DYNAMIZATION OF DECOMPOSABLE SEARCHING PROBLEMS YIELDING GOOD WORST-CASE BOUNDS

Mark H. Overmars and Jan van Leeuwen

RUU-CS-80-6

September 1980

Rijksuniversiteit Utrecht

Vakgroep Informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

DYNAMIZATION OF DECOMPOSABLE SEARCHING PROBLEMS YIELDING GOOD WORST-CASE BOUNDS

Mark H. Overmars and Jan van Leeuwen

Technical Report RUU-CS-80-6
September 1980

Department of Computer Science
University of Utrecht
P.O.Box 80.002, 3508 TA Utrecht
the Netherlands

DYNAMIZATION OF DECOMPOSABLE SEARCHING PROBLEMS YIELDING GOOD WORST-CASE BOUNDS

Mark H. Overmars* and Jan van Leeuwen

Department of Computer Science, University of Utrecht
P.O.Box 80.002, 3508 TA Utrecht, the Netherlands

Abstract. Dynamizations are techniques for converting static data structures for set problems into structures which permit insertions and deletions. A technique is described for dynamizing data structures of common types of decomposable searching problems, yielding good worst-case bounds rather than merely good average bounds on all operations to be performed on the resulting structures. Among the many applications are dynamized quad-trees, range-trees and segment trees.

Keywords and phrases. Decomposable searching problems, decomposable counting problems, DD-searching problems, dynamization, worst-case bounds, quad-trees, k-d trees, range-trees, segment trees, nearest neighbor searching.

1. Introduction.

Searching problems are problems in which we ask a question (query) about an object with respect to a set of other objects (points). Let $Q(x,V)$ denote the answer to a searching problem Q with object x and set V. Efficient data structures are usually devised to allow for speedy searches through the set of objects V.

Definition. A searching problem Q is said to be decomposable if and only if for any partition $V_1 \cup V_2$ of V one has

$$Q(x,V) = \square(Q(x,V_1), Q(x,V_2))$$

for an efficiently computable operator $\square$.

Decomposable searching problems were first defined by Bentley [2], as an outgrowth of his study of searching problems for multidimensional point sets. While many efficient data structures in this area are inflexible and only

---

* This author was supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

practical for static sets of points, Bentley [2] (see also Saxe and Bentley [17]) demonstrated that the very property of decomposability of a search problem enables one to build a dynamic data structure for the problem out of a known static solution. For decomposable searching problems we do not have to keep the whole set in a single data structure but we can build a "dynamic system" of static structures for disjoint subsets instead. The answer to a query can be obtained from the answers to the same query over the static structures by applying the operator $\square$. Any method of converting a static data structure into a structure for the same problem allowing insertions and deletions (or just insertions), is called a "dynamization" (cf. van Leeuwen and Wood [21]).

Over the past three years much effort has been devoted to obtaining efficient dynamizations for decomposable searching problems. Bentley [2] and Saxe and Bentley [17] gave a variety of methods to perform insertions, achieving different levels to efficiency. Soon after, several authors addressed and solved the problem of efficiently supporting both insertions and deletions (Maurer and Ottmann [9], van Leeuwen and Wood [21], Overmars and van Leeuwen [12], van Leeuwen and Maurer [20]). Recently Overmars and van Leeuwen [13] and Mehlhorn and Overmars [10] gave very general dynamization schemes inspired by an analogy to arbitrary number systems. Almost all methods resulted in good average bounds on the time required for insertions and deletions (or just insertions).

Saxe and Bentley [17] did show how insertions could be performed to get even low worst-case bounds. In this paper we settle the standing problem to extend this to deletions as well and devise an intricate scheme of partial construction, buffering and transaction delays to process both insertions and deletions for a large class of decomposable searching problems with low worst-case bounds. In section 2 we shall offer a method to perform insertions in low worst-case time, which is similar to the method of [17], but which can be more easily adapted to the case when deletions need to be processed also. In sections 3, 4 we indeed show that both for decomposable counting problems ([17,20]) and for DD-searching problems ([12,20]) efficient worst-case dynamizations can be devised that support both insertions and deletions. In section 5 we argue that the techniques offered can also be used for the general methods offered in [10,13], to change average time bounds in worst-case bounds. Among the many applications are dynamized quad-trees, range-trees and segment trees and dynamic structures for deciding containment in the intersection of a set of halfspaces and answering nearest neighbor queries.

Notation.

$Q_S(n)$ = the query time on a static structure of n points

$P_S(n)$ = the time required to build a static structure of n points

$Q_D(n)$ = the query time on a dynamic structure containing n points

$I_D(n)$ = the insertion time in a dynamic structure containing n points (worst-case bound)

$D_D(n)$ = the deletion time in a dynamic structure containing n points (worst-case bound)

We assume that Q_S is nondecreasing, that P_S is at least linear and that Q_S and P_S are "smooth" (e.g. $Q_S(cn) = O(Q_S(n))$ for every constant c).

2. Insertions in decomposable searching problems.

Dynamizations of decomposable searching problems normally are achieved by splitting a set in different size subsets, each statically structured, and maintaining these "blocks" in some efficient way. When a point x is inserted, one initiates a new static structure for it. To avoid that the number of structures becomes too big, one sometimes collects some small structures and builds one single (and larger) structure for their points instead. Queries are answered by querying all blocks separately and combining the answers, as we can in decomposable searching problems without any major loss of efficiency. It follows that sometimes an insertion is expensive (when we have to build a new structure of many points) but as the frequency of these large clean-ups can be kept small, insertions often are quite cheap and the average insertion time will stay low.

