TWO GENERAL METHODS FOR DYNAMIZING DECOMPOSABLE SEARCHING PROBLEMS

Mark H. Overmars and Jan van Leeuwen

RUU-CS-79-10

November 1979

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

TWO GENERAL METHODS FOR DYNAMIZING DECOMPOSABLE SEARCHING PROBLEMS

Mark H. Overmars and Jan van Leeuwen

Technical Report RUU-CS-79-10

November 1979

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

TWO GENERAL METHODS FOR DYNAMIZING DECOMPOSABLE SEARCHING PROBLEMS

Mark H. Overmars and Jan van Leeuwen

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

Abstract. We define two classes of Decomposable Searching Problems and consider ways of efficiently dynamizing them. For the first class, DD, we show that both insertions and deletions can be processed efficiently. For the second class, MD, we exploit a merge technique to obtain better insertion times. We also give a number of examples of problems to which the methods apply, including the dynamic maintenance of quad-trees and of the common intersection of finitely many halfspaces in the plane.

Keywords and phrases. decomposable searching problem, dynamization, merging, quad-trees, k-d trees, halfspaces, maximal elements.

1. Introduction.

In the past two years, several efforts were made to develop general methods for turning static solutions to problems into dynamic ones. Bentley [1] and later Saxe and Bentley [7] made the important observation that a general approach to "dynamization" is especially relevant to the class of so-called decomposable searching problems. A searching problem can be viewed as a problem in which one asks a question about an arbitrary object x of type T_1 and a (static or dynamic) set of objects (from now on called points) of type T_2, with an answer of type T_3. We can denote such a query as:

$$Q: T_1 \times 2^{T_2} \rightarrow T_3$$

For instance, in a Member query, T_1 and T_2 are the same and T_3 is boolean.

Definition 1.1. A searching problem $Q: T_1 \times 2^{T_2} \rightarrow T_3$ is called decomposable if there exists an efficiently computable operator $\square$ on the elements of T_3, satisfying:

$$\forall A,B \in 2^{T_2}, x \in T_1 \qquad Q(x, A \cup B) = \square(Q(x,A), Q(x,B))$$

For instance, a Member query is decomposable ($\square$ = or). Saxe and Bentley [7] presented several general methods to build dynamic structures out of static structures to deal with decomposable searching problems for a dynamic point set. The general approach is possible because the set of points, to which a

decomposable query applies, can be split into subsets and the answer of the query can be derived at only nominal extra cost from the answers to the same query on each of the subsets (using $\square$). We will briefly mention one of their methods below.

A disadvantage of the methods presented by Bentley [1] and Saxe and Bentley [7] is that it appears to be very hard to perform deletions. They suggest a brute force technique which always works and which occasionally leads to a very reasonable efficiency. Later, van Leeuwen and Wood [8] (see also [4]) provided a technique for maintaining a balanced partitioning of a set dynamically, to handle insertions and deletions in another way. In this paper we shall study a new method which will usually support deletions more efficiently. In section 2 we will define a subset of decomposable searching problems that have a static structure in which deletions are accomodated in some way (while insertions not necessarily are) and show how a dynamic structure can be devised for this class of problems by means of which insertions and deletions can be dealt with rather efficiently, while reasonable query times are maintained.

Saxe and Bentley [7] mention that for some special problems "merge techniques" can be exploited to obtain a better dynamic structure. In section 3 we shall characterize a class of decomposable searching problems for which a general merge technique can be used to obtain faster insertion times. In section 4 we shall give some examples of important searching problems to which our results apply and for which, therefore, dynamic structures can be given which are more efficient than previous dynamizations known.

Saxe and Bentley [7] developed three different types of dynamization methods for decomposable searching problems. One of these methods will be described here in some detail, because our further techniques build on it. To compare structures we will use the following quantities:

Definition 1.2. Let S be a static structure, containing N points.

$Q_S(N)$ = Time required for a query on S.

$S_S(N)$ = Storage required for S.

$P_S(N)$ = Preprocessing time required to build S.

Let D be a dynamic structure, containing N points.

$Q_D(N)$ = Time required for a query on D.

$S_D(N)$ = Storage required for D.

$P_D(N)$ = Time required to build D by N insertions.

$I_D(N)$ = Average time required for an insertion.

$D_D(N)$ = Average time required for a deletion (when applicable).

