GENERAL METHODS FOR ADDING RANGE RESTRICTIONS TO DECOMPOSABLE SEARCHING PROBLEMS

Hans W. Scholten and Mark H. Overmars

RUU-CS-85-21
august 1985

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

GENERAL METHODS FOR ADDING RANGE RESTRICTIONS TO DECOMPOSABLE SEARCHING PROBLEMS

Hans W. Scholten and Mark H. Overmars

Technical Report RUU-CS-85-21
august 1985

Department of Computer Science
University of Utrecht
P.O. Box 80.012
3508 TA Utrecht
the Netherlands

GENERAL METHODS FOR ADDING RANGE RESTRICTIONS TO DECOMPOSABLE SEARCHING PROBLEMS

Hans W. Scholten and Mark H. Overmars

Department of Computer Science, University of Utrecht
P.O.Box 80.012, 3508 TA Utrecht, the Netherlands

Abstract. In this paper we consider the problem of adding range restrictions to decomposable searching problems. Two classes of structures for this problem are described. The first class consists of structures that use little storage and preprocessing time but still have reasonable query time. The second class consists of structures that have a much better query time at the cost of an increase in the amount of storage required and preprocessing time. Both classes of structures can be tuned to obtain different trade-offs. First we only describe static structures. To dynamize the structures a new type of weight-balanced multiway tree (the MWB-tree) is introduced that is used as an underlying structure. The MWB-tree might be useful in other applications as well.

Keywords and phrases. decomposable searching problems, range searching, range restrictions, MWB-tree, global rebuilding, partial rebuilding.

1. Introduction.

Searching problems occur at many places in Computer Science and many efficient algorithms and data structures have been developed for a wide variety of these problems. In a searching problem we ask a question (query) about an object x with respect to a whole set of objects V. A common example of a searching problem is the so-called "member searching" problem: given a set of objects V and another object x, we ask whether x belongs to V. Two other important examples are the "nearest neighbor searching" problem and the

"range searching" problem. Let V be a set of n points in d-dimensional space. The nearest neighbor searching problem asks for a point in V nearest to a given query point x with respect to some prechosen metric. In the range searching problem the query object x is an axis-parallel hyper-rectangle (range) in d-dimensional space. A hyper-rectangle in d-dimensional space can be represented as $[A_0,B_0]\times[A_1,B_1]\times \ldots \times[A_{d-1},B_{d-1}]$. We now ask for all points $x = (x_0,\ldots,x_{d-1})$ in V that lie in the range, i.e. with $A_0 \leq x_0 \leq B_0$, $A_1 \leq x_1 \leq B_1$, ... and $A_{d-1} \leq x_{d-1} \leq B_{d-1}$

Solving a searching problem Q means finding a data structure S for Q that stores the set V of objects such that queries on V with different query objects x can be answered efficiently. The structure S can be static, half-dynamic or dynamic. We call a structure static if it is built once for the set V and is only used for answering queries. S is called half-dynamic if it is possible to insert points in S (and hence in V) efficiently. If S supports both insertions and deletions efficiently we call it dynamic.

During the last few years a lot of efforts were made in finding methods for transforming static solutions to searching problems into (efficient) dynamic solutions. These "dynamization" methods are especially relevant for so-called decomposable searching problems. Let Q(x,V) denote the answer to a searching problem Q with query object x over the set V.

Definition 1.1. A searching problem Q is called decomposable if and only if

$$Q(x,V) = \square[Q(x,A),Q(x,B)]$$

for any partition A U B = V and any query object x, where $\square$ takes O(1) time to compute.

Many searching problems are decomposable. For example, range searching is decomposable with $\square$ = "union", nearest neighbor searching is decomposable with $\square$ = "minimal distance" and member searching is decomposable with $\square$ = "or". Decomposability enables us to split the set in subsets, and derive the answer to a query over the whole set out of the answers to the same query over the subsets, at only nominal extra costs. Bentley [1] was the first to make this important observation and to use it for designing a general

dynamization method for decomposable searching problems. Many other methods have been designed since. (See Overmars[6] for an overview.)

In this paper we consider the problem of "adding range restrictions" to decomposable searching problems. This problem was first investigated by Bentley and Saxe [1,4]. Adding a range restriction means associating a new variable with every object in the set. Queries will now be restricted to those objects that have this new variable in a certain given range.

Definition 1.2. Let Q be a searching problem over a set V. To add a range restriction to Q we associate a value k_p to each point p in the set V. The new searching problem QR now becomes:

$$QR(x,[A:B],V) = Q(x,\{p \in V \mid A \leq k_p \leq B\}).$$

Addition of range restrictions occurs in a number of cases. Suppose we have a nearest neighbor searching problem in 2-dimensional space, in which the points represent cities. Now assume that with each city we are given its population. Instead of asking the city nearest to a certain point we ask for the nearest city to that point with population between, say, 100.000 and 200.000. Hence, we have added a range restriction to the nearest neighbor searching problem. Another example is the d-dimensional range searching problem itself. It is in fact the addition of range restrictions to the (d-1)-dimensional range searching problem.

Some notations are useful in comparing data structures.

Definition 1.3. Let S be a data structure for a searching problem, in which the set V contains n points.