To achieve a low worst-case bound on the insertion time, one cannot afford to do all constructions at once. Instead of building a new structure out of old structures at the moment this is needed, the work must be spread over a number of insertions to follow. Meanwhile the old structures remain in use for query answering. Although Saxe and Bentley [17] already provided a method to obtain worst-case insertion times along these lines, we shall describe a slightly different structure (with the same efficiency) that is more easily adapted to the case when both insertions and deletions must be handled (see sections 3, 4).

The dynamic structure we use consists of bags BA_0, BA_1, BA_2, ... Each bag BA_i contains at most 3 blocks $B_i^u[1]$, $B_i^u[2]$ and $B_i^u[3]$ of size 2^i that are "in use" (i.e., are available for query answering). Moreover, each BA_i contains at most one block B_i^c that is "under construction" and will become of size 2^i. To prevent the number of blocks in BA_i from becoming too big, as soon as a second B_i^u becomes available in BA_i, we "take them together" and start building a B_{i+1}^c of 2^{i+1} points in BA_{i+1} out of them (see figure 1). The work is spread over the next 2^{i+1} insertions, each time doing $P_S(2^{i+1})/2^{i+1}$ steps of the construction.

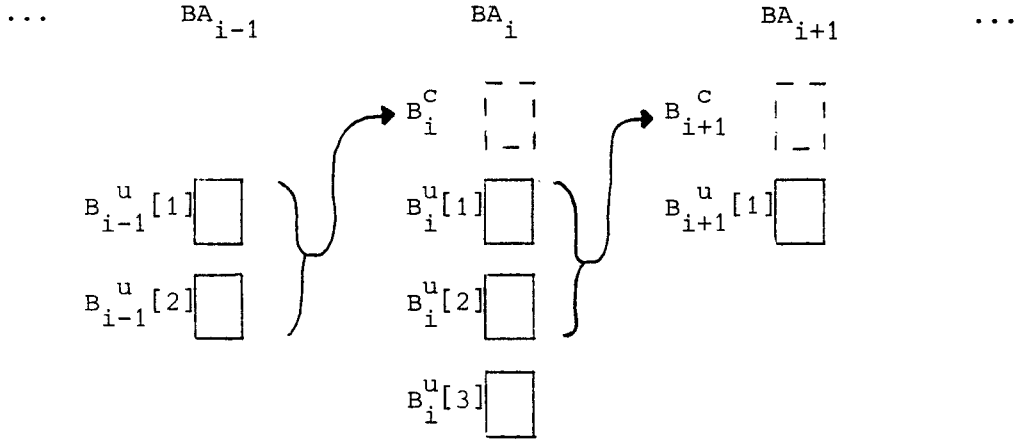

Figure 1.

In the meantime a third B_i^u may get into BA_i. By the time a fourth would come in, the construction of B_{i+1}^c will have been completed (see lemmas 2.3, 2.4). Hence the two B_i^u it was built from can be discarded (for clarity we assume this takes only constant time) and BA_i once again has just 2 blocks for use. Turning the B_{i+1}^c that was just built into a "block for use" (which is now legitimate), BA_{i+1} has room again to start another block under construction. At each insertion, the algorithm will thus attend to all bags simultaneously. For each bag BA_i we maintain the following information:

$B_i^u[1 \,..\, 3]$: the structures in use of size 2^i.

B_i^c : the block under construction.

$building_i$: a boolean that tells us whether B_i^c is still under construction.

$work_i$: the number of steps we have spent on building B_i^c.

emp_i : the first empty block B_i^u

We also have one global variable n that gives the number of points in the structure. In the beginning n = 0, all structures are empty, and for all i $building_i$ = false, $work_i$ = 0 and emp_i = 1. The procedure INSERT takes care of all actions that need be performed when we want to insert a new point into the set.

```
proc INSERT(p);
{p is the new point we want to insert.}
begin
    {add one to the total number of points.}
    n := n+1;
    {do work on building blocks under construction starting at the highest
     filled bag.}
```

```
for i := ⌊log n⌋ downto 1 do
   begin
      {if we are busy building B_i^c then do some work}
      if building_i
        then do P_S(2^i)/2^i work on building B_i^c out of B_{i-1}^u[1] and B_{i-1}^u[2];
             {note that we have done some work}
             work_i := work_i+1
      fi;
      {if the time has come to start using B_i^c then make it "in use"}
      if work_i = 2^i
        then B_i^u[emp_i] := B_i^c;
             {clear B_i^c}
             B_i^c := ∅;
             work_i := 0;
             building_i := false;
             {throw B_{i-1}^u[1] and B_{i-1}^u[2] away and shift B_{i-1}^u[3]}
             B_{i-1}^u[1] := B_{i-1}^u[3];
             emp_{i-1} := 2;
             {if the new block in BA_i is the second one we have to start
              building B_{i+1}^c}
             if emp_i = 2
               then building_{i+1} := true
             fi;
             emp_i := emp_i+1
      fi.
   end;
 {insert the new point p}
 build B_0^u[emp_0] out of p;
 if emp_0 = 2
   then building_1 := true
 fi;
   emp_0 := emp_0+1
end of INSERT;
```

Lemma 2.1. Every point p in the set is always contained in exactly one block in use.