We assume that Q_S is nondecreasing and that S_S and P_S "grow at least linearly", i.e., are functions f for which $\frac{f(x)}{x}$ is nondecreasing.

The usual type of dynamization method consists of building a dynamic structure out of different size static structures for the problem. This is feasible because the answer to a query over the whole dynamic structure can be combined out of the answers of the query over the separate static structures it is built from (by applying □). The dynamic structure D Saxe and Bentley [7] (also Bentley [1]) considered consists of a row of static structures $B_0, B_1, B_2, \ldots$. Every B_k is either empty or contains 2^k points. Hence, for every N there is precisely one way of filling the B's. For instance when N=26 the structures B_1, B_3 and B_4 are filled. Building the dynamic structure by inserting points $p_1, p_2, \ldots$ proceeds for each point p_i by a) finding the first k such that B_k is empty, b) unbuilding $B_0, \ldots, B_{k-1}$ and c) building B_k out of p_i and the points from the $B_0, \ldots, B_{k-1}$ just unbuilt.

Theorem 1.1. For all N $S_D(N) \leq S_S(N)$.

Proof

Let $N = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_k}$ for $i_1 < i_2 < \ldots < i_k$. Because S_S grows at least linearly, $S_D(N) = S_S(2^{i_1}) + S_S(2^{i_2}) + \ldots + S_S(2^{i_k}) \leq$

$$\left(\frac{1}{2^{i_k - i_1}} + \frac{1}{2^{i_k - i_2}} + \ldots + \frac{1}{2^{i_k - i_{k-1}}} + 1 \right) S_S(2^{i_k}) \leq 2 S_S(2^{i_k}).$$

Because $2^{i_k} \leq N$ this is of the same order as $S_S(N)$.

□

Theorem 1.2. For all N $Q_D(N) \leq Q_S(N) . O(\log N)$.

Proof

When D contains N points, the biggest k for which B_k is filled is $k = \lfloor \log N \rfloor$. Hence the answer to a query over D can be derived in O(log N) from the answers to the query over each B_i with $0 \leq i \leq \lfloor \log N \rfloor$. A query over a single B_i costs $Q_S(2^i)$. Hence the cost for a query over D can be estimated as

$$Q_D(N) \leq \sum_{i=0}^{\lfloor \log N \rfloor} Q_S(2^i) + O(\log N).$$

Because Q_S is nondecreasing $Q_S(2^i) \leq Q_S(N)$ $(0 \leq i \leq \lfloor \log N \rfloor)$ and hence $Q_D(N) \leq Q_S(N) . O(\log N)$.

□

It can be shown that $Q_D(N) = Q_S(N)$ when $Q_S(N) = \Omega(N^\varepsilon)$ for any positive ε.

Theorem 1.3. For all N $\quad P_D(N) \leq P_S(N).O(\log N)$.

Proof

The largest static substructure filled after N insertions is $B_{\lfloor \log N \rfloor}$. Hence every point has been built at most once into each B_i with $0 \leq i \leq \lfloor \log N \rfloor$. Dividing the cost for building a B_i over its points makes for $P_S(2^i)/2^i$ per point. Hence the total cost per insertion is

$$I_D(N) \leq \sum_{i=0}^{\lfloor \log N \rfloor} P_S(2^i)/2^i$$

and the cost for building D by N insertions is

$$P_D(N) \leq N \sum_{i=0}^{\lfloor \log N \rfloor} P_S(2^i)/2^i.$$

Because P_S grows at least linearly, $P_S(2^i)/2^i \leq P_S(N)/N \quad (0 \leq i \leq \lfloor \log N \rfloor)$ and hence $P_D(N) \leq P_S(N).O(\log N)$.

□

It can be shown that $P_D(N) = P_S(N)$ when $P_S(N) = \Omega(N^{1+\varepsilon})$ for any positive ε. With the given method to perform insertions in mind, we shall consider ways to extend it and to accomodate deletions efficiently as well.

2. Accomodating deletions.

A disadvantage of the dynamic structure described above, is the impossibility of efficiently deleting points. For instance, after 2^k insertions the dynamic structure consists of a single B_k. Deleting a point means splitting B_k into $B_0, B_1, \ldots, B_{k-1}$, inserting one point means rebuilding B_k out of $B_0, B_1, \ldots, B_{k-1}$, etc. and this will usually cost an extraordinary amount of time.