$Q_S(n)$ = the time needed to perform a query on S.
$P_S(n)$ = the time needed to build S (preprocessing time).
$I_S(n)$ = the time needed to perform an insertion on S.
$D_S(n)$ = the time needed to perform a deletion on S.
$M_S(n)$ = the amount of storage (memory) needed for S.
$U_S(n)$ = the time needed to perform an update (insertion or deletion) on S.

All bounds are worst-case bounds, for amortized bounds we add a superscript a. We assume that all functions are non-decreasing and smooth and that P_S and M_S are at least linear.

Bentley [1,4] showed how a static data structure S for a decomposable searching problem can be transformed into a structure R for the transformed problem QR achieving

$$Q_R(n) = O(\log n)\, Q_S(n),$$
$$M_R(n) = O(\log n)\, M_S(n),$$
$$P_R(n) = O(\log n)\, P_S(n).$$

In [4] Bentley and Saxe introduced some new transformations for adding range restrictions to decomposable searching problems.

Theorem 1.1. [4] Let w be some positive integer. Let S be a structure that solves a decomposable searching problem Q. Transform Q into problem QR by adding a range restriction to Q. There exists a structure R for solving QR achieving

$$Q_R(n) = O((2w-1)n^{1/w})\, Q_S(n),$$
$$M_R(n) = O(w)\, M_S(n),$$
$$P_R(n) = O(w)\, P_S(n),$$

and another structure R achieving

$$Q_R(n) = O(2w-1)\, Q_S(n),$$
$$M_R(n) = O(\tfrac{w}{2}\, n^{2/w})\, M_S(n),$$
$$P_R(n) = O(\tfrac{w}{2}\, n^{2/w})\, P_S(n).$$

Both these structures are static. In [7] Willard and Lueker give a dynamic solution for the problem of adding range restrictions to decomposable searching problems.

Theorem 1.2. [7] Let D be a dynamic data structure for a decomposable searching problem Q. Then there exists a transformed structure R for the problem QR with range restriction capability achieving

$$Q_R(n) = O(\log n)\, Q_D(n),$$
$$M_R(n) = O(\log n)\, M_D(n),$$
$$U_R(n) = O(\log n)\, U_D(n).$$

In this paper we will develop two new classes of structures for solving the problem of adding range restrictions to decomposable searching problems. These two classes will have a variety of trade-offs with all previous known results contained in them. Both methods are extensions of the so-called "binary transformation" of Bentley [1] which we shortly recall in section 2. In section 3 we will develop a class of so-called M-structures that use little memory and have reasonable query time and preprocessing costs. The class of Q-structures developed in section 4 has a much better query time at the cost of an increase in storage costs and preprocessing time. The main idea in both classes of structures is to replace the binary tree in the binary transformation by a multiway tree in which all internal nodes contain some F(n) sons. By varying F we get different structures with different trade-offs for query time, memory cost and preprocessing time. Examples of these trade-offs are shown in section 5. M-structures and Q-structures are static. To make them dynamic we introduce a new weight-balanced multiway tree (the MWB-tree) in section 6. In section 7 we use the MWB-tree to obtain dynamic structures for the addition of range restrictions. Finally in section 8 we will use the structures in both classes to solve the d-dimensional range searching problem. This is done by applying addition of range restrictions recursively. The results obtained contain many previously known bounds (e.g.,the structures developed by Bentley and Friedman in [2] and by Bentley and Maurer in [3]). These known bounds were obtained by using many different types of structures. In our approach we only have to vary F(n). Moreover, all our results are dynamic.

2. The binary transformation.

In this section we will recall the so-called "binary transformation" described by Bentley[1] to solve the problem of adding range restrictions to

decomposable searching problems in a uniform way. Let Q be a decomposable searching problem and let S be a static data structure for solving Q. To solve the transformed problem QR we construct a balanced binary tree in which we store the points in the leaves, ordered with respect to the variable added. In internal nodes we store values that split the set in the well known way for binary trees. To each internal node β we associate a S-structure of all points contained in the subtree rooted at β (see figure 2.1.).

To answer a query Q(x,[A:B],V) we search with both A and B in the tree. For some time A and B will follow the same path and nothing needs to be done. But at some internal node β (possibly the root) A and B will turn to different sons β_a and β_b respectively. All points that have their range variable in [A:B] are stored in one of the subtrees rooted at β_a or β_b. We will only show how we search the subtree rooted at β_a, the subtree rooted at β_b can be searched in a similar way. Suppose our search is at node α. If A turns to the left son of α, all points in the subtree rooted at the right son of α have their added variable in the range and we have to perform the query on them. This can easily be done by performing the query on the S-structure stored in the right son of α. If A turns to the right son of α nothing needs to be done. When A and B reach leaves, we have performed the

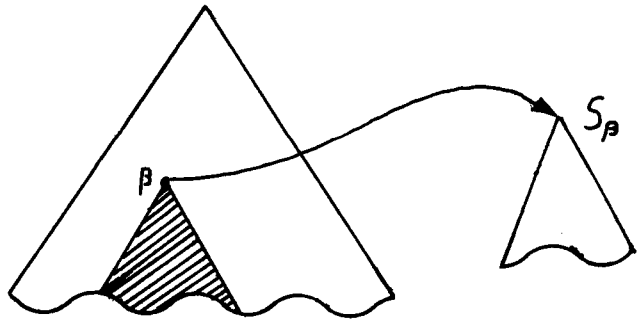

Figure 2.1.

query on all points that have their added variable in the range [A:B]. The answers to the queries on the S-structures can be combined using the composition operator □.