Proof

When we insert p it is built in one B_0^u. p can only disappear when we throw the B_i^u it is part of away. But we do so only at the same moment we start using a bigger block p is part of. p can never become part of more than one block in use because, when we start using a new block that contains p, the old block in use is thrown away.

□

Lemma 2.2. Each block in BA_i contains (or will contain) 2^i points.

Proof

Blocks in BA_0 are built from one point and blocks in BA_i are built out of the points in two blocks in BA_{i-1}. Hence, by induction, the result follows.

□

Lemma 2.3. When we complete a block B_i^c and turn it into a block in use there is at least one B_i^u empty.

Proof

Consider how blocks in BA_i develop. At the moment BA_i contains two blocks in use we start building a B_{i+1}^c out of them. The construction is spread over the next 2^{i+1} insertions. Observe that BA_i got a second block in use because B_i^c was ready. It follows that when we start building B_{i+1}^c, B_i^c is empty. After (at least) 2^i insertions another B_i^c is ready and it is changed into $B_i^u[3]$. After another 2^i insertions, yet another B_i^c is ready and we would like to change it into a B_i^u. But at this moment B_{i+1}^c is ready also, and, because we treat bags from high to low index, $B_i^u[1]$ and $B_i^u[2]$ have just been discarded. Hence there is room again for a new B_i^u.

□

Corollary 2.4. When $B_i^u[2]$ becomes present B_{i+1}^c is empty.

Proof

From the proof of lemma 2.3. it follows that BA_i gets a second block in use only when BA_{i+1}^c has just become empty (or BA_i is the highest filled bag).

□

2.1. - 2.4. show that the algorithm described maintains the structure correctly.

Theorem 2.5. Given a decomposable searching problem, one can dynamize it such that

$$Q_D(n) \leq 3 \log n . Q_S(\tfrac{1}{2}n)$$
$$I_D(n) \leq (\log n+1) . P_S(n)/n$$

Proof

We use the data structure and the insertion algorithm described above. Let us consider the query time first. From the discussion in the proof of lemma 2.3. it follows that at the moment a bag BA_i gets a new block in use, all preceding bags contain three blocks in use. It follows that BA_i will not contain a block in use as long as $n < 3.2^i - 3$. Hence for $i \geq \lfloor \log n \rfloor$, BA_i will contain no blocks in use. So the highest filled bag is $BA_{\lfloor \log n \rfloor - 1}$. It follows that at most $\lfloor \log n \rfloor$ bags contain blocks in use. As each bag contains at most three blocks in use the total number of blocks on which a query must be performed is bounded by 3 log n. The biggest such block is in $BA_{\lfloor \log n \rfloor - 1}$ and has size $2^{\lfloor \log n \rfloor - 1} \leq \frac{1}{2}n$. Hence the query time is bounded by $3 \log n . Q_S(\frac{1}{2}n)$.

When we perform an insertion we have to do $P_S(2^i)/2^i$ work in every bag BA_i containing a block under construction. As the highest bag containing blocks in use is $BA_{\lfloor \log n \rfloor - 1}$, the highest bag possibly containing a B_i^C is $BA_{\lfloor \log n \rfloor}$. Hence the number of such bags is bounded by log n+1. From the assumption that P_S is at least linear it follows that $P_S(2^i)/2^i \leq P_S(n)/n$ $(i \leq \lfloor \log n \rfloor)$ and that the insertion time is bounded by $(\log n+1)\ P_S(n)/n$.

□

We can rewrite the result of theorem 2.5. into

$$Q_D(n) = O(\log n)\ Q_D(n)$$
$$I_D(n) = O(\log n)\ P_S(n)/n$$

It can even be shown that $Q_D(n) = O(Q_S(n))$ when $Q_S(n) = \Omega(n^\varepsilon)$ $(\varepsilon > 0)$ and that $I_D(n) = O(P_S(n)/n)$ when $P_S(n) = \Omega(n^{1+\varepsilon})$ $(\varepsilon > 0)$. It follows that the method presented here yields bounds of the same order as the method of Saxe and Bentley [17].

3. Deletions in decomposable counting problems.

Thus far we have only considered the problem of inserting points efficiently, but often one would like to be able to delete points from the set as well. Deletions often are a stumbling block in dynamizations. Saxe and Bentley [17] proved that there cannot exist a dynamization method for decomposable searching problems in general which achieves both low query times and low insertion and

deletion times. On the other hand, they showed that for a special subclass of decomposable searching problems deletions can be accomodated with good average bounds. The following terminology is from van Leeuwen and Maurer [20].

Definition. A decomposable searching problem Q is called a decomposable counting problem if and only if for any object x and for any set of points V and any subset V_1 of V the answer $Q(x,V\backslash V_1)$ can be synthesized at only nominal extra costs from the answers $Q(x,V)$ and $Q(x,V_1)$.

A typical example would be the question of counting the number of points within a certain rectangular range in the plane. For decomposable counting problems we do not have to perform deletions at the very moment they occur but we can buffer them for some time. We shall exploit the idea of [17] of putting deleted points in a separate ghost structure for a while and only cleaning up the main structure at suitably chosen later moments. We continue querying the main structure, but each time "subtract" the answer over the ghost structure. The following method will do to spread the work for clean-ups of the main structure, when the proportion of "deleted" elements has become too large, over a number of transactions that follow.

