximal number of objects that can be represented by our data structures. We use in this subsection—and in the following ones—the notations introduced in Subsection 14.3.1.

We have a client structure DS and a central structure DS', and we want to implement these structures as arrays. These data structures are composed of "indivisible pieces of information" of constant size, such as pointers, integers, etc. Each such indivisible piece will be stored in one array location. Since the data structures represent at most N objects, we take a *client array* A of $S(N)$ entries, containing DS, and a *central array* A' of $S'(N)$ entries, containing DS'. If n is the current number of objects, $S(n)$ entries of the client array and $S'(n)$ entries of the central array are occupied. We assume that the first $S(N)$ entries of the central array are identical to those of the client array. (If we assume that our data structures only contain fixed size records, this can always be achieved. Otherwise, if variable sized records are allowed, we can split these into fixed size records, and apply the techniques developed here.) Finally, we introduce two stacks FE and FE' of free entries. In FE we store those indices of the first $S(N)$ entries of the client array A, that are unoccupied. Similarly, the stack FE' contains those indices of the last $S'(N) - S(N)$ entries of the central array A' that are unoccupied. The purpose of these stacks is to perform our own memory management. Note that by maintaining the client array in the first $S(N)$ entries of the central array A', we guarantee that the client array A can be stored in $S(N)$ consecutive memory locations in

a client processor. Hence the amount of space used by the client array is $S(N)$. (If we did not store the information in consecutive locations, there would have been gaps in the client's memory.)

The multiple representation: The transformed client structure consists of the array A. The transformed central structure consists of the array A' and the stacks FE and FE'.

The update algorithm: Suppose we want to insert or delete an object. We assume that there is space in the arrays for a new object. Then we first perform this update in the central structure. If we need new entries, we take them from the appropriate stack FE or FE', and if entries become unoccupied, we put them on the stack where they belong. Next we send to the clients, the indices of the entries in the array A that are changed together with the new contents of these entries. Using this information, each client structure is updated.

Note that the client structures do not need to contain the stack FE of free array indices: The memory management of all processors is arranged by the central structure. Hence we utilize the central processor as much as possible.

At each moment the client structure is up-to-date and, hence, it can be used to answer queries.

Theorem 14.3.1 *Let DS be a client structure solving some searching problem, with complexity $S(n)$, $Q(n)$ and $C(n)$. Let DS' be the corresponding central structure, with complexity $S'(n)$ and $U'(n)$. We can transform these structures into a multiple representation, such that each client structure*

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

Here N is the maximal number of objects that can be represented by the structures, and n is the current number of objects. Furthermore, the

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

Proof. The size of the central structure is bounded by $O(S'(N))$ for the array A', and by $O(|FE|+|FE'|) = O(S'(N))$ for the stacks. Hence the total size of the central structure is bounded by $O(S'(N))$. It is clear that the update of the central structure takes $O(U'(n))$ time. The client array can be updated in time proportional to the number of changed entries. So in our notation we have $F(n) = O(C(n))$ and $G(n) = O(C(n))$. The other bounds follow from the above discussion. □

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

14.3.3 An amortized solution

Suppose that the data structures initially represent n objects. We store each structure in an array that can contain a data structure for $3n/2$ objects. In this way there is space in the structures for $n/2$ insertions. So in the notation of the preceding subsection, we take $N = 3n/2$. The client structure consists of the array A of length $S(N)$. The central structure contains the array A' of length $S'(N)$, and the stacks FE and FE'. The information is stored in these data structures as in the previous subsection, and updates are performed in exactly the same way. As soon as the number of objects becomes either $n/2$ or $3n/2$, we rebuild our data structures. That is, if m is the number of objects at that moment, we build a new array A' and new stacks FE and FE', that are large enough to contain a data structure for $3m/2$ objects, and we send the subarray containing the first $S(3m/2)$ entries of A'—this subarray will be the new client structure A—to the clients, where this new array replaces the old one. Then we proceed in the same way.