Theorem 2.1. Let S be a structure for a decomposable searching problem Q. Let QR be the transformed problem that adds a range restriction to Q. Then there exists a structure R for QR achieving

$$Q_R(n) = O(\log n)\, Q_S(n),$$
$$M_R(n) = O(\log n)\, M_S(n),$$
$$P_R(n) = O(\log n)\, P_S(n).$$

Proof.

As the main tree has depth O(log n), we only have to search O(log n) S-structures. As each S-structure contains at most n points the query time is bounded by $O(\log n)Q_S(n)$.

The bound on the amount of storage required follows from the fact that at each level of the tree each point is stored in exactly one S-structure and M_S is assumed to be at least linear. To build the structure we first build the main tree and next add the associated structures. Building the tree takes time O(n.log n). Building the associated S-structures takes $O(\log n)P_S(n)$ by the same arguments as used for the amount of storage required.

Q.E.D.

3. M-structures.

In this section we will generalize the binary transformation to a whole class of so-called M-structures that have lower storage cost at an increase in query time. For the binary transformation we used a binary search tree in which at each node β we associated a S-structure containing all points in the subtree rooted at β. To generalize this approach we replace the binary search tree by a F(n)-ary tree. To be more precise: Let F(n) be an integer function with $2<F(n)\leq n$. Let S be a static data structure for a decomposable searching problem Q. A M-structure of order F(n) is a B-tree of order F(n) in which each internal node β with sons $\beta_1, \ldots, \beta_k$ contains S-structures S_1^{β},

..., S_k^β where S_i^β contains all points in the subtree rooted at β_i. (Internal nodes we structure as balanced binary trees to be able to locate sons in O(log F(n)) time.)

Let R be such a M-structure. If we want to perform a query Q(x,[A:B],V) we search with both A and B in R. For some time A and B will follow the same path and nothing needs to be done. But at some internal node β (possibly the root) A and B will turn to different sons β_a and β_b respectively (see figure 3.1.). All points that are contained in the subtrees rooted between β_a and β_b have their added variable in the range, and we have to perform the query on them. This can easily be done by performing the query on the S-structures S^β_{a+1}, ..., S^β_{b-1}. Next we have to handle the points that belong to the subtrees T_{β_a} and T_{β_b} rooted at β_a and β_b. We will only describe the actions in T_{β_a}, T_{β_b} can be handled in a similar way. We continue our search in subtree T_{β_a}. Suppose our search has reached a node α, and the search has to be continued in son α_a. All points that belong to subtrees at the right of α_a have their added variable in the range (see figure 3.2.) and we have to perform the query on them. This can easily be done by performing the query on the S-structures S^α_{a+1}, ... We continue our search with A in T_{α_a}. When A and B reach leaves of R, we have performed the query on all points that have their added variable in the range [A:B]. Because the problem is decomposable we

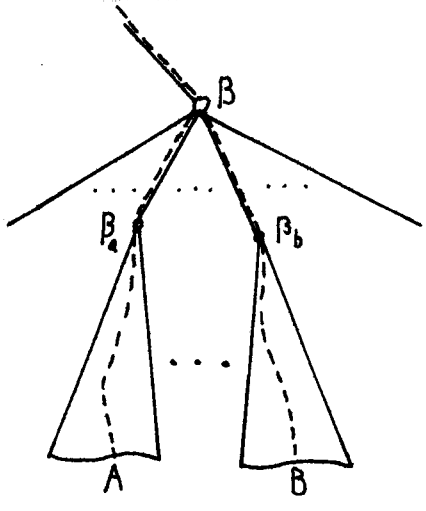

Figure 3.1.

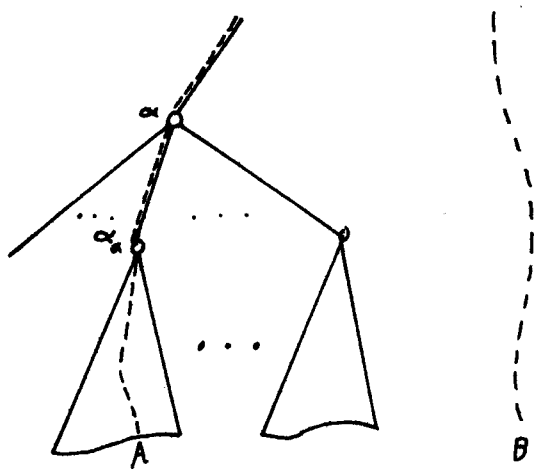

Figure 3.2.

can combine the answers to the different queries using the composition operator □ for the problem.

Theorem 3.1. Let F be an integer function such that $2 < F(n) \leq n$. Let S be a data structure for a decomposable searching problem Q. Let QR be the transformed problem that adds a range restriction to Q. Then there exists a data structure R that solves QR achieving

$$Q_R(n) = O(\frac{\log n}{\log F(n)} F(n))\ Q_S(n),$$

$$M_R(n) = O(\frac{\log n}{\log F(n)})\ M_S(n),$$

$$P_R(n) = O(\frac{\log n}{\log F(n)})\ P_S(n) + O(n \log n).$$

Proof.

