SEARCHING IN THE PAST I

Mark H. Overmars

RUU-CS-81-7

April 1981

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

SEARCHING IN THE PAST I

Mark H. Overmars

Technical Report RUU-CS-81-7

April 1981

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

Proposed running head:

SEARCHING IN THE PAST I

All correspondence to:

Mark H. Overmars
Dept. of Computer Science
University of Utrecht
P.O. Box 80.002
3508 TA Utrecht
the Netherlands

SEARCHING IN THE PAST I*

Mark H. Overmars

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

Abstract. Most known data structures for searching problems are unable to remember the situation they held at moments in the past. We will show that for a number of common searching problems, "in-the-past" versions can be solved efficiently as well. We will give structures for member searching and k^{th} element/rank searching in the past, yielding a query time of $O(\log N)$ and an update time of $O(\log N)$ (N the number of updates performed on an initially empty structure). Dobkin and Munro [2] use k^{th} element/rank searching in the past to solve some polyhedra problems. As our structure yields a better query time than theirs we obtain an improvement of the bounds for these polyhedra problems. Moreover we consider the in-the-past version of the d-dimensional range searching and counting problem. For the range counting problem we obtain a structure with a query and update time of $O(\log^d N)$. For the range searching problem we consider the case in which only insertions occur and prove that a query time of $O(\log^d N + k)$ and an insertion time of $O(\log^d N)$ can be obtained (k the number of answers).

Keywords and phrases. In-the-past searching, member searching, k^{th} element searching, rank searching, range searching.

1. Introduction.

Dynamic structures for searching problems are primary devised to keep a set of objects up-to-date while insertions and/or deletions occur, such that queries can be answered efficiently. Hence, when we want to delete an object, we usually eliminate all information about this object from the structure. On the other hand, in a number of applications one might be interested in the situation the structure held at some moment in the past. For instance, if we have a salary administration in a database, it might be important to be able to ask questions like: How many people had a salary $\geq x$ at some given date T. Hence we require that the data structure can

---

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

remember relevant information concerning its own history. Most known dynamic data structures are unable to do so and hence, cannot give information about the situation they held in the past.

Definition. Let S be a dynamic data structure for a searching problem Q

T_0 = the moment we initiated the (empty) structure.
T_i = the moment before the i^{th} update on S.

Although in practice no precise moment T_i exists (since the updating of S takes time), we may assume such a moment is given (preferably the moment we start doing the update). N will always denote the number of the next coming update (that will be performed after T_N). In fact, we will view T_N as being "now". Hence an ordinary dynamic data structure allows only for queries over the situation at T_N. We say that a structure solves a searching problem Q in the past if we can perform updates at moment T_N and we can perform queries over the situation at any moment T with $T_0 < T \leq T_N$. Let $T_i < T \leq T_{i+1}$; performing a query at moment T means performing the query over the pointset after the i^{th} update.

A first structure for answering queries in the past was given by Dobkin and Munro [2]. They gave a structure for performing in-the-past queries for the k^{th} element searching problem that asks for the k^{th} element in an ordered set of points, and for the rank searching problem that asks for the rank of a given element. Their structure is static in the sense that all updates are given beforehand and hence, once their structure is built no updates have to be performed anymore. The structure can be built in O(N log N), takes O(N log N) space and has a query time of $O(\log^2 N)$. Dobkin and Munro [2] use their structure to solve some polyhedra problems.

We will concentrate on the online version of in-the-past searching problems, in which the updates are not known beforehand. Hence our structures have to be dynamic. In section 2 we show that member searching in the past can be solved within O(log N) query and update time bounds, while the storage required remains O(N). In section 3 we consider the online version of the k^{th} element/rank searching problem. We will devise a structure to solve the in-the-past version of this problem within a query time of O(log N) and an update time of O(log N), while the storage requirement becomes O(N log N). In this way we even improve the bounds of the static structure of Dobkin and Munro [2]. It will have an immediate effect on the applications Dobkin and Munro give of their structure. In section 4 we will solve the in-the-past variant of another important searching problem, the range searching problem. Also for this problem we show that efficient

structures can be devised. In section 5 we will give some concluding remarks and directions for further research.