As primary structure we use a structure MAIN of the form described in section 2 and as ghost structure (to buffer deletions) we use a similar structure GHOST. Insertions are performed on MAIN and "deletions" are insertions in GHOST. When GHOST becomes too big we have to do a clean-up. Such a clean-up consists of building all points to be retained into a new static structure STAY. We spread the work for building STAY over a number of deletions that follow.

We choose to do a clean-up by the time

$$|GHOST| \geq \tfrac{1}{2}(|STAY| + |MAIN|)$$

($|S|$ means the number of points in S.) Let the "real" size of the old STAY and MAIN at the moment we start with the clean-up be n_0. We call the old STAY, MAIN and GHOST respectively S, M and G. So we have to build STAY out of M and S, discarding the elements of G. We spread this work over the next $\frac{1}{2}n_0$ deletions. New insertions will be performed on a new MAIN structure and new "deletions" will be inserted in a new GHOST structure (see figure 2). For query answering we use both the old structures S, M and G, and the new MAIN and GHOST, until the new STAY structure is completed. (If it is, then the old structures M, S and G are discarded.)

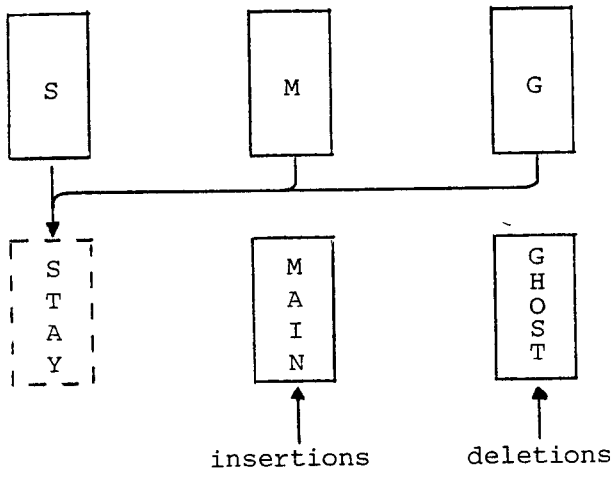

Figure 2.

We will now describe the algorithm more formally. We will use the procedure INSERT of section 2 again but we call it

```
proc INS(p, STRUCT, n);
```

in which p is the point to be inserted and STRUCT denotes the structure on which the insertion has to be performed. n is no longer a global variable. Instead of it we use i for the number of points in STAY and MAIN and d for the number of points in GHOST. Initially all structures are empty and i = d = 0. Performing an insertion consists of inserting the point in MAIN and increasing i.

```
proc INSERT(p);
{p is the point that must be inserted.}
begin
   INS(p, MAIN, i)
end of INSERT;
```

To perform a deletion we first have to insert the point in the GHOST structure. Sometimes we have to do some work on building STAY and, when the number of deleted points becomes too big we have to start building a new STAY. Let $building_s$ be a boolean, denoting that we are or, are not busy building STAY.

```
proc DELETE(p);
{p is the point that need be deleted.}
begin
   {insert p in GHOST and increase d.}
   INS(p, GHOST, d);
   {when we are busy building STAY out of S, M and G we have to do some
    work on this construction.}
```

```
if building_s
  then do 2P_S(n_0)/n_0 work on building STAY from S, M and G;
       if STAY is ready
         then  S := ∅; M := ∅; G := ∅;
               building_s := false
       fi
fi;
{when d has become too big we have to start building a new STAY structure.}
if d ≥ ½i
  then {change names and clear structures.}
       S := STAY;
       M := MAIN;
       G := GHOST;
       STAY := ∅;
       MAIN := ∅;
       GHOST := ∅;
       {update the number of points in STAY, MAIN and GHOST.}
       i := i-d;
       d := 0;
       {remember the number of points that will come in STAY.}
       n_0 := i;
       building_s := true
fi
end of DELETE;
```

Lemma 3.1. While STAY is being built (building$_s$ = true), DELETE will never call for a clean-up (i.e., $d < \frac{1}{2}i$ while STAY is being built).

Proof

At the moment we start building STAY (building$_s$ becomes true) $i = n_0$ and $d = 0$. With each deletion we do $2P_S(n_0)/n_0$ work on building STAY and because STAY will have size n_0, we are ready (and building$_s$ becomes false) after (at most) $\frac{1}{2}n_0$ deletions. Thus d remains $\leq \frac{1}{2}n_0$ while we are building STAY. During the building of STAY, i can only increase (when new insertions occur) and hence $d \leq \frac{1}{2}n_0 \leq \frac{1}{2}i$. At the moment d becomes $\frac{1}{2}i$ the old STAY must have been completed (possible during the same deletion) and we can indeed start building a new STAY-structure out of all points left.

□

Lemma 3.2. Every inserted point is in exactly one structure (STAY, MAIN, S or M) and every buffered deletion is in GHOST or in G.

Proof

This follows trivially from the way we rename and clear structures. □

From lemmas 3.1. and 3.2. it follows that the procedures INSERT and DELETE maintain the set correctly.