When we want to perform a query on R with [A:B] as range, we search with A and B as discussed above. There are $O(\frac{\log n}{\log F(n)})$ levels in the tree, in each node the search time is O(log F(n)), hence the total time needed for searching with A and B in the tree is O(log n). Note that at each level of R we have to perform the original query on at most 2F(n) S-structures, each containing at most n points. The bound for the query time follows.

At each level each point is stored in exactly one S-structure. Because M_S is at least linear all S-structures on the same level together take

$O(M_S(n))$ storage. The bound follows. The bound on the time required for building a M-structure follows in a similar way.

Q.E.D.

See section 5 for examples of the efficiency of this transformation. When $Q_S(n) = \Omega(n^\varepsilon)$ for any positive ε it can be shown that $Q_R(n) = O(F(n))\ Q_S(\frac{n}{F(n)})$. This follows from the fact that at lower levels the S-structures contain less points. In the same way it can be shown that $M_R(n) = O(M_S(n))$ when $M_S(n) = \Omega(n^{1+\varepsilon})$ for any positive ε and that $P_R(n) = O(P_S(n))$ when $P_S(n) = \Omega(n^{1+\varepsilon})$ for any positive ε.

4. Q-structures.

When query time rather than storage is our main concern, the M-structures are of no use. Hence, in this section we define a second class of structures, Q-structures, that have much better query times at the cost of an increase in the amount of storage required and the preprocessing time.

Let $F(n)$ be an integer function with $4<F(n)\leq n$. Let $G(n)=\lceil\sqrt{F(n)}\rceil$. Let S be a static data structure for a decomposable searching problem Q. A Q-structure of order $F(n)$ is a B-tree of order $G(n)$ in which a node β with sons $\beta_1,\ldots,\beta_k$ contains S-structures $S^{\beta}_{i,j}$ for all $1\leq i\leq j\leq k$ where $S^{\beta}_{i,j}$ contains the point in the subtrees below $\beta_i,\ldots,\beta_j$.

Let R be such a Q-structure. If we want to perform a query $Q(x,[A:B],V)$, we search with both A and B through R. For some time A and B will follow the same path and nothing needs to be done. Then at some internal node β (possibly the root) A and B will turn to different sons β_a and β_b respectively. The points that are contained in subtrees that are rooted at sons strictly between β_a and β_b belong to our range, and the query has to be performed on them. This can easily be done by performing the query on structure $S^{\beta}_{a+1,b-1}$. We still have to handle the points in the subtrees rooted at β_a and β_b. We will only discuss the way we handle β_a. β_b can be handled in a similar way. Continue the search with A through the subtree rooted at β_a. Suppose our search is at node α, where A turns to son α_a. Now all points that are stored in subtrees at the right of α_a are in our range, and we have to perform the query on them. This can easily be done by performing the

query on structure $S^{\alpha}_{a+1,G(n)}$. We continue our search with A in the subtree rooted at α_a. When A and B reach leaves of R we have performed the query on all points that have their range variable in range [A:B].

Theorem 4.1. Let F be an integer function such that $4 < F(n) \leq n$. Let S be a data structure for a decomposable searching problem Q. There exists a data structure R for solving QR achieving

$$Q_R(n) = O(\frac{\log n}{\log F(n)})\ Q_S(n) + O(\log n),$$

$$M_R(n) = O(\frac{\log n}{\log F(n)}\ F(n))\ M_S(n),$$

$$P_R(n) = O(\frac{\log n}{\log F(n)}\ F(n))\ P_S(n).$$

Proof.

When we want to perform a query on R with [A:B] as range, we search with A and B through the tree as discussed above. Searching with A and B takes $O(\log G(n))$ time in each node. There are $O(\frac{\log n}{\log G(n)})$ levels in the tree, so the total time needed for searching with A and B is $O(\log n)$. Note that at each level of R we have to perform the original query on at most two S-structures, each containing at most n points. Hence the total query time is $O(\frac{\log n}{\log G(n)})\ Q_S(n) + O(\log n) = O(\frac{\log n}{\log F(n)})\ Q_S(n) + O(\log n)$.

At each level every point is contained in $O(G^2(n))=O(F(n))$ S-structures. Each S-structure has size at most n. Hence, because $M_S(n)$ is at least linear, the S-structures at one level take at most $O(F(n)M_S(n))$ storage. As a result the total amount of storage required is bounded by $O(\frac{\log n}{\log G(n)}\ F(n))\ M_S(n) = O(\frac{\log n}{\log F(n)}\ F(n))\ M_S(n)$.

To build the structure R we first order the n points with respect to the added variable, and construct the B-tree. This will cost $O(n \log n)$ time. Next we construct the associated structures. This takes $O(\frac{\log n}{\log F(n)}\ F(n))\ P_S(n)$, using the same arguments as for determining the amount of storage required. (Note that we can ignore the time needed for sorting the points, because $\frac{F(n)}{\log F(n)}\ P_S(n) = \Omega(n)$.)

Q.E.D.

For any positive ε it can be shown that $Q_R(n) = O(Q_S(n))$ when $Q_S(n) = \Omega(n^{\varepsilon})$, $M_R(n) = O(F(n))\ M_S(n)$ when $M_S(n) = \Omega(n^{1+\varepsilon})$ and that $P_R(n) = O(F(n))\ P_S(n)$ when $P_S(n) = \Omega(n^{1+\varepsilon})$.