Saxe and Bentley [7] suggest a solution to this deficiency for a special subclass of the decomposable searching problems. They build a second dynamic structure containing the deleted points. When this structure becomes big (approximately half the size of the first structure) they destroy both and build a completely new dynamic structure of all points which should remain. The problems that can be handled with this structure must satisfy

$$\forall A \in 2^{T_2}, B \subset A, x \in T_1 \quad Q(x, A/B) = \Delta(Q(x,A), Q(x,B))$$

for some efficiently computable operator Δ on the elements of T_3.

We will show how the dynamic structure of section 1 can be modified to make it possible to process deletions for another subclass of decomposable searching problems. We will define this class by putting conditions on the

static structures known for the problems. This might seem odd, because a problem class should be independent of the existence of certain static structures. On the other hand, the dynamic structures built for these problems are highly dependent on the static structures known for them.

Definition 2.1. A decomposable searching problem R, together with its static structure S, is called Deletion Decomposable (DD) iff, whenever S contains N points, a point can be deleted from S in time $D_S(N)$ without increasing the query time, deletion time and storage required for S.

Note that the query time may stay the same as more and more points are deleted (the structure may get out of "balance"). We assume that $D_S(N)$ is non-decreasing.

Again the structure we propose for dynamizing DD-searching problems is built of static structures $B_0, B_1, B_2, \ldots$, but now we impose the following weaker conditions:

a) $\lfloor 2^{k-2} \rfloor < |B_k| \leq 2^k$ or B_k is empty (the notation $|B|$ is used for the number of points contained in B).

b) The query time for B_k is at most $Q_S(2^k)$.

c) Whenever a B_k is built, it contains at least $\lfloor 2^{k-1} \rfloor + 1$ points.

Because we want to handle deletions, we must be able to locate the B_k an arbitrary point belongs to efficiently. To this end we add to the dynamic structure a dictionary T that contains the required information for each point currently in the set. We also keep with every point a pointer to its location in T, to achieve fast update times when we rebuild B's. We assume that T itself requires only linearly storage and is fully dynamic. Think of a balanced search tree or some sort of extendible hashing for it.

Definition 2.2. The time to search, insert or delete in a dictionary T of N points is F(N).

Insertions are processed in almost the same way as in the old dynamic structure, described in section 1. When we want to insert a new point p, we first determine the smallest k with B_k empty. If $|B_0| + |B_1| + \ldots + |B_{k-1}| + 1 > 2^{k-1}$, then we build a B_k out of the points of $B_0, B_1, \ldots, B_{k-1}$ and p. Otherwise we build a B_{k-1} out of the points (there always are more than 2^{k-2} points). We might also have to update the values recorded for the points of $B_0, \ldots, B_{k-1}$ in T and must insert p in T. One easily verifies that the resulting structure again satisfies the conditions.

When we must delete a point p, we first determine the substructure B_k it is part of, by a search on T. We delete p from B_k, and update the dictionary. If

$|B_k| > 2^{k-2}$ we are done (the structure still satisfies a), b) and c)). Otherwise, if $|B_k| = 2^{k-2}$ (it cannot be less), then we have to rebuild substructures in the following manner:

1) If B_{k-1} is not empty, then we take the points of B_{k-1} and B_k together. If $|B_{k-1}| > 2^{k-2}$, then we build a B_k out of them, otherwise we build a B_{k-1}.
2) If B_{k-1} is empty, but B_{k-2} is not, then we build a B_{k-1} out of the points in B_k and B_{k-2} ($|B_k| + |B_{k-2}| > 2^{k-2}$).
3) If both B_{k-1} and B_{k-2} are empty, then we build a B_{k-2} out of the points of B_k.

One can easily see that all conditions are fulfilled.

Theorem 2.1. $Q_{DD}(N) \leq Q_S(4N).O(\log N)$.

Proof

When the structure contains N points, the biggest k for which B_k is non-empty is at most $k = \lfloor \log N \rfloor + 2$. Hence the answer to a query can be derived in $O(\log N)$ from the query over the B_i $(0 \leq i \leq \lfloor \log N \rfloor + 2)$. A query over B_i takes at most $Q_S(2^i)$, so the total cost for a query is

$$Q_{DD}(N) \leq \sum_{i=0}^{\lfloor \log N \rfloor + 2} Q_S(2^i) + O(\log N).$$

Because Q_S is nondecreasing $Q_S(2^i) \leq Q_S(4N)$ $(0 \leq i \leq \lfloor \log N \rfloor + 2)$. Hence, the total query time $Q_{DD}(N) \leq Q_S(4N).O(\log N)$.