Theorem 3.3. Given a decomposable counting problem, one can dynamize it such that

$$Q_D(n) \leq (12\log n+12)\ Q_S(2n) + Q_S(4n)$$
$$I_D(n) \leq (\log n+2)\ P_S(2n)/2n$$
$$D_D(n) \leq (\log n+1)\ P_S(n)/n + P_S(2n)/n$$

Proof

We start doing a clean-up at the moment d becomes $\frac{1}{2}i$. It follows that, because the actual number of points at that moment is n_0, $|STAY| + |MAIN| = 2n_0$ and $|GHOST| = n_0$. Hence, during the construction of STAY $|S| \leq 2n_0$, $|M| \leq 2n_0$ and $|G| \leq n_0$. Because STAY is ready after $\frac{1}{2}n_0$ deletions $n \geq \frac{1}{2}n_0$ and hence $|S| \leq 4n$, $|M| \leq 4n$ and $|G| \leq 2n$. One easily verifies that always $|STAY| \leq 2n$, $|MAIN| \leq 2n$ and $|GHOST| \leq n$.

When we want to perform a query we have to perform it on STAY, MAIN and GHOST or on S, M, G, MAIN and GHOST. Hence, on one static structure of size at most 4n and on at most four dynamic structures, one of size 4n, two of size 2n and one of size n. From theorem 2.5. it follows that the query time is bounded by $3\log 4n\ Q_S(2n) + 6\log 2n\ Q_S(n) + 3\log n\ Q_S(\frac{1}{2}n) + Q_S(4n) \leq (12\log n+12)\ Q_S(2n) + Q_S(4n)$.

An insertion consists only of an insertion in MAIN and because $|MAIN| \leq 2n$, it follows from theorem 2.5. that $I_D(n) \leq (\log n+2)\ P_S(2n)/2n$. A deletion consists of an insertion in the GHOST-structure that has size at most n and $2P_S(n_0)/n_0$ work on building STAY. Because n_0 is bounded by 2n if follows that $D_D(n) \leq (\log n+1)\ P_S(n)/n + P_S(2n)/n$.

□

Because of the smoothness of Q_S and P_S we can rewrite the result of theorem 3.3. into

$$Q_D(n) = O(\log n)\ Q_S(n)$$
$$I_D(n) = O(\log n)\ P_S(n)/n$$
$$D_D(n) = O(\log n)\ P_S(n)/n$$

It follows that for decomposable counting problems we can achieve worst-case bounds on insertion and deletion time that are of the same order as the known average bounds.

4. Deletions in DD-searching problems.

There are a fair number of decomposable searching problems for which known static data structures actually do permit easy "deletions without rebalancing". A typical example is range query answering from range trees (cf. Bentley [2]). Overmars and van Leeuwen [12] showed that such problems can be dynamized with good average bounds on the insertion and deletion times. The following terminology is adapted from [12] and [20].

Definition. A decomposable searching problem Q, together with a static data structure S for it, is called a DD-searching problem if and only if S allows one to delete points, or to replace them by a dummy object that will not affect later queries, without causing an increase in the future deletion or query time of the structure.

Notation. $D_S(n)$ = the "deletion"-time in a static structure of n points (worst-case bound)

We assume that D_S is nondecreasing and "smooth". Note that, although it is possible to perform deletions in the static structure S of a DD-searching problem, the query time may stay the same while more and more points get deleted (the structure will go "out of balance").

To dynamize DD-searching problems, van Leeuwen and Maurer [20] gave routines directly based on the structure of Bentley [2]. Insertions are processed as usual and deletions are by replacing a point by an explicit or "implicit" dummy. By the time the proportion of dummies (as "buffered deletions") becomes too large, they clean up the structure by rebuilding it. Using the structure of section 2 as a start, instead of the structure of Bentley [2], will not do as such to get good worst-case bounds. For, in general, one cannot even process deletions from blocks under construction before they are completed. Yet we shall prove that there is a scheme for handling deletions in the structure of section 2.

To locate the blocks (one in use and possible one under construction) to which a point p we want to delete belongs, we add to the structure a dictionary DICT in which we keep this information for each point in the set. When we implement DICT as a balanced search tree we can perform such a search in O(log n). Let the block in use p is in be B_i^u. Performing the deletion on this block can

easily be done, but it is possible that we are also busy building B^c_{i+1} and p becomes part of it. In this case we cannot perform the deletion on B^c_{i+1} before the block is ready. Hence we have to buffer the deletion in a list BUF_{i+1}. We also have to make sure that, once B^c_{i+1} is ready, we have time "left" to perform the deletion of p. (This deletion will take $D_S(2^{i+1})$.) To have this time, we do $D_S(2^{i+1})$ more work on the construction of B^c_{i+1}. It follows that when B^c_{i+1} is built we still have $D_S(2^{i+1})$ time left to perform the deletion of p before B^c_{i+1} was assumed to be ready. In other words, by doing $D_S(2^{i+1})$ work on the construction of B^c_{i+1} we take care that the total time needed before B^c_{i+1} is ready for use (with the buffered deletions performed) is not increased. Therefore B^c_{i+1} will again be ready for use after 2^{i+1} insertions. We have shown that it is possible to perform deletions on the structure of section 2 in a way that does not increase the query time, nor the insertion time. But as more and more points get deleted, the total structure goes out of balance. Therefore, when the number of deleted points becomes too big we have to do a clean-up. This can be done very similar to the method of section 3, except that no GHOST structure is used anymore. A clean-up is required whenever the number of dummies gets equal to n_0, the "real" size of STAY and MAIN. When a clean-up is called for, the construction of a new structure STAY is initiated from the points to be retained from the old structures STAY and MAIN. We call these old structures S and M. As in section 3 we want that the new STAY is ready (and can take over from S and M) after at most $\frac{1}{2}n_0$ deletions. To this end, we do with each deletion $2P_S(n_0)/n_0$ work on building STAY. Insertions are performed in a new MAIN structure and for query answering we use S, M and MAIN. Deletions we perform in the block of S, M or MAIN the point belongs to, but it is possible that we are also busy building the point we want to delete in STAY. As noted before we cannot perform deletions in blocks under construction and therefore we have to buffer this deletion. For this task we use a buffer BUF_s. To save time for performing this deletion once STAY is built we again speed up the building of STAY by doing $D_S(n_0)$ extra work on the construction at this time. It follows that STAY will be ready after $\frac{1}{2}n_0$ deletions and S and M can be thrown away. We assume this takes only constant time. In general this is not true, because we have to update DICT but, without loss of generality, we can assume this time is included in the building time of STAY. We will now describe the insertion and deletion routines more formally. The meaning of $building_i$ and $building_s$ is somewhat different from the previous algorithms.