Theorem 14.3.2 *Let DS be a client structure solving some searching problem, with complexity $S(n)$, $Q(n)$ and $C(n)$. Let DS' be the corresponding central structure, with complexity $S'(n)$ and $U'(n)$. We can transform these structures into a multiple representation, such that each client structure*

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

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

Proof. The bounds on the amount of space used by the structures follow from Theorem 14.3.1, and from the fact that N—the maximal number of objects that can be represented—and n—the current number of objects—satisfy $n = \Theta(N)$. Clearly, the query time for a client structure remains $O(Q(n))$. Since the structures are rebuilt at most once every $n/2$ updates, the amortized values of both $F(n)$ and $G(n)$ are bounded by $O(C(n)+S(n)/n)$, which is $O(C(n))$ by Lemma 14.3.1. Rebuilding of the new central structure takes $O(P'(n))$ time for A' and $O(S'(n))$ time for the two stacks. So the amortized update time of the central structure is bounded by $O(U'(n) + P'(n)/n + S'(n)/n)$, which is $O(U'(n))$ by Lemma 14.3.1. □

Remark. The rebuilding of the new central array A' *cannot* be performed by just walking along the old array and putting the entries into a new one of size $S'(3m/2)$: We have to take care that the pointers keep their correct meaning. Therefore we charged in the above proof $O(P'(n))$ time for this rebuilding, which is clearly an upper bound.

14.3.4 A worst-case solution

In this subsection we assume that the update time $U'(n)$ of the central structure and the amount of data $C(n)$ that an update changes in the

client structure are worst-case bounds. We show how the amortized bounds of the preceding section can be made into worst-case bounds. The idea is to spread out the construction of the new structures over a number of updates. The technique is related to the global rebuilding technique given in Overmars [42]. See also Subsection 9.3.3.

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

The update algorithm: Consider a sequence of $m/2$ updates. (Note that the array A has space for at least $m/2$ new objects.) We split this sequence into 3 stages.

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

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

Second stage: The second stage consists of the next $m/8$ updates. These updates are performed as in the first stage. Also, a new client array A_0 is built in the central computer during the first $m/16$ updates of this second stage. This array has length $S(l_0)$, and it stores the client data structure as it was after the first stage. (Later we shall describe how the central processor builds this new array; we now just assume that it is there.) This new array is sent to the clients during the last $m/16$ updates of the second stage. In each update we send an amount of $O(S(l_0)/m) = O(S(m)/m)$, which is bounded by $O(C(m))$ by Lemma 14.3.1.

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

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

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

After this third stage, the client array A_0 is up-to-date, and the old array A is discarded.

So we end with a client structure consisting of an array A_0 of length $S(l_0)$. Let n be the number of objects that are represented by the structures at this moment. If we can show that $3n/2 \leq l_0 \leq 3n$, then we are in the same situation as the one we started with, and hence we can proceed in the same way.

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

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

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

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

Clearly, m and n are related by

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

It follows that

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

and

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

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

The central structure: The central structure consists of two copies of each of the structures A', FE and FE', and one copy of a list L (we use the notations of the preceding subsection). All $m/2$ updates are carried out on one of A', FE and FE'. Hence at each moment the central structure contains an up-to-date data structure. In the second stage, in each update we add the object together with information whether it has to be inserted or deleted, to the list L.

It remains to describe what happens to the other structures A', FE and FE'. In the first stage, the updates are performed on these structures as usual. During the first $m/16$ updates of the second stage we convert them into new structures A'_0, FE_0 and FE'_0. Here A'_0 is an array of length $S'(l_0)$ that will contain the data structure as it is at the beginning of the second stage, and FE_0 and FE'_0 are the corresponding stacks of free entries in this new array. This converting can be performed in $O(P'(l_0) + S'(l_0)) = O(P'(m) + S'(m)) = O(P'(m))$ time. In each of the $m/16$ updates we do an amount of $O(P'(m)/m)$ of this converting. It follows from Lemma 14.3.1 that the update time for the central structure remains $O(U'(n) + P'(m)/m) = O(U'(n))$, where n is the current number of present objects.

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

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

queries.

Theorem 14.3.3 *Let DS be a client structure solving some searching problem, with worst-case complexity $S(n)$, $Q(n)$ and $C(n)$. Let DS' be the corresponding central structure, with worst-case complexity $S'(n)$ and $U'(n)$. We can transform these structures into a multiple representation, such that each client structure*

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

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

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

We have proved that we can bound the update time for the client structures by $O(C(n))$, which is the size of the changes in the structure. Hence our goal is to design structures for searching problems for which $C(n)$ is small. It is not important whether the changes can be found efficiently (although this would make the amount of work on the central structure small). In the next section, we give two examples of such structures.