2. Member searching in the past.

The member searching problem is the following: Given a set of objects V and another object x, we want to know whether $x \in V$. The in-the-past version of the problem is the following: Given a dynamically changing set of objects V, another object x and a moment in the past T, we want to know whether $x \in V$ at moment T or not. For the ordinary member searching problem many data structures have been devised, storing the elements of V in such a way that queries can be answered within O(log n) time, where $|V| = n$ (see e.g. Aho, Hopcroft and Ullman [1] or Knuth [4]). All of these structures use O(n) storage and most of them are dynamic, yielding an update time of O(log n) steps. But all these structures are unable to remember the situation they held at moments in the past, because they all throw deleted elements away. When we want to answer member queries in the past we have to keep track of all points that have once been in the set. We will allow that objects that are deleted can be reinserted at some later time, but we assume that when we want to insert a point, it is not present and when we want to delete a point that it is present. (It can always be checked whether or not these conditions are satisfied by first performing a query with the point at time T_N.) Let us look at some point p that has been (or possibly is) in the set. At T_0 (the moment we initiated the structure) p surely was not present, but at some moment T_{i_1} we inserted p. It is possible that at some later moment T_{i_2} ($i_2 > i_1$) p was deleted. Maybe it was reinserted again at T_{i_3}, deleted again at T_{i_4} etc. Hence we get a number of (nonoverlapping) intervals of time $[T_{i_1} : T_{i_2}]$, $[T_{i_3} : T_{i_4}]$ ... with the last one possibly open (hence upto T_N) at which p was present in the set. See figure 1; a 1 denotes that p is present, a 0 that it is not present.

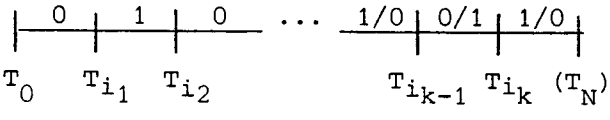

Figure 1.

These observations lead to a fairly simple data structure for member searching in the past. As a main structure we use a balanced search tree S in which we store all points that have once been in the set. For simplicity, we assume that the points are stored in the leaves of S only, but

this is in no way necessary. With each point p we associate a structure S_p that represents the intervals of time at which p was present. Because these intervals are nonoverlapping we can use for S_p a balanced search tree in which we store (T_0), T_{i_1}, T_{i_2}, ... With each T_{i_j} we mark whether p was present after T_{i_j} or not. Hence we get the structure displayed in figure 2. To perform a query Q(x,T) that asks whether x was present in the set at moment T we first search for x in S. If we do not find x, we know that x has never been present in the set and, hence, surely not at time T. Otherwise, we find a structure S_x that holds all intervals at which x was

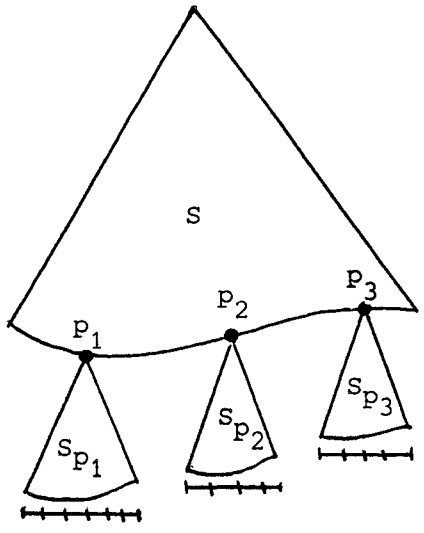

Figure 2.

present. We search in S_x for the biggest $T_i < T$. The information stored at T_i tells us if x was present at moment T or not.

To insert a point p in the structure, we first search for p in S. If we do not find p we know that p has never before been in V and we must insert it in S. Next we build a S_p only consisting of T_N (i.e., the moment of insertion), noting that p is present after T_N. Otherwise, if we find p in S, we insert T_N in S_p (at the right most side), noting that p is present after T_N. If we want to delete a point p we first search for p in S (it must be present). Next we insert T_N in S_p, noting that p is not present after T_N.

Theorem 2.1. There exists a structure S for member searching in the past yielding a query time of O(log N) and an update time of O(log N). The amount of storage required is bounded by O(N).