□

Again, it can be shown that $Q_{DD}(N) = Q_S(4N)$ when $Q_S(N) = \Omega(N^\varepsilon)$ for any positive ε. Normally $Q_S(4N)$ will be of the same order as $Q_S(N)$ and the query time $Q_{DD}(N) \leq Q_S(N).O(\log N)$.

Theorem 2.2. $S_{DD}(N) \leq S_S(4N)$.

Proof

Because deleting a point from a static structure does not increase the storage required, every substructure containing n points needs at most $S_S(4n)$ storage. Since S_S grows at least linearly, the total amount of storage is $S_{DD}(N) \leq S_S(4N)$ by the same argument as in theorem 1.1.

□

Normally $S_S(4N)$ will be of the same order as $S_S(N)$ and theorem 2.2. shows that in this case the dynamic structure will require about as much storage as a static structure would.

It is somewhat more difficult to prove the time bounds for insertions and deletions. We have to divide the costs for the occasional reconstruction of substructures over the insertions and the deletions.

Theorem 2.3. $I_{DD}(N) \leq P_S(N)/N.O(\log N) + F(N)$
$D_{DD}(N) \leq D_S(N) + P_S(N)/N + F(N)$

Proof

There are some special costs for all insertions and deletions alike. When we insert a point p we must also insert it in T, which costs F(N). When we want to delete a point p, we have to find the substructure it is in (by a search on T), we have to delete p from its structure, and we have to delete p from T. The costs are F(N), $D_S(N)$ (certainly a sufficient bound) and again F(N), respectively. The cost for the rebuilding of substructures occasionally required afterwards must be averaged over the insertions or deletions which led up to it.

Let us first consider the reconstruction of B_k structures possibly following each deletion (and thus included in the cost for it). Very often nothing will have to be done, because a B_k always starts off with at least $2^{k-1}+1$ points when it is built and no rebuilding will be called for (as an action following a deletion) unless sufficiently many deletions have taken place after its latest creation to bring $|B_k|$ down to 2^{k-2}. It will obviously require at least 2^{k-2} deletions from a fresh B_k to let this happen. The reconstruction called for will produce a B_{k-2}, a B_{k-1} or a B_k and will cost at most $P_S(u_k)$ for some $u_k \leq \min(2^k, N)$. It follows that the average cost for reconstruction immediately after a deletion is bounded by $P_S(u_k)/2^{k-2}$, which is $O(P_S(N)/N)$ because P_S grows at least linearly. The total average cost spent on a deletion can therefore be estimated as

$$D_{DD}(N) \leq D_S(N) + F(N) + P_S(N)/N$$

in order of magnitude.

The average cost for reconstruction after each single insertion is harder to estimate. We shall simplify the matter conceptually by just determining the average time spent on reconstruction during all first t transactions, regardless of whether it were insertions or deletions. Because this average will come out to a higher amount than it was for deletions separately and there obviously must be more insertions than deletions among the first t transactions, it means that the average found must be accounted for by an average over the insertions only that is of the same order of magnitude.

Consider the slot for a B_k structure, for any fixed k. We shall first estimate the average cost of all transactions if we only count the reconstruction

activities at this particular slot. The first time the slot gets filled, it must be because at least 2^{k-1} earlier insertions have filled up all preceding slots. (Note that many deletions and more insertions may have occurred as well.) It means that the first time a B_k is built (necessarily as the result of an insertion), the cost for doing so can be charged to at least 2^{k-1} earlier transactions (insertions even). Keep watching this slot after a B_k was just built and consider the interval of time before another B_k gets built. There can be two cases when this next B_k is built:

case (i) The B_k gets built as the result of a deletion.

This was discussed before and can happen only when one partner in the reconstruction comes from this slot (i.e. is the old B_k) or the next or the one after that and has lost at least half of its number of points since it was last created. The cost for reconstruction can thus be charged to $\geq 2^{k-2}$ deletions that were not charged to earlier.

case (ii) The B_k gets built as the result of an insertion.