14.4 Examples

14.4.1 Binary search trees

Most classes of balanced binary search trees, such as AVL-trees, BB[α]-trees, etc., have the property that in an update $\Omega(\log n)$ rotations might be necessary to rebalance them. Hence for such trees, an update can change $\Omega(\log n)$ nodes. In Section 2.2, however, we defined Olivié's

class of αBB-trees. These trees have the interesting property that they can be maintained in logarithmic time, by at most a constant number of rotations, if $\alpha \in \{1/2, 1/3\}$. See Theorem 2.2.3.

So let T be an αBB-tree, where $\alpha \in \{1/2, 1/3\}$, without the balance information at the nodes. Suppose T contains a set of n objects in its nodes. In this tree, member queries can be solved in $O(\log n)$ time. By the above mentioned result of Olivié, we can maintain T by means of $O(1)$ rotations. Hence an update changes only $O(1)$ nodes in T. (Note that if the tree would contain balance information, an update would change $\Omega(\log n)$ nodes, since then the balance information would have to be updated.) Applying Theorem 14.3.3, we get:

Theorem 14.4.1 *For solving the member searching problem, there exists a client structure with complexity:*

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

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

In the solution given above, we stored the objects in the nodes of the tree. We have seen applications, however, in which we store the objects in sorted order in the leaves. Then, in order to be able to search in the tree, we have to store information in the internal nodes to guide these searches. (In each node we must decide in some way whether we proceed to the left or to the right son.) Suppose we store in each node the maximal element in its subtree. Clearly, we can use this information to solve member queries in time proportional to the height of the tree. If we now delete the maximal element in the tree, then in each node on the rightmost path, the search information has to be changed. Therefore, if the tree is balanced, an update changes $\Theta(\log n)$ nodes. So we have to be careful regarding the "search information" that is stored in the internal nodes.

Suppose now that we store in each internal node v, the maximal element in the left subtree of v. Note that this maximal element is stored in the unique leaf that is reached by making one step to the left in node v, followed by a maximal number (possibly none) of steps to the right. It is not difficult to prove that in this case an update changes $O(1)$ nodes, if we do not rebalance the tree: The search information in a node is changed iff the maximal element in its left subtree is changed. (Note that this is an interesting result on its own. In fact, I have not seen this observation anywhere in the literature.)

So let T be an αBB-tree, containing a set of n elements in sorted order in its leaves, without balance information. Each internal node contains the maximal element in its left subtree. Then, in T member queries can be solved using the search information of the internal nodes in $O(\log n)$ time. Now let $\alpha \in \{1/2, 1/3\}$. Then it follows from the above that an update changes only $O(1)$ nodes in T. Applying Theorem 14.3.3, we obtain:

Theorem 14.4.2 *For solving the member searching problem, we can take for the client structures a leaf search tree, having complexity:*

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

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

14.4.2 Range trees

The orthogonal range searching problem, and an efficient data structure solving this problem—the range tree—were studied already extensively in this thesis. In the following definition we modify the balance conditions of these range trees somewhat.

Definition 14.4.1 Let V be a set of points in the d-dimensional euclidean space. A *d-dimensional (α, α')-range tree* T, representing the set V, is defined as follows.

1. If $d = 1$, then T is an αBB-tree, containing the points of V in sorted order in its leaves.

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

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

The update algorithm: The update algorithm for these (α, α')-range trees is similar to the one in Section 2.3. In fact, only rebalancing is done in a different way. Consider a d-dimensional (α, α')-range tree, and suppose we want to insert or delete a point p. 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 update algorithm for αBB-trees using rotations; otherwise we use the same procedure recursively.) Next we insert or delete p among the leaves in the main tree, and we walk back to the root. During this walk, we rebalance the main tree: Each node that is out of balance is rebalanced by means of rotations. Note that we have to rebuild the associated structures of the nodes that are involved in these rotations, and this will take a lot of time when these structures are large. It turns out, however, that the amortized update time is low.

The following theorem gives the complexity of (α, α')-range trees. For a proof for the bound on the amortized update time, see Willard

and Lueker [66]. The other bounds follow in the same way as in Theorem 2.3.1.

Theorem 14.4.3 *A d-dimensional (α, α')-range tree, representing n points, has size $O(n(\log n)^{d-1})$, and can be built in $O(n(\log n)^{d-1})$ time. In this tree, updates can be performed in amortized time $O((\log n)^d)$, and orthogonal range queries can be solved in $O((\log n)^d + t)$ time, where t is the number of reported answers, without using the balance information stored at the nodes.*

Consider a d-dimensional (α, α')-range tree for a set of n points, without the balance information. We store in internal nodes of the trees search information as in Subsection 14.4.1. We take for the one-dimensional structures αBB-trees with $\alpha \in \{1/2, 1/3\}$. Let $C(n, d)$ denote the amortized number of nodes that an update changes in the range tree.