Proof

The main structure S contains at most N points and all associated structures S_p together contain at most N moments of time. Hence, a query consists of two simple searches on balanced structures of at most N points, and thus takes at most O(log N). Because an insertion consists of a search on S and an insertion in S or in a S_p and a deletion consists also of a search on S and an insertion in a S_p, the update time is bounded by O(log N). Also the bound on the storage required follows trivially. □

3. K^{th} element/rank searching in the past.

The k^{th} element searching problem asks for the k^{th} element (k given as a query object) in an ordered set of points, and the rank searching problem asks for the rank of a given object in an ordered set. For both problems structures are known with a query time of O(log n) and an update time of O(log n) when the set contains n points, but these structures do not keep track of their history. A first structure for k^{th} element/rank searching in the past was devised by Dobkin and Munro [2]. Their structure is static in the sense that all updates must be given beforehand. In that case the structure can be built in O(N log N) steps, and from that moment on we can perform in-the-past queries on it within $O(\log^2 N)$ steps each. We will develop a dynamic structure for the problem with even better time bounds than the static structure of Dobkin and Munro [2].

The structure we use consists of an augmented BB[α] tree S (see e.g. Willard [11] and Lueker [5]) in which we keep all points that have once been in the set in the appropriate order at the leaves. At each internal node β we would like to have some information about the number of the points in the subtree, rooted at β, that where present at different moments of time. Therefore, we associate with each internal node β a list L_β that contains all moments of time T_{i_j} at which a point in the subtree, rooted at β, got inserted or deleted. With each such moment T_{i_j} we give the number of points below β present between T_{i_j} and $T_{i_{j+1}}$. With β we have two pointers first(β) and last(β) pointing to the first, resp. the last moment in L_β (see figure 3). To be able to search fast through the associate lists, we link them internally in a way similar to the structure for range searching described by Willard [10].

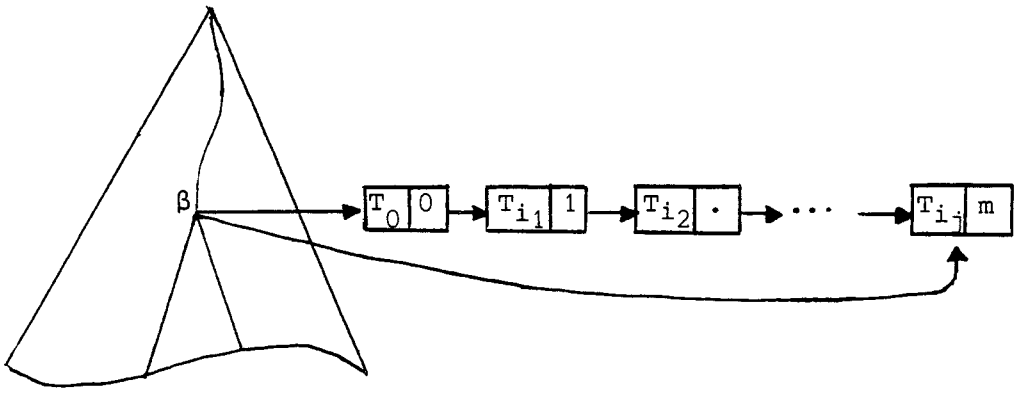

Figure 3.

To this end we add to each record in L_β with time moment T_{i_j} a pointer lson to the last moment before or equal to T_{i_j} in $L_{lson(\beta)}$ (the list associated with the leftson of β) and a pointer rson to the last moment before or equal to T_{i_j} in $L_{rson(\beta)}$. Hence, each record in a list L_β contains the following fields (see figure 4)

time: the moment of time,

numb: the number of point in the subtree below β between moment "time" and the next moment in the list,

lson: the pointer to $L_{lson(\beta)}$,

rson: the pointer to $L_{rson(\beta)}$,

next: a pointer to the next record in the list.