5. Examples.

In this section we will give some examples of trade-offs we can get by choosing different functions F(n). It will appear that all previous known results are contained in one of the classes of structures developed in this paper.

We start with choosing F a constant, say $F(n) = c$. Both classes give the same results, namely

$$Q_R(n) = O(\log n)\ Q_S(n),$$
$$M_R(n) = O(\log n)\ M_S(n),$$
$$P_R(n) = O(\log n)\ P_S(n).$$

These are the exact results of the binary transformation developed by Bentley [1].

Let us consider $F(n) = n^{\delta}$, where δ is a constant satisfying $0 < \delta < 1$. Now $O(\frac{\log n}{\log F(n)}) = O(\frac{\log n}{\delta \log n}) = O(1)$, hence the results for M-structures are the following

$$Q_R(n) = O(n^{\delta})\ Q_S(n),$$
$$M_R(n) = O(M_S(n)),$$
$$P_R(n) = O(P_S(n)) + O(n \log n).$$

For Q-structures we obtain

$$Q_R(n) = O(Q_S(n)) + O(\log n),$$
$$M_R(n) = O(n^{\delta})\ M_S(n),$$
$$P_R(n) = O(n^{\delta})\ P_S(n).$$

Both results are the same as the ones Bentley and Saxe developed in [4] stated in theorem 1.1. (Note that in theorem 1.1. w is some constant.)

Many other trade-offs can be obtained as well. The result of theorem 3.1. can be rewritten as

$$Q_R(n) = O(B(n))\ Q_S(n),$$
$$M_R(n) = O(A(n))\ M_S(n),$$
$$P_R(n) = O(A(n))\ P_S(n) + O(n \log n),$$

and the result of theorem 4.1. as

$$Q_R(n) = O(A(n))\ Q_S(n) + O(\log n),$$
$$M_R(n) = O(B(n))\ M_S(n),$$
$$P_R(n) = O(B(n))\ P_S(n),$$

with

$$A(n) = \frac{\log n}{\log F(n)} \quad \text{and} \quad B(n) = \frac{\log n}{\log F(n)} F(n).$$

In figure 5.1. we show some possible choices that can be used to get different trade-offs.

6. The MWB-tree.

In this section we develop a dynamic multiway weight-balanced search tree, the MWB-tree, that will be used in section 7 to transform the static M- and Q-structures into dynamic structures with reasonable insertion and deletion times. Let n_β denote the number of keys stored in the subtree rooted at β. Let $0<\alpha<1/2$. Let $k>2$.

F(n)	A(n)	B(n)
c	$\log n$	$\log n$
n^δ	1	n^δ
$2^{\sqrt{\log n}}$	$\sqrt{\log n}$	$\sqrt{\log n}\ 2^{\sqrt{\log n}}$
$\log^\delta n$	$\frac{\log n}{\log \log n}$	$\frac{\log^{1+\delta} n}{\log \log n}$

Figure 5.1.

Definition 6.1. A MWB[α]-tree of order k is a multiway tree, storing the keys in the leaves, with the following properties:

- Each internal node, except the fathers of the leaves have between $\lceil k/2 \rceil$ and k sons.
- The fathers of the leaves have between 2 and k sons.
- For each internal node β, $n_\beta \leq \left\lceil \frac{1}{\lceil k/2 \rceil (1-\alpha)} n_{father(\beta)} \right\rceil$.

The MWB[α]-tree is somewhat in between a BB[α]-tree and a B-tree of order k. But note that we do not force all leaves to be at the same level. As k will be replaced by F(n) in section 7, we do not treat k as a constant. α will be treated as a constant. Splitting values in internal nodes are stored in a binary tree to speed up searching.

Lemma 6.1. The depth of a MWB[α]-tree of order k is bounded by $O(\frac{\log n}{\log k})$.

Proof.

Let β be an internal node at depth i in the MWB-tree. Clearly the number of keys in the subtree rooted at β is bounded by $\left\lceil \frac{n}{\lceil k/2 \rceil^i (1-\alpha)^i} \right\rceil$. The lemma follows.

Q.E.D.

Definition 6.2. Let β be a node in a MWB[α]-tree of order k. We call β "perfectly balanced" if each subtree rooted at a son of β contains at most $\left\lceil \frac{1}{\lceil k/2 \rceil} n_\beta \right\rceil$ keys, and β has exactly $\lceil k/2 \rceil$ sons. We call a MWB[α]-tree of order k "perfectly balanced" if the fathers of the leaves each contain at most $\lceil k/2 \rceil$ sons (keys), and all other internal nodes are perfectly balanced.

Lemma 6.2. Given an ordered set of n keys, a perfectly balanced MWB-tree can be built in O(n) time.

Proof.

We cannot use the standard technique of building a B-tree by starting at the leaves because we cannot decide how many sons fathers of leaves should get. Hence, we use a different technique. We first build, from the root to the leaves, a skeleton tree without filling in the keys and internal splitting values. This can easily be done in O(n) time. Next, working from the leaves upwards, we add the keys and the splitting values. It can easily be shown that this can be done in O(n) time. The details are left as an exercise to the reader.

Q.E.D.

To perform insertions and deletions efficiently we will use two techniques from Overmars[6]. Insertion will use "partial rebuilding". Deletions will be handled in a global way using "global rebuilding". We will not describe these techniques in great detail. See [6] for more details.