Lemma 14.4.1 *For a proper choice of α', we have $C(n, d) = O((\log n)^{d-1})$.*

Proof. We have seen in Subsection 14.4.1 already that $C(n, 1) = O(1)$. Let $d > 1$. To perform an update we start in the root of the main tree, and we update its associated structure. This changes, amortized, at most $C(n, d-1)$ nodes. Then we repeat the same procedure for the appropriate son of the root, which is the root of a range tree for at most $(1-\alpha')n$ points. Hence this changes, amortized, at most $C((1-\alpha')n, d)$ nodes. If the root of the main tree gets out of balance, we perform a rotation and, hence, we have to rebuild the associated structures of the sons of the root. Since these associated structures are $(d-1)$-dimensional (α, α')-range trees, this changes $O(n(\log n)^{d-2})$ nodes. It was shown by Blum and Mehlhorn [13] that for a proper choice of α' the root of the main tree gets out of balance at most once every $\Omega(n)$ updates. Hence the amortized number of nodes that are changed due to our visit to the root of the main tree is bounded by $O((\log n)^{d-2})$. It follows that $C(n, d)$ satisfies the following recurrence:

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

This proves the lemma. □

So we have a class of range trees that can be maintained in amortized time $O((\log n)^d)$, whereas in the structures without balance information an update changes, amortized, only $O((\log n)^{d-1})$ nodes. Applying Theorem 14.3.2 leads to:

Theorem 14.4.4 *For solving the orthogonal range searching problem, there exists a client structure with complexity:*

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

Chapter 15

Summary and concluding remarks

We have studied the problem of maintaining a number of copies of a dynamic data structure in a network of processors. In order to avoid that each processor spends a lot of time in updating its copy, we first "preprocess" the update in a central structure. Then we broadcast information about the update to the processors, and, using this information, each of these processors updates its structure.

The most interesting results are as follows.

For each order decomposable set problem PR, there exists a client structure of size $O(PR(n))$, that can be maintained at the cost of $O(PR(n))$ transport and computing time. Here, $PR(n)$ is the size of the answer for a set of n objects. The maintenance of the answer is completely arranged by the central structure. See Section 14.1.

In Section 14.2 we have given techniques for decomposable searching problems. The most general result is Theorem 14.2.3: Suppose we are given a data structure for a decomposable searching problem PR of size $S(n)$ and query time $Q(n)$. Then for each positive integer k there is a client structure solving PR, of size $O(S(n))$ having a query time of $O(k \times Q(n))$. Insertions into this client structure can be performed at the cost of $O(k \times (S(n)/n) \times (n/Q(n))^{1/k})$ transport and computing time.

So this technique gives a client structure of the same size as the structure we started with, having asymptotically the same query time

(if k is a constant). This new client structure, however, has a fast insert algorithm.

In Section 14.3 we have given a technique that applies to any data structure: Suppose we have a client structure DS of size $S(n)$, having a query time $Q(n)$. Let $C(n)$ be the amount of data that an update changes in DS. Then we can transform this structure into another client structure of size $O(S(n))$, having a query time $O(Q(n))$, that can be updated in $O(C(n))$ transport and computing time. See Theorem 14.3.3. An interesting application of this result is given in Subsection 14.4.2. Here, we show that we can maintain a class of d-dimensional range trees, such that the central structure needs $O((\log n)^d)$ time for an update, whereas the client structure can be updated in $O((\log n)^{d-1})$ transport and computing time.

Note that most of the techniques of this part share ideas with those given for the reconstruction problem. Especially the general techniques of Sections 9.3 and 14.3 are strongly related to each other.

There remain several interesting directions for future research.

Just as in case of the reconstruction problem, other general solutions may be possible. Other classes of searching problems can be studied, and other techniques for decomposable searching problems can be designed. Again, more examples are needed of specific data structures. For example, (client versions of) data structures are wanted for which $C(n)$—the amount of data that is changed in an update—is small.

One can also investigate the idea of performing sets of updates, instead of performing each update separately. Special classes of searching problems and general techniques may exist.

Finally, a very general research direction is to study other multiple representation problems. For example, what should be done if the client structures do not necessarily have to represent the same set of objects? We have seen that the problems of Parts III and IV are related, and that many solutions to these problems are based on the same ideas. So one could investigate multiple representations in a more general setting.