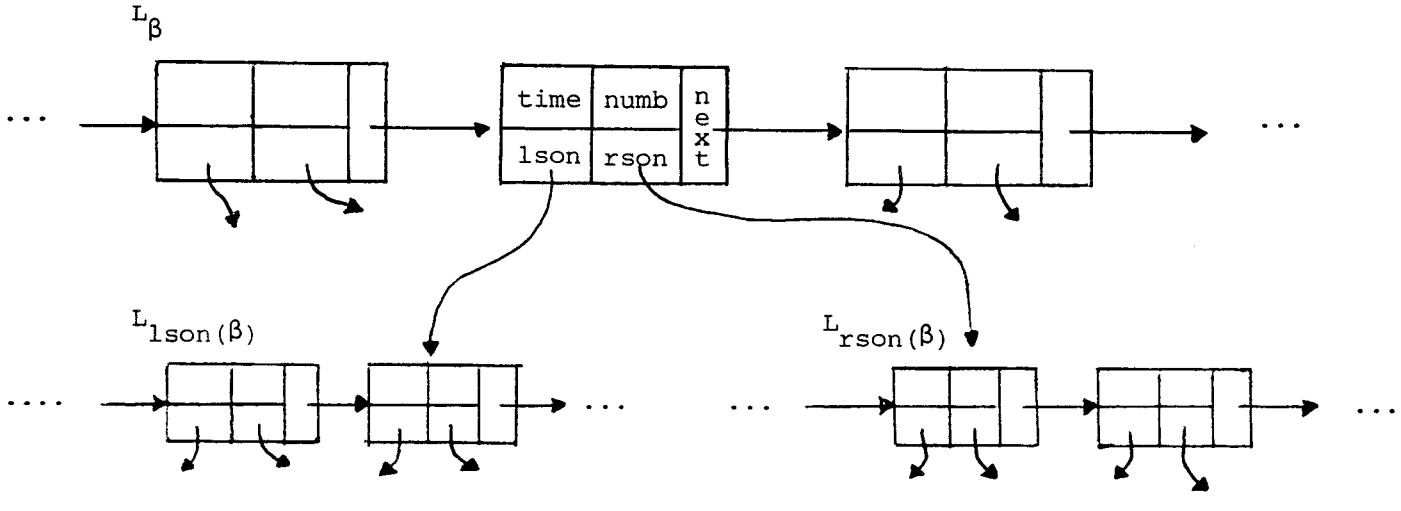

Figure 4.

Moreover, to be able to search in the list L_{root} associated with the root of S, that contains all moments of time T_i, we build a balanced search tree S_T on top of L_{root}.

We will first describe how to perform queries on the structure described. When we want to perform a k^{th} element query at time T we first search in S_T for the last $T_i < T$. In this way we come into a record rec in L_{root}. The actions we have to perform from this moment on are best described by the following recursive procedure that steadily works its way down the tree with one pointer in the tree S to find the appropriate k^{th} element and another pointer in the associated lists to guide the search.

```
procedure FINDK(node β, record rec, integer k);
# the point we are searching for is the k^th element at time T below β.
  rec is the record in L_β that contains the last moment before T at
  which some point below β got inserted or deleted (or T_0) #

begin
   if β is a leaf
      then if k = 1 and numb of rec = 1
              then # β contains the point we searched for. #
                   report the point contained in β
              else # there were too few points in S at T.#
                   report that the point does not exist
           fi
      else integer l := numb of lson of rec;
           # l is the number of points present below lson(β) at T. #
           if l ≥ k
              then # the point we search for lies below lson(β). #
                   FINDK(lson(β), lson of rec, k)
              else # the point we search for is the k-l^th element
                     below rson(β) at moment T. #
                   FINDK(rson(β), rson of rec, k-l)
           fi
end of FINDK;
```

We call the procedure as FINDK(root, rec, k);

Lemma 3.1. A k^{th} element query in the past takes at most O(log N) steps.

Proof

Searching for the appropriate $T_i < T$ in S_T takes at most O(log N) as S_T is balanced and contains N moments of time. Beside the recursive call, FINDK takes only O(1) time. With each recursive call we go one level deeper in S and, as S is balanced and contains no more than N points, also FINDK takes no more than O(log N) steps to finish. □

To perform a rank query with a point x at moment T we again search with one pointer through S and with the other through the associated lists but this time the pointer in S guides the search and the pointer in the lists will compute the answer. Again we start searching with T in S_T to they find the last $T_i < T$. In this way we find a record rec in L_{root}. The remaining steps are best described by the following recursive procedure FINDR.

```
procedure FINDR(point x, node β, record rec, integer k);
# x is the query object of which we want to know the rank, β is the node
  we most recently reached on our search towards x, rec is the record in Lβ
  containing the last time moment <T at which a point below β got inserted
  or deleted and k gives the number of points we already found to be smaller
  than x at moment T. #

begin
   if β is a leaf
      then if β contains x and numb of rec = 1
              then report k + 1   # x is present at T #
              else report k       # x is not present at T #
           fi
      else if x below lson(β)
              then FINDR(x, lson(β), lson of rec, k)
              else # all elements below lson(β) are smaller than x #
                   FINDR(x, rson(β), rson of rec, k + numb of lson of rec)
           fi
   fi
end of FINDR;
```

We call the procedure as FINDR(x, root, rec, 0);

Lemma 3.2. A rank query in the past takes at most O(log N) steps.