We will first only consider insertions. Suppose we want to insert a key K. First we search for K in the tree. this will lead to the node β in the tree where K has to be inserted as a son. We insert K here at the right place, find an appropriate splitting value and insert this splitting value in the balanced binary tree stored in β. If β now has more than k sons we rebuild the subtree rooted at β as a perfectly balanced MWB-tree.

Next we walk back from K towards the root, and adjust n_β at each node β on the path. We also locate the highest node β that is out of balance, i.e., whose subtree contains too many keys. There are two possible ways of restoring the balance. If the father of β has less than k sons, we reconstruct the subtree rooted at β as two perfectly balanced MWB-trees, each containing at most $\lceil n_\beta/2 \rceil$ keys, and replace β by these two subtrees. This clearly brings the tree back into balance. If the father of β has k sons we rebuild the whole subtree rooted at the father of β as a perfectly balanced MWB-tree. Although this sometimes takes a lot of time we will show that the amortized insertion time will be low.

Lemma 6.3. The amortized insertion time in a MWB-tree of n keys in $O(\log n)$.

Proof.

Remember that k cannot be treated as a constant. Hence, we have to be careful in estimating bounds. When inserting a key we have to walk down and up the tree, sometimes rebuild the subtree rooted at the father of the inserted leave, and sometimes rebuild the subtree rooted at some internal node. Clearly, the cost for walking up and down the tree is bounded by $O(\log n)$. Amortizing the cost for rebuilding subtrees is done in a similar way as described in chapter 4 of [6] and will not be described in detail here.

Rebuilding the subtree rooted at the father of a leaf takes time $O(k)$. When this node was created, as part of a rebuild perfectly balanced subtree, it had $\lceil k/2 \rceil$ sons. Now it has k+1 sons. Hence, we can charge the $O(k)$ work to at least $\lceil k/2 \rceil$ keys that have been inserted since.

Assume a node β went out of balance. Then $n_\beta > \left\lceil \frac{1}{\lceil k/2 \rceil (1-\alpha)} n_{father(\beta)} \right\rceil$. At the moment the subtree rooted at β was rebuild for the last time it contained $\left\lceil \frac{1}{\lceil k/2 \rceil} n_{father(\beta)} \right\rceil$ keys. Hence, there have been at least $\Omega(n_\beta)$ insertions in the subtree since (see [6] for details). When the subtree rooted at β has to be rebuild this takes $O(n_\beta)$ time (see Lemma 6.2.). Charging this cost to the insertions that have taken place makes for $O(1)$ per insertion. When the father of β has to be rebuild it must have k sons. At the moment this father of β was rebuild the last time it had $\lceil k/2 \rceil$ sons. Hence, $\lfloor k/2 \rfloor$ times a son must have been split and, hence, there must have been $\Omega(n_{father(\beta)})$ insertions below sons of father(β). Charging the rebuilding cost to these insertions is $O(1)$ per insertion. It can easily be shown (see [6]) that each insertion is charged at most $O(\frac{\log n}{\log k})$ times $O(1)$ work. The amortized insertion time bound follows.

Q.E.D.

To delete a key K we search for K in the tree. If K is not present we are done. Otherwise we remove K as a leaf. We don't do any rebalancing of the tree. In this way, clearly, the tree will (slowly) go out of balance. On the other hand, deletions of this sort are "weak" in the sense that they

do not increase the query time. (See Overmars[6] for a precise definition of weak updates.) In [6] it was shown that a structure S that allows for weak deletions in time $WD_S(n)$ can be transformed into a structure that supports real deletions in time $O(WD_S(n)+P_S(n)/n)$, without any essential loss in the query time and amount of storage required. In our case $WD_S(n)=O(\log n)$. Hence, we obtain the following result:

Theorem 6.4. Insertions and deletions in a MWB-tree can be performed in $O(\log n)$ time. The insertion time bound being amortized.

In this section we developed a new weight-balanced multiway search tree, with the normal $O(\log n)$ query and update time bounds, using $O(n)$ storage. The structure has a depth of $O(\frac{\log n}{\log k})$ for any prechosen k. In the next section we will make k behave like a function of n. In this way we obtain a structure that has a depth that is essential smaller than log n. This structure we will use as the basis for a dynamic structure for the addition of range restrictions.

7. A dynamic solution for adding range restrictions.

To use the MWB-tree for adding range restrictions to decomposable searching problems we must take care that k behaves like F(n) (or G(n)). To this end, we sometimes rebuild the whole structure with k=F(n) and do not change k as long as the size of the set is less than 2n and more than n/2. When the size of the set becomes 2n or n/2 we rebuild the whole structure with the new k. Hence, at least n/2 updates take place at which the structure needs not be rebuild. We can charge the rebuilding cost to these n/2 updates. This will only add a small amount of extra cost to the amortized update time, but guarantees that k behaves like F(n). (This technique is a kind of global rebuilding [6].) We will from now on call such a structure a MWB-tree of order F(n).

A dynamic M-structure is a MWB-tree of order F(n) in which all points are stored in the leaves, ordered with respect to their added variable. S-structures are associated to internal nodes in exactly the same way as in section 3.

Theorem 7.1. A perfectly balanced dynamic M-structure can be constructed in $O(\frac{\log n}{\log F(n)})P_S(n))+O(n.\log n)$ time and uses $O(\frac{\log n}{\log F(n)})M_S(n))$ storage.