$building_i = 0$ means: not busy building

$building_i = 1$ means: busy building

$building_i = 2$ means: busy performing buffered deletions

Similar for $building_s$.

```
proc INSERT(p);
begin
   n := n+1;
   insert p in DICT;
   for i := ⌊log n⌋ downto 1 do
      begin
         if building_i = 1
           then  {busy building}
                 do P_S(2^i)/2^i work on building B_i^c from B_{i-1}^u[1] and B_{i-1}^u[2];
                 work_i := work_i+1;
                 if B_i^c becomes ready
                   then building_i := 2
                 fi
       elif building_i = 2
           then {busy performing buffered deletions}
                do P_S(2^i)/2^i work on performing deletions from BUF_i;
                work_i := work_i+1
         fi;
         if work_i = 2^i
           then {B_i^c must be ready now}
                 B_i^u[emp_i] := B_i^c;
                 do the necessary changes in blocknames and discard the old
                 blocks like in the insertion routine in section 2
         fi
      end;
      do the work in BA_0
end of INSERT;
```

```
proc DELETE(p);
begin
   d := d+1;
   search for p in DICT;
   perform the deletion of p in the appropriate block in use;
```

```
if p present in B_i^c
  then if building_i = 1
         then put p in BUF_i;
              do D_S(2^i) work on the construction of B_i^c;
              if B_i^c becomes ready
                then building_i := 2
              fi
         else perform the deletion of p from B_i^c
       fi
fi;
if p present in STAY
  then if building_S = 1
         then {busy building STAY}
              put p in BUF_S;
              do D_S(n_0) work on the construction of STAY out of S and M;
              if STAY becomes ready
                then building_S := 2
              fi
         else perform the deletion of p from STAY
       fi
fi;
delete p from DICT;
{do work on the construction of STAY (when needed).}
if building_S = 1
  then do 2P_S(n_0)/n_0 work on building STAY from S and M;
       if STAY becomes ready
         then building_S := 2
       fi
elif building_S = 2
  then do 2P_S(n_0)/n_0 work on performing deletions from BUF_S;
       if BUF_S becomes empty
         then M := ∅; S := ∅;
              (update DICT;)
              building_S := 0
       fi
fi;
{when the number of deletions becomes too big start building STAY.}
```

```
  if d ≥ ½n
    then M := MAIN; MAIN := ∅;
         S := STAY; STAY := ∅;
         n := n-d;
         n_0 := n;
         d := 0;
         building_S := 1
  fi
end of DELETE;
```

Lemma 4.1. B_i^c is ready for use after 2^i insertions and STAY is ready for use after $\frac{1}{2}n_0$ deletions.

Proof

The total time needed to build B_i^c is $P_S(2^i)$. With each insertion we do $P_S(2^i)/2^i$ work on its construction and hence B_i^c is ready for use after 2^i insertions, provided deletions do not increase the time needed to complete B_i^c. The only deletions that can influence this bound are deletions that need to be performed on B_i^c itself. Because we buffer these, each such deletion apparently increases the time needed to complete B_i^c by $D_S(2^i)$. But because we always do $D_S(2^i)$ work on the construction of B_i^c when another deletion is buffered, the increase of time is absorbed again.

The building of STAY takes $P_S(n_0)$. With each deletion we do $2P_S(n_0)/n_0$ work and hence we are finished after $\frac{1}{2}n_0$ deletions. As for B_i^c, it can be shown that deletions that need to be performed on STAY do not increase the time needed to finish the construction.

□

The lemmas 2.1. to 2.4., 3.1., 3.2. and 4.1. show the correctness of the structure and the algorithms.

Theorem 4.2. Given a DD-searching problem, one can dynamize it such that

$$Q_D(n) \leq (6 \log n+9)\ Q_S(2n) + Q_S(4n)$$
$$I_D(n) \leq (\log n+2)\ P_S(2n)/2n$$
$$D_D(n) \leq P_S(2n)/n + 2D_S(2n) + O(\log n)$$

Proof

One easily shows that always $|MAIN| \leq 2n$, $|STAY| \leq 2n$, $|M| \leq 4n$, $|S| \leq 4n$ and $n_0 \leq 2n$. A query consists of a query on MAIN and STAY or a query on MAIN, S and M. Hence, it takes at most $(3\log 2n)\ Q_S(n) + Q_S(4n) + (3\log 4n)\ Q_S(2n) \leq (6\log n+9)\ Q_S(2n) + Q_S(4n)$.