Proof

The arguments are exactly the same as in the proof of lemma 3.1. □

Hence, the structure enables us to perform both k^{th} element and rank queries in the past efficiently. It remains to be shown that the structure can be updated efficiently as well.

We again assume that, when we insert a point p, p is not present (but it might have been present before) and that, when we want to delete a point, it is present. We will first consider insertions. Note that an insertion always is performed at time T_N. So in all lists associated to nodes on the path towards the newly inserted point p we have to add a record containing T_N. As T_N is bigger than every previous moment it has to be added at the right side of the lists, which can be located easily following the pointer last. There is only one problem. Because we update the tree it may have gone out of balance. The task of rebalancing we delegate to a procedure BALANCE that will be described later. The procedure INSERT described below works its way down the tree towards the place the new point needs to be inserted, meanwhile updating the lists. After it reaches the leaf it calls for BALANCE that works back to the root, meanwhile making the appropriate rebalancings.

```
procedure INSERT(point p, node β);
# p is the point we want to insert, β is the node we most recently
  reached on our way down the tree towards the place where p must be in-
  serted. #

begin
   # add a new record to L_β #
   record rec;
   time of rec := T_N;
   numb of rec := numb of last(β) + 1;
   next of last(β) := rec;
   last(β) := rec;
   # fill in the lson and rson fields in the records we added to father(β). #
   if β ≠ root
     then lson of last(father(β)) := last(lson(father(β)));
          rson of last(father(β)) := last(rson(father(β)))
   fi;
```

```
if β is a leaf
   then if β does not contain p
           then # p has never been present before #
                build a node for p with a list consisting only
                of T_0 and T_N;
                insert this node in the tree;
                BALANCE
        fi
   else if p must come below lson(β)
           then INSERT(p,lson(β))
           else INSERT(p,rson(β))
        fi
fi
end of INSERT;
```

We call the procedure as INSERT(p,root);. Afterwards we insert T_N in S_T and give it a pointer to the last record in L_{root}. In this way the structure is maintained in the appropriate way, provided that we have an algorithm for BALANCE.

As our main structure, S, is a BB[α]-tree, rebalancing consists of performing some single or double rotations along the search path towards the inserted point. We will only consider the case in which single rotations are needed as double rotations can be treated similar. See figure 5 for the notation we follow.

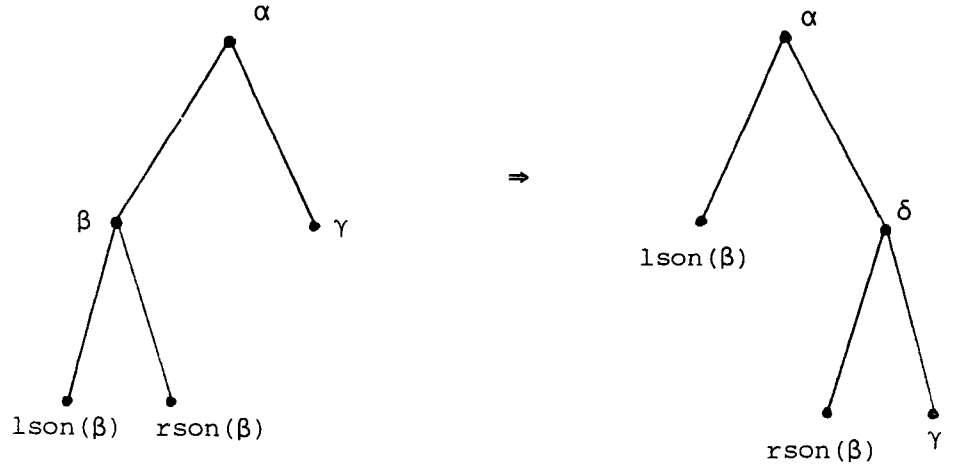

Figure 5.

To perform such a single rotation we have to build the list L_δ and to recompute the pointers lson and rson of the records in L_α. Building L_δ can be done by merging (and copying) the lists $L_{rson(\beta)}$ and L_γ. Recomputing the values in L_α can be done during one walk along $L_{lson(\beta)}$, L_δ and the old L_α. It follows that, when the size of L_α is N', the total amount of

work needed for such a single rotation is bounded by O(N'). A similar result holds for performing a double rotation. One can show (see e.g. Lueker [5] and Willard [11]) that we can divide this work over some c.N' insertions that will be performed in the subtree rooted at α after the rotation is needed, each time doing O(1) work on the construction of the lists, while, in the meantime, no rotation is needed for any of the nodes concerned. For query answering we still use the information in the old lists or in the list of some near descendents of α. It has been shown that in this way the structure remains balanced and that the query time does not increase in order of magnitude (see Lueker [5] and Willard [11] for more details). It follows that with each insertion we need to do at most O(1) work on reconstruction for each node on the search path towards the new point and hence at most O(log N) work total.