Proof.

Constructing the MWB-tree takes $O(n.\log n)$ time. Buiding the associated structures takes $O(\frac{\log n}{\log F(n)})P_S(n))$ time by the same arguments as used in the proof of theorem 3.1. Also the bound on the amount of storage required follows in the same way.

Q.E.D.

To insert (or delete) a point (p,k_p) we search with k_p in the MWB-tree. For every node β on the search path where k_p turns to son β_i we insert (delete) p in S_i^β. Next we insert (delete) k_p in the MWB-tree and walk back to the root to find the highest node β that is out of balance. To rebalance we rebuild subtrees in the way described in section 6.

Theorem 7.2. Let F be an integer function such that $2 < F(n) \leq n$. Let S be a dynamic data structure for a decomposable searching problem Q. There exists a data structure R that solves QR achieving

$$Q_R(n) = O(\frac{\log n}{\log F(n)} F(n))\ Q_S(n),$$

$$M_R(n) = O(\frac{\log n}{\log F(n)})\ M_S(n),$$

$$P_R(n) = O(\frac{\log n}{\log F(n)})\ P_S(n) + O(n \log n),$$

$$I_R^a(n) = O\left(\frac{\log n}{\log F(n)}\ (I_S(n) + \frac{\log n}{\log F(n)}\ P_S(n)/n)\right),$$

$$D_R^a(n) = O\left(\frac{\log n}{\log F(n)}\ (D_S(n) + P_S(n)/n)\right).$$

Proof.

The depth of the dynamic M-structure is bounded by $O(\frac{\log n}{\log F(n)})$. Hence, the query time follows in the same way as in the proof of theorem 3.1. The storage cost and preprocessing time follow from theorem 7.1.

The average update time can be split into four parts. i) The time required for searching in the MWB-tree and performing the insertion or deletion in this tree. This takes O(log n) time (see section 6). ii) The amount of time required for inserting or deleting points from associated structures. This clearly is bounded by $O(\frac{\log n}{\log F(n)} U_S(n))$. iii) The amortized amount of time required for rebuilding subtrees needed for rebalancing. By the same arguments described in section 6, using partial rebuilding, this can be estimated as $O(\frac{\log n}{\log F(n)} P_R(n)/n) = O(\frac{\log^2 n}{\log^2 F(n)} P_S(n)/n)$ per insertion. For deletions this amount is much less. Using global rebuilding it adds an average of $O(P_R(n)/n) = O(\frac{\log n}{\log F(n)} P_S(n)/n)$ per deletion. iv) The amortized cost for rebuilding the whole structure when k must be updated. It can easily be shown that this adds $O(P_R(n)/n) = O(\frac{\log n}{\log F(n)} P_S(n)/n)$ per update. The amortized insertion and deletion time follows.

Q.E.D.

Some better estimates can be made when $P_S(n)=\Omega(n^{1+\varepsilon})$ or $Q_S(n)=\Omega(n^{\varepsilon})$ or $U_S(n)=\Omega(n^{\varepsilon})$ for some $\varepsilon>0$.

Dynamic Q-structures are defined in exactly the same way. Again we use a MWB-tree of order G(n) and associate structures in the way as described in section 4. (Remember $G(n)=\sqrt{F(n)}$.) To insert or delete a point (p,k_p) we search with k_p in the MWB-tree. For every internal node β where we have to continue the search in the son β_i we insert or delete p in all structures $S^{\beta}_{s.t}$ for all $s\leq i$ and $i\leq t$. Next we insert or delete k_p in the tree, find the highest node that is out of balance and rebuild a subtree in the way described in section 6. It can easily be shown that this leads to the following result:

Theorem 7.3. Let F be an integer function such that $4 < F(n) \leq n$. Let S be a data structure for a decomposable searching problem Q. There exists a data structure R that solves QR achieving

$$Q_R(n) = O(\frac{\log n}{\log F(n)})\ Q_S(n) + O(\log n),$$

$$M_R(n) = O(\frac{\log n}{\log F(n)}\ F(n))\ M_S(n),$$

$$P_R(n) = O(\frac{\log n}{\log F(n)}\ F(n))\ P_S(n),$$

$$I^a_R(n) = O\left(\frac{\log n}{\log F(n)}\ (F(n)\ I_S(n) + \frac{\log n}{\log F(n)}\ F(n)\ P_S(n)/n)\right),$$

$$D^a_R(n) = O\left(\frac{\log n}{\log F(n)}\ (F(n)\ D_S(n) + F(n)\ P_S(n)/n)\right).$$

Let us look at some examples. When we take $F(n) = c$ we obtain a structure with the following bounds

$$Q_R(n) = O(\log n)\ Q_S(n),$$

$$I^a_R(n) = O(\log n)\ I_S(n) + O(\log^2 n)\ P_S(n)/n,$$

$$D^a_R(n) = O(\log n)\ D_S(n) + O(\log n)\ P_S(n)/n.$$

When we take $F(n) = n^\delta$ the M-structure will have the following bounds

$$Q_R(n) = O(n^\delta)\ Q_S(n),$$

$$I^a_R(n) = O(\ I_S(n) + P_S(n)/n\ ),$$

$$D^a_R(n) = O(\ D_S(n) + P_S(n)/n\ ).$$

For Q-structures we obtain

$$Q_R(n) = O(Q_S(n)) + O(\log n),$$

$$I^a_R(n) = O(\ n^\delta\ I_S(n) + n^\delta\ P_S(n)/n\ ),$$

$$D^a_R(n) = O(\ n^\delta\ D_S(n) + n^\delta\ P_S(n)/n\ ).$$

In many practical cases the update times can be improved by making use of the fact that, when a subtree has to be rebuild, the old subtree is available. Let $B_S(n)$ be the time required to build a S-structure from an ordered

set of points (some prechosen ordering). If $B_S(n)$ is smaller than $P_S(n)$ it is easy to see that one can replace the $P_S(n)$ in the amortized insertion and deletion time bounds in theorems 7.2. and 7.3. by $B_S(n)$.