An insertion is always performed on MAIN. To perform an insertion we do at most $P_S(2^i)/2^i$ work in each bag BA_i like in section 2. Moreover we have to do O(log n) work for the dictionary update. Hence the total time needed is bounded by (log 2n+1) $P_S(2n)/2n$ + O(log n).

With a deletion we have to do O(log n) work on searching and updating DICT, $D_S(2n)$ work on deleting the point from a block in use, $D_S(2n)$ time on the construction of STAY or on a block under construction in MAIN and $2P_S(n_0)/n_0$ work on the construction of STAY. Hence the total time needed is bounded by $P_S(2n)/n$ + $2D_S(2n)$ + O(log n).

□

Because of the smoothness of Q_S, P_S and D_S we can rewrite this result into

$$Q_D(n) = O(\log n)\ Q_S(n)$$
$$I_D(n) = O(\log n)\ P_S(n)/n$$
$$D_D(n) = O(P_S(n)/n + D_S(n) + \log n)$$

Hence the worst-case bounds are of the same order as the known average bounds for DD-searching problems (compare Overmars and van Leeuwen [12]).

5. Extensions and applications.

Overmars and van Leeuwen [12] have defined another class of decomposable searching problems, called MD-searching problems, for which a fast merge technique for building blocks can be used to save a log-factor in the average insertion time. The same savings can be made in the worst-case bounds, when these problems are dynamized as above. In a recent paper Mehlhorn and Overmars [10] (as an extension of Overmars and van Leeuwen [13]) describe a method to obtain all optimal dynamizations of decomposable searching problems. Their general dynamization scheme consists of bags also but they allow the number of blocks in a bag to be different for different bags (and also dependent on the number of points in the set) or that the size of the block in a bag may change. Their bounds are average results. The techniques of the present paper can be applied to prove the same bounds as worst-case times.

In a number of ways theorem 4.2. "settles" the problem of finding good worst-case dynamizations. It is extremely useful to obtain good bounds on problems that were not adequately dynamized or that were only dynamized by means of complex ad-hoc constructions before, as the following applications may show.

a. Range querying.

Willard [22] (see also Lueker [8]) first described a fully dynamic structure for the d-dimensional range query problem. His structure can be built

in $P_S(n) = O(n \log^{d-1} n)$ and achieves

$Q(n) = O(\log^d n)$ (+the time required to print the answers, which we ignore)

$I(n) = O(\log^d n)$

$D(n) = O(\log^d n)$

It can be shown that range querying using this structure as a static structure is a DD-searching problem achieving $D_S(n) = O(\log^{d-1} n)$ (cf. [20]). Applying theorem 4.2., we get

Theorem 5.1. There is a dynamic data structure D for the d-dimensional range query problem such that

$Q_D(n) = O(\log^{d+1} n)$

$I_D(n) = O(\log^d n)$

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

($d \geq 2$) as worst-case bounds.

Hence at the cost of a factor log n in query time we have saved a factor log n in deletion time.

b. Quad- and k-d trees.

Quad-trees were introduced by Finkel and Bentley [6] as a data structure for answering various types of queries on 2- and higher dimensional point sets. Overmars and van Leeuwen [14] introduced a modified version of it called psuedo quad-trees and proved that on such structures both insertions and deletions can be processed in $O(\log^2 n)$ average time. It is easily seen that most problems using quad- or pseudo quad-trees are DD-searching problems with $D_S(n) = O(\log n)$ (cf. [12]). As $P_S(n) = O(n \log n)$ we can apply theorem 4.2. to get

Theorem 5.2. Given "any" DD-searching problem using quad trees, one can dynamize it such that

$Q_D(n) = O(\log n).Q_S(n)$

$I_D(n) = O(\log^2 n)$

$D_D(n) = O(\log n)$

Thus, at the cost of a factor of log n in query time we get previously known average bounds as worst-case bounds, with an even better bound for the deletion time.

Another data structure for queries on d-dimensional point sets, k-d trees, was introduced by Bentley [3]. Overmars and van Leeuwen [14] again gave a

dynamic version of it, called pseudo k-d trees, with $O(\log^2 n)$ average insertion and deletion time. Like for quad trees, theorem 4.2. shows

Theorem 5.3. Given "any" DD-searching problem using k-d trees, one can dynamize it such that

$$Q_D(n) = O(\log n).Q_S(n)$$
$$I_D(n) = O(\log^2 n)$$
$$D_D(n) = O(\log n)$$

c. Halfspaces and maximal elements.

Overmars and van Leeuwen [15,16] described a structure for determining whether a point lies in the common intersection of a set of 2-dimensional half-spaces. Their structure is able to process both insertions and deletions and yields

$$Q(n) = O(\log n)$$
$$I(n) = O(\log^2 n)$$
$$D(n) = O(\log^2 n)$$

It can easily been shown that the problem is both a DD-searching problem with $D_S(n) = O(\log^2 n)$ as a MD-searching problem (cf. [12]). Hence we can dynamize it such that

$$Q_D(n) = O(\log^2 n)$$
$$I_D(n) = O(\log n)$$
$$D_D(n) = O(\log^2 n)$$

Hence we can decrease the insertion time by increasing the query time. Similar results can be obtained for the question whether a point is maximal to a set of points (cf. [12,15]).