Lemma 3.3. Insertions take at most O(log N) time each.

Proof

The procedure INSERT takes, besides the recursive call and BALANCE, O(1) steps, and hence, because S is balanced and BALANCE takes O(log N) time, a total of O(log N) steps. Also the updating of S_T takes at most O(log N). □

Deleting a point p is much easier than inserting a point. We only have to add a record with T_N to the list of each node β on the search path towards p but in this case with numb of rec := numb of last(β) - 1, and to insert T_N in S_T. Because we do not actually delete p from S, the structure remains balanced.

Lemma 3.4. Deletions take at most O(log N) each.

Proof

The bound follows from the proof of lemma 3.3., ignoring the cost for BALANCE. □

We can now state the main result of this section.

Theorem 3.5. There exists a structure for k^{th} element/rank searching in the past such that queries, insertions and deletions take at most O(log N) steps each. The storage required is bounded by O(N log N).

Proof

The bounds on query, insertion and deletion time follow from lemma 3.1. - 3.4. One easily verifies that with each update we add O(log N) records to lists. It follows that after N updates (i.e., "now") there are at most O(N log N) records. The size of S and S_T is clearly bounded by O(N). □

It follows that our dynamic structure for k^{th} element/rank searching in the past is better than the static structure described by Dobkin and Munro [2], which had a query time of $O(\log^2 N)$. It follows that the bounds on the applications given by Dobkin and Munro [2] can be improved as well. For example, we get the following results:

Theorem 3.6. (= theorem 2 in [2]) If P (resp. Q) is a polyhedron of p (resp. q) vertices and R is a polygon of r vertices, we may preprocess P and Q individually in O(p log p) (resp. q log q) operations after which

i) we can detect intersections of P and Q in $O(\log^4 (p+q))$ operations.
ii) we can detect intersections of P and R in $O(\log^3 (p+r))$ operations.

Theorem 3.7. (= theorem 3 in [2]) We may preprocess a planar subdivision P of p vertices in $O(p \log^2 p)$ operations so that we may determine the i^{th} edge of P which intersects any line in $O(\log^2 p)$ operations.

4. Range searching in the past.

The d-dimensional range searching problem is the following: Given a set V of points $x = (x_1,\ldots,x_d)$ in d-dimensional space and a range (= a rectilinearly oriented hyper-rectangle) $R = ([a_1:b_1],\ldots,[a_d:b_d])$, list all points x in V that lie within R, i.e., with $a_1 \le x_1 \le b_1$, $a_2 \le x_2 \le b_2$, ... and $a_d \le x_d \le b_d$. If we are only interested in the number of points of V that lie within R then we call the problem the range counting problem. A number of data structures have been devised for solving the d-dimensional range searching (resp. range counting) problem. The most efficient dynamic structure known was independently devised by Lueker [5] and Willard [11]. It yields a query time of $O(\log^d n + k)$ (resp. $O(\log^d n)$ for the counting problem) and an insertion and deletion time of $O(\log^d n)$, where n is the number of points in the set and k is the number of reported objects. The structure can be built in $O(n \log^{d-1} n)$ and takes $O(n \log^{d-1} n)$ storage. Recently, the deletion time was improved to $O(\log^{d-1} n)$ when $d \ge 2$ (see Overmars and van Leeuwen [9]).

We will first show how to solve the in-the-past version of the range counting problem. Using the structure for rank searching described in

section 3 we can very easily solve the 1-dimensional range counting problem, in which we have a set of points on a line. Given a segment [a:b] on the line, the number of points that lie within [a:b] is equal to the number of points ≤b minus the number of point <a, i.e., if a is not in the set it is rank b - rank a and otherwise it is rank b - rank a + 1. Hence, we can solve the 1-dimensional range counting problem using the rank searching problem. To solve the in-the-past version of the 1-dimensional range counting problem we can use the structure described in section 3, yielding a query and an update time of O(log N).