Bibliographic comments

The problem of maintaining dynamic data structures in a network was posed to the author in 1986 by Mark Overmars. The results of this part are based on joint work with Mark Overmars, Leen Torenvliet and Peter van Emde Boas, see [54, 55].

Bibliography

[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] A.V. Aho, J.E. Hopcroft and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, 1983.

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

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

[6] J.L. Bentley. *Multidimensional divide and conquer.* Comm. of the ACM **23** (1980), pp. 214-229.

[7] J.L. Bentley and J.R. Friedman. *Data structures for range searching.* Computing Surveys **11** (1979), pp. 397-409.

[8] J.L. Bentley and J.B. Saxe. *Decomposable searching problems I: static to dynamic transformations.* J. of Algorithms **1** (1980), pp. 301-358.

[9] M. Bezem and J. van Leeuwen. *On estimating the complexity of logarithmic decompositions.* Inform. Proc. Lett. **26** (1987/88), pp. 321-324.

[10] G. Blankenagel and R.H. Güting. *XP-trees: externe priority search trees.* Forschungsbericht 260, Abteilung Informatik, Universität Dortmund, 1988.

[11] 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.

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

[13] N. Blum and K. Mehlhorn. *On the average number of rebalancing operations in weight-balanced trees.* Theor. Comp. Sci. **11** (1980), pp. 303-320.

[14] K.Q. Brown. *Geometric transforms for fast geometric algorithms.* Techn. Report CMU-CS-80-101, Department of Computer Science, Carnegie-Mellon University, 1980.

[15] B. Chazelle. *A functional approach to data structures and its use in multidimensional searching.* SIAM J. Comput. **17** (1988), pp. 427-462.

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

[17] D. Comer. *The ubiquitous B-tree.* Computing Surveys **11** (1979), pp. 121-137.

[18] H. Edelsbrunner. *A note on dynamic range searching.* Bull. of the EATCS **15** (1981), pp. 34-40.

[19] H. Edelsbrunner, M.H. Overmars and D. Wood. *Graphics in flatland: a case study.* In: F.P. Preparata (ed.), Advances in Computing Research, Vol. 1, Computational Geometry, J.A.I. Press, London, 1983, pp. 35-59.

[20] I.G. Gowda. *Dynamic problems in computational geometry.* M.Sc. Thesis, Department of Computer Science, University of British Columbia, 1980.

[21] I.G. Gowda and D.G. Kirkpatrick. *Exploiting linear merging and extra storage in the maintenance of fully dynamic geometric data structures.* Proc. 19-th Annual Allerton Conf. on Communication, Control and Computing, 1980, pp. 1-10.

[22] R.L. Graham, D.E. Knuth and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, 1989.

[23] L.J. Guibas and R. Sedgewick. *A dichromatic framework for balanced trees.* Proc. 19-th Annual IEEE Symp. on Foundations of Computer Science, 1978, pp. 8-21.

[24] P.H. Hartel, M.H.M. Smid, L. Torenvliet and W.G. Vree. *A parallel functional implementation of range queries.* ITLI Prepublication Series CT-89-05, Departments of Mathematics and Computer Science, University of Amsterdam, 1989. To appear in: Proc. Computing Science in the Netherlands, 1989.

[25] K. Hinrichs. *The grid file system: implementation and case studies of applications.* Ph.D. Thesis, ETH Zürich, 1985.

[26] K. Hinrichs. *Implementation of the grid file: design concepts and experience.* BIT **25** (1985), pp. 569-592.

[27] Ch. Icking, R. Klein and Th. Ottmann. *Priority search trees in secondary memory.* Proc. WG'87, Lecture Notes in Computer Science, Vol. 314, Springer-Verlag, Berlin, 1988, pp. 84-93.

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

[29] D.G. Kirkpatrick. *Optimal search in planar subdivisions.* SIAM J. Comput. **12** (1983), pp. 28-35.

[30] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[31] 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.

[32] K. Mehlhorn. *Data Structures and Algorithms, Volume 1: Sorting and Searching.* Springer-Verlag, Berlin, 1984.

[33] K. Mehlhorn. *Data Structures and Algorithms, Volume 3: Multi-Dimensional Searching and Computational Geometry.* Springer-Verlag, Berlin, 1984.

[34] L. Monier. *Combinatorial solutions of multidimensional divide-and-conquer recurrences.* J. of Algorithms **1** (1980), pp. 60-74.