d. Segment trees

In a variety of problems (including many "rectangle" problems) it is required to maintain a collection of n segments on a line, such that one can efficiently determine to what segments a given point belongs. Bentley [1] and Bentley and Wood [4] introduced a data structure (with a lot of internal administration) called a segment tree, which enables them to answer such queries in only O(log n) time (plus the time required to print the answers which we again ignore). The segment tree has been used in a large number of constructions, typically for DD-searching problems such as querying for point containment in rectangles. As $P_S(n) = O(n \log n)$ and $D_S(n) = O(\log n)$ one can immediately conclude

Theorem 5.4. Given a DD-searching problem Q using segment trees, one can dynamize it such that

$$Q_D(n) = O(\log n)\ Q_S(n)$$
$$I_D(n) = O(\log^2 n)$$
$$D_D(n) = O(\log n)$$

Very recently, Edelsbrunner [5] obtained a direct method to maintain dynamic segment trees with $Q_D(n) = O(Q_S(n))$, $I_D(n) = O(\log n)$ and $D_D(n) = O(\log n)$.

e. Nearest neighbor searching.

Recently, Overmars [11] described a data structure for the nearest neighbor searching problem based on the Voronoi diagram (see Shamos [18], Shamos and Hoey [19] and Kirkpatrick [7]) achieving

$$Q(n) = O(\log n)$$
$$I(n) = O(n)$$
$$D(n) = O(n)$$

Using this structure, the nearest neighbor searching problem is a DD-searching problem with $P_S(n) = O(n \log n)$ and $D_S(n) = O(n)$. Hence, we can apply theorem 4.2. and transform it into a structure yielding

$$Q(n) = O(\log^2 n)$$
$$I(n) = O(\log^2 n)$$
$$D(n) = O(n)$$

6. References.

[1] Bentley, J.L., Algorithms for Klee's rectangle problems, unpubl. notes, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, 1977.

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

[3] Bentley, J.L., Multidimensional binary search trees used for associative searching, Comm. of the ACM 18 (1975) pp. 509-517.

[4] Bentley, J.L. and D. Wood, An optimal worst-case algorithm for reporting intersections of rectangles, IEEE Transactions on Computers C-29 (1980) pp. 571-577.

[5] Edelsbrunner, H., Dynamic data structures for orthogonal intersection queries, Bericht 59, Inst. f. Informationsverarbeitung, TU Graz, 1980.

[6] Finkel, R.A. and J.L. Bentley, Quad trees: a data structure for retrieval on composite keys, Acta Informatica 4 (1974) pp. 1-9.

[7] Kirkpatrick, D.G., Efficient computation of continuous skeletons, Proc. of the 20th Annual IEEE Symp. on Foundations of Computer Science, 1979, pp. 18-27.

[8] Lueker, G.S., A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems, Techn. Rep., Dept. of Computer Science, University of California, Irvine, 1978.

[9] Maurer, H.A. and Th. Ottmann, Dynamic solutions of decomposable searching problems, Bericht 33, Inst. f. Informationsverarbeitung, TU Graz, 1979.

[10] Mehlhorn, K. and M.H. Overmars, Optimal dynamization of decomposable searching problems, Techn. Rep., Fachbereich 10, Universität des Saarlandes, Saarbrücken, 1980.

[11] Overmars, M.H., Dynamization of order decomposable set problems, Techn. Rep. RUU-CS-80-9, Dept. of Computer Science, University of Utrecht, 1980.

[12] Overmars, M.H. and J. van Leeuwen, Two general methods for dynamizing decomposable searching problems, Techn. Rep. RUU-CS-79-10, Dept. of Computer Science, University of Utrecht, 1979. (To appear in Computing.)

[13] Overmars, M.H. and J. van Leeuwen, Some principles for dynamizing decomposable searching problems, Techn. Rep. RUU-CS-80-1, Dept. of Computer Science, University of Utrecht, 1980. (To appear in Inform. Proc. Lett.)

[14] Overmars, M.H. and J. van Leeuwen, Dynamic multidimensional data structures based on quad- and k-d trees, Techn. Rep. RUU-CS-80-2, Dept. of Computer Science, University of Utrecht, 1980.

[15] Overmars, M.H. and J. van Leeuwen, Maintenance of configurations in the plane, Techn. Rep. RUU-CS-79-9, Dept. of Computer Science, University of Utrecht, 1979/1980.

[16] Overmars, M.H. and J. van Leeuwen, Notes on maintenance of configurations in the plane, Techn. Rep. RUU-CS-80-5, Dept. of Computer Science, University of Utrecht, 1980.

[17] Saxe, J.B. and J.L. Bentley, Transforming static data structures into dynamic structures, Proc. 20th Annual IEEE symp. on Foundations of Computer Science, 1979, pp. 148-168.

[18] Shamos, M.I., Computational geometry, to be published by Springer-Verlag.

[19] Shamos, M.I. and D. Hoey, Closest-point problems, Proc. of the 16th Annual IEEE Symp. on Foundations of Computer Science, 1976, pp. 151-162.

[20] Van Leeuwen, J. and H.A. Maurer, Dynamic systems of static data structures, Bericht 42, Inst. f. Informations verarbeitung, TU Graz, 1980.

[21] Van Leeuwen, J. and D. Wood, Dynamization of decomposable searching problems, Inform. Proc. Lett. 10 (1980) pp. 51-56.

[22] Willard, D.E., The super B-tree algorithm, TR-03-79, Aiken Computation Lab., Harvard University, 1979.