This time the argument is more subtle. Let the reconstruction be called for at the moment τ. Observe from the way k is chosen after an insertion, that there must be $B_{k'}$ structures present for all $k' < k$, i.e., in all preceding slots. It is useful to refer back to the last moment $\tau' < \tau$ that a B_k was created as the result of an insertion (such a moment exists). At that time the slots before B_k were all emptied out. And now (at time τ) they are all filled up again. It can be that all points in the current B_{k-1} structure are all due to renewed insertions which took place after time τ'. In this case we can charge the cost for reconstructing the B_k to the $\geq 2^{k-2}$ insertions that led to the B_{k-1} (which were not charged to earlier). It can also be that among the points in the current B_{k-1} there are which migrated here from a higher numbered slot (i.e. which got here because sometime between τ' and now it happened that a B_k or B_{k+1} got dumped into lower numbered slots). It will now pay to look at the B_{k-2} structure currently present. Again, it can be that the points of this B_{k-2} structure are all due to renewed insertions which took place after time τ'. It gives at least 2^{k-3} insertions again which were not charged to earlier and which can be used to charge the present reconstruction of a B_k to. However, it can also be that some (or many) points of the B_{k-2} structure can be traced back to a moment (before τ') that they were in higher numbered slots $B_{k''}$ for $k'' \geq k$. These points can have gotten back past the B_{k-1} slot only if after time τ' (i.e. between τ' and now) some B_{k-1} or B_k lost at least half the number of its points and was "thrown back". It follows that we can identify yet another collection of $\geq 2^{k-3}$ deletions which were not charged to earlier to which the reconstruction of the B_k can be charged. Note that when the B_k gets constructed

all slots $B_{k'}$, for $k' < k$ are emptied out again (in this case) and the transactions we charged the cost to now will not be charged to ever again.

We conclude that in all cases the construction of a new B_k can be charged to another set of $\geq 2^{k-3}$ transactions every time. It follows that a new B_k can be built at most $t/2^{k-3}$ times, which means a total reconstruction time of at most $t/2^{k-3}.P_S(u_k)$ for $u_k = \min(2^k, N)$.

Taking the results of the analysis together for all slots, it follows that the total time spent on rebuilding substructures during the first t transactions must be bounded by

$$\sum_{k=1}^{\log n} \frac{t}{2^{k-3}} P_S(u_k) = 8t. \sum_{k=1}^{\log n} \frac{P_S(u_k)}{2^k} = O(t. \frac{P_S(N)}{N} . \log N)$$

(using that P_S grows at least linearly). The costs come out to an average of $\frac{P_S(N)}{N} . \log N$ per transaction. By our earlier argument, this average must be of the same magnitude as the average reconstruction time after insertions alone. As insertions do not give rise to any additional costs except for an update of the dictionary, the average insertion time will be no higher than

$$I_{DD}(N) \leq F(N) + \frac{P_S(N)}{N} . \log N$$

□

3. Merge method.

For many decomposable searching problems dynamic structures can be given, faster than the ones described by Saxe and Bentley [7]. These structures are often based on the possibility of building a new structure more efficiently by merging static structures than by rebuilding it from scratch. To formalize this concept we define another subclass of decomposable searching problems. Again we do this by putting conditions on the static structure known for solving the problem.

Definition 3.1. A decomposable searching problem R, together with its static structure S, is called Merge Decomposable (MD) iff it is possible to split the cost for building S out of the points $p_1 \cdots p_N$ in O(N log N) for ordering $p_1 \cdots p_N$ and only O(N) additional steps for building the actual structure S from the ordered set of points.

We will show how the dynamic structure of section 1 can be modified so as to be able to merge static substructures of MD-searching problems efficiently. The structure we use is almost the same as the structure in section 1. We only add to every substructure B_k a doubly linked list L_k of the points in B_k

in sorted order. When we want to insert a new point p we first determine the smallest k with B_k empty. Then we merge p and $L_0, L_1, \ldots, L_{k-1}$ to obtain L_k. After that, we build B_k using L_k.

Theorem 3.1. $Q_{MD}(N) \leq Q_S(N).O(\log N)$.

Proof

The lists added do not effect the query time, hence the result follows from theorem 1.2.

□

Theorem 3.2. $S_{MD}(N) \leq S_S(N)$.

Proof

Because every element is contained in only one list, the total amount of storage added is O(N). Hence, because $S_S(N)$ grows at least linearly, the bound follows.