[35] R. Motwani and P. Raghavan. *Deferred data structuring: query-driven preprocessing for geometric search problems.* Proc. 2-nd Annual ACM Symp. on Computational Geometry, 1986, pp. 303-312.

[36] J. Nievergelt, H. Hinterberger and K.C. Sevcik. *The grid file: an adaptable, symmetric multikey file structure.* ACM Trans. Database Systems **9** (1984), pp. 38-71.

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

[38] H.J. Olivié. *A study of balanced binary trees and balanced one-two trees.* Ph.D. Thesis, Department of Mathematics, University of Antwerp, 1980.

[39] H.J. Olivié. *On α-balanced binary search trees.* Proc. 5-th GI-Conference, Lecture Notes in Computer Science, Vol. 104, Springer-Verlag, Berlin, 1981, pp. 98-108.

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

[41] M.H. Overmars. *Dynamization of order decomposable set problems.* J. of Algorithms **2** (1981), pp. 245-260.

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

[43] M.H. Overmars. *Efficient data structures for range searching on a grid.* J. of Algorithms **9** (1988), pp. 254-275.

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

[45] M.H. Overmars, M.H.M. Smid, M.T. de Berg and M.J. van Kreveld. *Maintaining range trees in secondary memory, part I: partitions.* Report FVI-87-14, Department of Computer Science, University of Amsterdam, 1987. To appear in: Acta Informatica.

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

[47] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, 1985.

[48] J. Riordan. *Combinatorial Identities.* John Wiley & Sons, New York, 1968.

[49] A.M. Schönhage, M. Paterson and N. Pippenger. *Finding the median.* J. Comput. System Sci. **13** (1976), pp. 184-199.

[50] M.H.M. Smid. *General lower bounds for the partitioning of range trees.* ITLI Prepublication Series CT-88-02, Departments of Mathematics and Computer Science, University of Amsterdam, 1988.

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

[52] M.H.M. Smid. *Dynamic deferred data structures.* ITLI Prepublication Series CT-89-01, Departments of Mathematics and Computer Science, University of Amsterdam, 1989. To appear in: Inform. Proc. Lett.

[53] M.H.M. Smid and M.H. Overmars. *Maintaining range trees in secondary memory, part II: lower bounds.* Report FVI-87-15, Department of Computer Science, University of Amsterdam, 1987. To appear in: Acta Informatica.

[54] M.H.M. Smid, M.H. Overmars, L. Torenvliet and P. van Emde Boas. *Maintaining multiple representations of dynamic data structures.* ITLI Prepublication Series CT-88-03, Departments of Mathematics and Computer Science, University of Amsterdam, 1988. To appear in: Information and Computation.

[55] M.H.M. Smid, M.H. Overmars, L. Torenvliet and P. van Emde Boas. *Multiple representations of dynamic data structures.* In: G.X. Ritter (ed.), Information Processing 89, Proc. IFIP 11-th World Computer Congress, Elsevier Science Publishers, Amsterdam, 1989, pp. 437-442.

[56] 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.

[57] R.E. Tarjan. *Efficiency of a good but not linear set union algorithm.* J. of the ACM **22** (1975), pp. 215-225.

[58] R.E. Tarjan. *A class of algorithms which require nonlinear time to maintain disjoint sets.* J. Comput. System Sci. **18** (1979), pp. 110-127.

[59] R.E. Tarjan. *Updating a balanced search tree in $O(1)$ rotations.* Inform. Proc. Lett. **16** (1983), pp. 253-257.

[60] R.E. Tarjan and J. van Leeuwen. *Worst-case analysis of set union algorithms.* J. of the ACM **31** (1984), pp. 245-281.

[61] 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.

[62] P. van Emde Boas. *Preserving order in a forest in less than logarithmic time and linear space.* Inform. Proc. Lett. **6** (1977), pp. 80-82.

[63] P. van Emde Boas, R. Kaas and E. Zijlstra. *Design and implementation of an efficient priority queue.* Math. Systems Theory **10** (1977), pp. 99-127.

[64] J. Wiedermann. *Searching Algorithms.* Teubner Texte zur Mathematik, Band 99, Teubner Verlagsgesellschaft, Leipzig, 1987.

[65] D.E. Willard. *New data structures for orthogonal range queries.* SIAM J. Comput. **14** (1985), pp. 232-253.

[66] D.E. Willard and G.S. Lueker. *Adding range restriction capability to dynamic data structures.* J. of the ACM **32** (1985), pp. 597-617.

[67] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, 1976.