To solve the d-dimensional range counting problem in the past we use a generalization of the structure for the 1-dimensional case, similar to the one described by Lueker [5] and Willard [11]. As a main structure we use a BB[α]-tree in which we keep all points that have once been in the set, ordered with respect to their first coordinate. With each internal node β we associate a structure for d - 1 dimensional range counting in the past on the remaining coordinates, of all points in the subtree rooted at β. To perform a query with a range $([a_1:b_1],\ldots,[a_d:b_d])$ at some moment T in the past, we search with both a_1 and b_1 on the structure. For some time a_1 and b_1 will follow the same path, but at some moment a_1 will go to the left while b_1 goes to the right. The points that lie between a_1 and b_1 are precisely those that lie under a node β whose father is on the search path of a_1 or b_1 but that is not on the search path itself, and lies in between the two paths (see figure 6). There are at most O(log N) nodes of this kind. On the points below these nodes we want

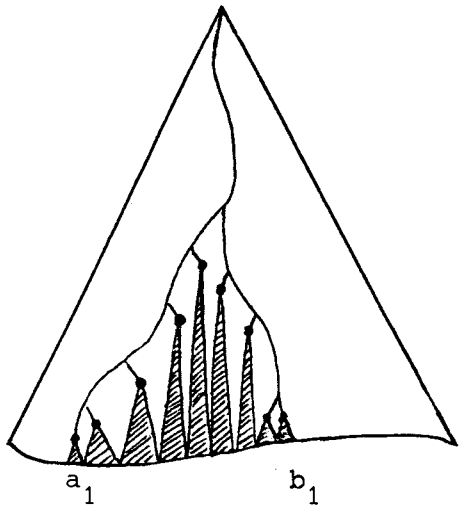

Figure 6.

to perform a range query at T with the remaining coordinates. To do so we perform a range query with $([a_2:b_2],\ldots,[a_d:b_d])$ at T on the structures associated with the indicated nodes. Afterwards we add up the answers. Note that the moment of time T is only referred to by the time we have come to the associated 1-dimensional structures. Besides the different 1-dimensional substructures, the structure is completely similar to the one described by Lueker [5] and Willard [11]. Also insertions and deletions can be carried out in exactly the same way, except that when we perform a deletion we not actually delete the point. We only update the lists of the 1-dimensional substructures.

Theorem 4.1. There exists a structure for d-dimensional range counting in the past yielding a query time of $O(\log^d N)$ and an insertion and deletion time of $O(\log^d N)$. The storage required is bounded by $O(N \log^d N)$.

Proof

A d-dimensional query consists of O(log N) d - 1 dimensional queries plus O(log N) additional work. As a 1-dimensional range query in the past takes O(log N) the bound on the query time follows. The bound on the update time follows from Lueker [5] and Willard [11]. The d-dimensional structure uses, besides the associated structures, O(N) space. As each point is in at most O(log N) associated d - 1 dimensional structures and the 1-dimensional structure takes O(N log N), the bound on the required amount of storage follows. □

It should be noted that the improvement in deletion time obtained for the ordinary range counting problem (see [9]) does not carry over to the structure for the in-the-past version of the problem.

We will now consider the range searching problem in the past. We restrict ourselves to the case in which only insertions occur. As a 1-dimensional structure we use a structure very similar (but simpler) to the structure described in section 3. We again store all point that have once been in the set (= the set at T_N, as we do not delete points) in a BB[α]-tree S. To each internal node β we associate a list L_β in which we store the points below β in order of insertion, together with the moments of insertion. A pointer first(β) points to the first record in the list and a pointer last(β) to the last record (see figure 7). To perform a range query with range [a:b] at time T, we first search with a and b in S. Next we consider

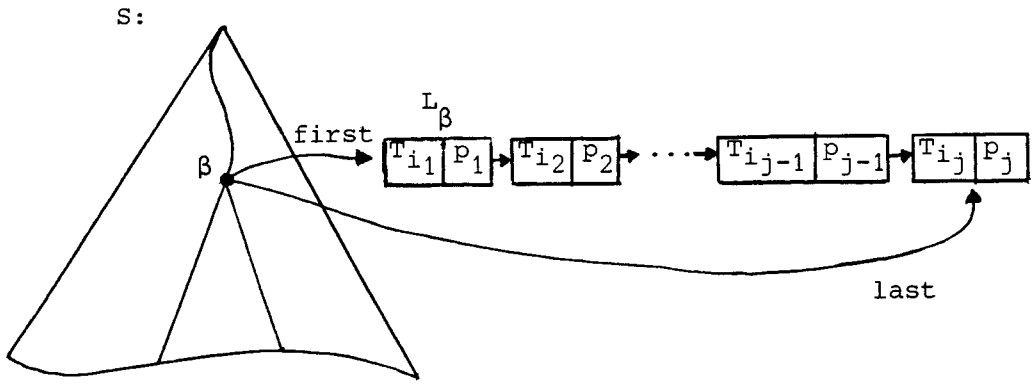

Figure 7.