□

Theorem 3.3. $P_{MD}(N) \leq P_S(N)$.

Proof

$P_{MD}(N)$ depends heavily on the way we merge the lists. When we need to build L_k out of p, $L_0, L_1, \ldots, L_{k-1}$ we proceed as follows. Let $\cup$ denote the merge operator and let L be a temporary list.

```
L:=p ∪ L0
FOR i:=1 TO k-1 DO L:=L ∪ Li
Lk:=L
```

So we merge the L_i in increasing order. The costs are

$$2 + \sum_{i=1}^{k-1} 2.2^i = 2^{k+1} - 2.$$

Distributed over the points, this makes for O(1) per point. Building a B_k from an L_k also costs $O(2^k)$, which is O(1) per point. Every point is built into at most O(log N) structures, hence the total cost per point is O(log N). So $P_{MD}(N) \leq O(N \log N) = P_S(N)$.

□

So we have a dynamic structure which, at an extra cost of only a factor of O(log N) in query time, can handle insertions in O(log N). Note that, if $Q_S(N) = \Omega(N^\varepsilon)$ for any positive ε, we do not need to pay the penalty in query time.

If a problem is a MD-, as well as a DD-searching problem, then we can combine the two dynamic structures described. We use the structure of section 2 that was able to handle deletions, and add again to each B_k a doubly linked list L_k that contains the points of B_k in sorted order. The only problem that arises is that we must be able to delete arbitrary points from the lists. This is easily solved by adding to each point p in the dictionary T a pointer to its position in the appropriate list. Because the lists are double linked, updating them is easy. Adding the lists to the structure does not change the query time, and the amount of storage required increases by only a constant factor. The time needed for rebuilding again decreases and becomes O(log N) per insertion and only O(1) per deletion. The cost for searching and updating the dictionary stays the same. So we have shown:

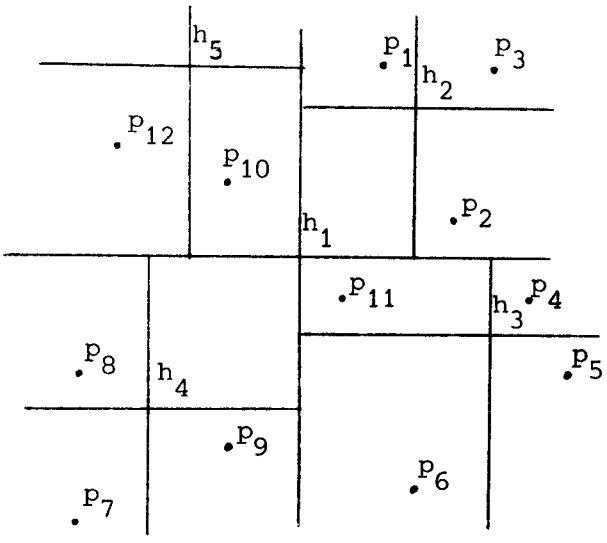
Theorem 3.4. If a problem R together with its static structure S, is both a MD- and a DD-searching problem, then there exists a dynamic structure DMD for R with

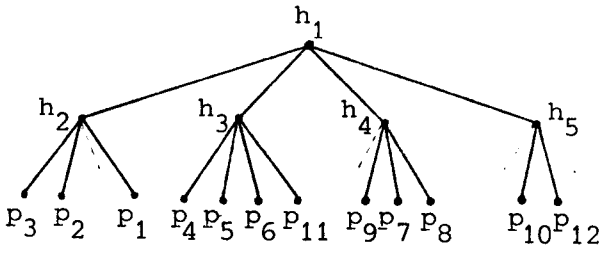
$$Q_{DMD}(N) \leq Q_S(4N).O(\log N)$$
$$S_{DMD}(N) \leq S_S(4N)$$
$$I_{DMD}(N) \leq O(\log N) + F(N)$$
$$D_{DMD}(N) \leq D_S(N) + F(N).$$

4. Examples.

To show how useful the dynamization methods described in the sections 2 and 3 can be, we shall present some important examples of DD- and MD-searching problems.