8. d-dimensional range searching.

In this section we will use the M- and Q-structures recursively to solve the d-dimensional range searching problem. The 1-dimensional range searching problem can be solved in $O(\log n)$ query time, $O(\log n)$ update time and $O(n.\log n)$ preprocessing time using $O(n)$ storage. Clearly, $B_S(n)=O(n)$. Next we apply addition of range restrictions d-1 times. Let us look what we get for different choices of $F(n)$.

When we choose $F(n)=c$ we obtain

$$
\begin{aligned}
Q_R(n) &= O(\log^d n),\\
M_R(n) &= O(n \log^{d-1} n),\\
P_R(n) &= O(n \log^d n),\\
B_R(n) &= O(n \log^{d-1} n),\\
U_R^a(n) &= O(\log^d n).
\end{aligned}
$$

These bounds are near to the best known results. (See e.g. Lueker[5] and Willard and Lueker[7].)

Next we apply theorem 7.2. with $F(n) = n^\delta$. It shows that for each $\varepsilon>0$ there exists a dynamic M-structure with

$$
\begin{aligned}
Q_R(n) &= O(n^\varepsilon),\\
M_R(n) &= O(n),\\
P_R(n) &= O(n \log n),\\
B_R(n) &= O(n),\\
U_R^a(n) &= O(\log n),
\end{aligned}
$$

Bentley and Maurer [3] developed a data structure with the same bounds but their structure is static.

Applying theorem 7.3. with $F(n) = n^\delta$ we obtain for each $\varepsilon>0$ a dynamic Q-structure with

$$Q_R(n) = O(\log n),$$
$$M_R(n) = O(n^{1+\varepsilon}),$$
$$P_R(n) = O(n^{1+\varepsilon}),$$
$$B_R(n) = O(n^{1+\varepsilon}),$$
$$U_R^a(n) = O(n^{\varepsilon}),$$

Also these bounds were obtained in [3] but again only in a static structure.

As a final example we apply theorem 7.2 with $F(n) = \log^{\delta} n$. It shows that for each $\varepsilon>0$ there exists a dynamic M-structure with

$$Q_R(n) = O(\frac{\log^{d+\varepsilon} n}{\log^{d-1}\log n})$$

$$M_R(n) = O(n \frac{\log^{d-1} n}{\log^{d-1}\log n})$$

$$P_R(n) = O(n \frac{\log^{d} n}{\log^{d-1}\log n})$$

$$B_R(n) = O(n \frac{\log^{d-1} n}{\log^{d-1}\log n})$$

$$U_R(n) = O(n \frac{\log^{d} n}{\log^{d-1}\log n})$$

The above examples only give a small number of possible trade-offs. Many other trade-offs can be obtained as well. They cover almost all known results on range searching. Moreover, all the structures are dynamic and obtained by applying one general technique.

9. Conclusions and open problems.

In this paper we have presented two classes of structures for the addition range restrictions to decomposable searching problems. These classes give us a wide variety of structures with different trade-offs for query time, memory cost and preprocessing time. Both static and dynamic structures have been considered. As an application we applied the results to the d-dimensional range searching problem. In this way we did obtain a whole class of structures, containing most of the known results and adding many new results.

Some open problems do remain. The transformations described in this paper only yield good amortized update time bounds. It is quite easy to change the deletion time bound in a worst-case bound (by applying techniques from [6]) but the insertion time remains amortized. It is an interesting question whether this amortized bound can be turned into a worst-case bound as well. It is not clear whether the bounds in this paper are optimal. Other, better transformations might exist. Hence, there is a need for lower bounds on the efficiency of the transform. Some extra techniques, like e.g. presorting, might be useful in reducing the bounds obtained in this paper further for some special subclasses of decomposable searching problems.

10. References

[1] Bentley, J.L., Decomposable searching problems, Inform. Proc. Lett. 8 (1979), pp. 244-251.

[2] Bentley, J.L., Friedman, J.H., Data structures for range searching, ACM Comput. Surveys 11 (1979), pp. 397-409.

[3] Bentley, J.L., Maurer, H.A., Efficient worst-case data structures for range searching, Acta Inform. 13 (1980), pp. 155-168.

[4] Bentley, J.L., Saxe, J.B., Transformations adding range variables to data structures, extended abstract, Dept. of Computer Science, Carnegie-Mellon University, 1979.

[5] Lueker, G.S., A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems, Techn. Rep. #129, Dept. of Inform. and Computer Science, University of California, Irvine, Calif., 1979.

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

[7] Willard, D.E., Lueker, G.S., Adding range restriction capability to dynamic data structures, J. of the ACM 32 (1985), pp. 597-617.