all internal nodes β bordering the search-paths of a and b and lying in between the two paths. The lists associated to these nodes contain each point that lies between a and b exactly once. As we only want to list the points that were present at time T, i.e., that where inserted before time T, we walk along each list, reporting the points we find, until we reach a record with time ≥T. One easily verifies that in this way each point that lies between a and b and that was present at time T, is reported exactly once. The total amount of work is bounded by O(log N+k), where k is the number of reported answers. To insert a point p we search for p in S and for each node β on the search-path we add a record, containing T_N and p to the list. Rebalancing (when needed) is done in the same way as in section 3. It follows that an insertion takes at most O(log N) work. The amount of storage is clearly bounded by O(N log N). Generalizing the result to the d-dimensional case is done in exactly the same way as for the counting problem. One easily verifies that this leads to the following result:

Theorem 4.2. There exists a structure for d-dimensional range searching in the past such that queries take $O(\log^d N+k)$ (k the number of reported answers) and insertions can be performed in $O(\log^d N)$. The amount of storage needed is bounded by $O(N \log^d N)$.

When both insertions and deletions occur, we can no longer use the structure described above. In this case, we can use a general technique, based on the decomposability of the problem, to transform known dynamic data structures for the ordinary problem into structures for solving the in-the-past variant (see Overmars [7]). It yields a structure with a

query time of $O(\log^{d+1} N+k)$, an insertion time of $O(\log^{d} N)$ and a deletion time of $O(\log^{d+1} N)$, while the storage required is bounded by $O(N \log^{d} N)$.

5. Concluding remarks.

We have shown that for some searching problems their "in-the-past" versions can be solved very efficiently, making use of the special properties of the dimension time. It remains a topic of further research whether similar results can be obtained for other searching problems as well. The method described for range searching in section 4 carries over quite easily to structures for rectangle intersection searching as described by Edelsbrunner and Maurer [3]. But for structures like the ones for convex hull searching (Overmars and van Leeuwen [8]) or nearest neighbor searching (Overmars [6]) it might be very hard to transform them into structures that remember their history. On the other hand, it is possible to give a very general method for transforming dynamic data structures for a searching problem Q into structures for solving Q in the past, but the resulting structures are not very efficient (see Overmars [7]). Also, for decomposable searching problems one can give a general method for transforming static structures into structures that remember history, but these structures only support insertions. When fully dynamic data structures are known for a decomposable searching problem, they can be transformed into fully dynamic structures that remember history, without essential loss in efficiency (see [7]).

6. References.

[1] Aho, A.V., J. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley, Reading, Mass., 1974.

[2] Dobkin, D.P. and J.I. Munro, Efficient uses of the past, Proc. 21th IEEE Symp. on Foundations of Computer Science, 1980, pp. 200-206.

[3] Edelsbrunner, H. and H.A. Maurer, On the intersection of orthogonal objects, Bericht 60, Inst. f. Informationsverarb., TU Graz, 1980.

[4] Knuth, D.E., The art of computer programming, vol. 3: sorting and searching, Addison-Wesley, Reading, Mass., 1973.

[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, 1978.

[6] Overmars, M.H., Dynamization of order decomposable set problems, Techn. Rep. RUU-CS-80-9, Dept. of Computer Science, University of Utrecht, 1980. (To appear in the Journal of Algorithms.)

[7] Overmars, M.H., Searching in the past II: general transforms, Techn. Rep. RUU-CS-81-9, Dept. of Computer Science, University of Utrecht, 1981.

[8] 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. Revised version: RUU-CS-81-3. (To appear in J. Comput. Syst. Sci.)

[9] Overmars, M.H. and J. van Leeuwen, Worst-case optimal insertion and deletion methods for decomposable searching problems, Techn. Rep. RUU-CS-80-10, Dept. of Computer Science, University of Utrecht, 1980. (To appear in Inform. Proc. Lett.)

[10] Willard, D.E., New data structures for orthogonal queries, Techn. Rep. TR-22-78, Aiken Computation Lab., Harvard University, 1978.

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