a) Finkel and Bentley [3] described a structure for answering queries about points in a d-dimensional space (especially range queries), which they called a quad-tree. We will work with a slightly changed version of it, called a pseudo quad-tree. (See Overmars and van Leeuwen [6] for some properties of pseudo quad-trees.) A pseudo quad-tree differs from an ordinary quad-tree only by allowing nodes of the tree to be in an arbitrary point rather than in a point of the set. The leaves of the pseudo quad-tree are the points of the set. A typical example of a 2-dimensional pseudo quad-tree is shown in the following diagram ($p_1 \ldots p_{12}$ are the points of the set, $h_1 \ldots h_5$ are the arbitrary points that serve as nodes for the tree)





Overmars and van Leeuwen [6] show that when the dimension d is a constant a pseudo quad-tree of N points with depth at most $^{d+1}\log N$ can be built in $O(N \log N)$. Because the points only exist at the leaves of the pseudo quad-tree a point can be deleted in $O(^{d+1}\log N)$ by searching it and throwing it away. This does not increase the query time, nor the amount of storage. (To keep a pseudo quad-tree balanced may take a lot of time. See [6] for a solution to this problem.) Decomposable searching problems that use a pseudo quad-tree are a prime example of DD-searching problems. Using for T a balanced search tree we can assert a dynamic structure with

$$Q_{DD}(N) \leq Q_S(4N).O(\log N)$$
$$S_{DD}(N) = O(N)$$
$$I_{DD}(N) = O(\log^2 N)$$
$$D_{DD}(N) = O(\log N)$$

If we use the pseudo quad-tree for range queries the $O(\log N)$ penalty need not even be paid.

b) In the same way, k-d trees (see Bentley [2]) can be transformed to pseudo k-d trees (see Overmars and van Leeuwen [6]). Willard [9] already developed a special dynamization method for pseudo k-d trees (which he called k-d* trees). Our method is somewhat simpler and achieves the same bounds, namely

$$Q_{DD}(N) \leq Q_S(4N).O(\log N)$$
$$S_{DD}(N) = O(N)$$
$$I_{DD}(N) = O(\log^2 N)$$
$$D_{DD}(N) = O(\log N)$$

c) Overmars and van Leeuwen [5] describe a structure for determining whether a point lies in the common intersection of a dynamically changing set of halfspaces. The structure is able to process both insertions and deletions. The time and space bounds proved are

$$Q(N) = O(\log N)$$
$$S(N) = O(N)$$
$$I(N) = O(\log^2 N)$$
$$D(N) = O(\log^3 N)$$

After a certain ordering of the halfspaces is established, the structure can be built in only $O(N)$. So it has both the properties of a DD- and a MD-searching problem and we can transform it into a structure with (using for T a balanced search tree again)

$$Q(N) = O(\log^2 N)$$
$$S(N) = O(N)$$
$$I(N) = O(\log N)$$
$$D(N) = O(\log^3 N)$$

Hence, at the cost of only a moderate increase in query time, we achieve a decrease in insertion time.

d) Overmars and van Leeuwen [5] also present a structure for determining whether a point is maximal with respect to a dynamically changing set. For this structure the time bounds are

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

This structure also has the properties of both a DD- and MD-searching problem and (hence) it can be transformed into a structure with

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

e) For many other searching problems on totally ordered sets static structures exist with the properties of both DD- and MD-searching problems. Some of them are Member searching, Successor, Predecessor, Rank, Count, Min, Max etc. (ref. Saxe and Bentley [7]).

5. References.

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

[2] Bentley, J.L., Multidimensional Binary Search Trees Used for Associative Searching, Comm. of the ACM 18 (1975) pp. 509-517.

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

[4] Maurer, H.A. and Th. Ottman, Dynamic solutions of decomposable searching problems. Bericht 33, Institut f. Informationsverarbeitung, TU Graz, 1979.

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

[6] Overmars, M.H. and J. van Leeuwen, Dynamic multidimensional datastructures based on quad- and k-d trees, in preparation.

[7] Saxe, J.B. and J.L. Bentley, Transforming static data structures to dynamic structures, Techn. Rep., Dept. of Computer Science, Carnegie-Mellon University, 1979.

[8] van Leeuwen, J. and D. Wood, Dynamization of decomposable searching problems, Techn. Rep. RUU-CS-79-5, Dept. of Computer Science, University of Utrecht, 1979.

[9] Willard, D.E., Balanced forests of k-d* trees as a dynamic data structure, TR-23-78, Aiken Computation Lab., Harvard University, 1978.