SIAM J. COMPUT.
Vol. 17, No. 3, June 1988

A FUNCTIONAL APPROACH TO DATA STRUCTURES AND ITS USE IN MULTIDIMENSIONAL SEARCHING*

BERNARD CHAZELLE†

Abstract. We establish new upper bounds on the complexity of multidimensional searching. Our results include, in particular, linear-size data structures for range and rectangle counting in two dimensions with logarithmic query time. More generally, we give improved data structures for *rectangle problems* in any dimension, in a static as well as a dynamic setting. Several of the algorithms we give are simple to implement and might be the solutions of choice in practice. Central to this paper is the nonstandard approach followed to achieve these results. At its root we find a redefinition of data structures in terms of functional specifications.

Key words. functional programming, data structures, concrete complexity, multidimensional search, computational geometry, pointer machine, range search, intersection search, rectangle problems

CR Categories. 5.25, 3.74, 5.39

1. Introduction. This paper has two main parts: in § 2, we discuss a method for transforming data structures using functional specifications; in the remaining sections, we use such transformations to solve a number of problems in multidimensional searching. To begin with, let us summarize the complexity results of this paper.

The generalization of the notion of rectangle in higher dimensions is called a *d-range*: it is defined as the Cartesian product of d closed intervals over the reals. Let V be a set of n points $\Re^d$ and let v be a function mapping a point p to an element $v(p)$ in a commutative semigroup $(G, +)$. Let W be a set of n d-ranges.

(1) **Range counting:** given a d-range q, compute the size of $V \cap q$.
(2) **Range reporting:** given a d-range q, report each point of $V \cap q$.
(3) **Semigroup range searching:** given a d-range q, compute $\sum_{p \in V \cap q} v(p)$.
(4) **Range searching for maximum:** semigroup range searching with maximum as semigroup operation.
(5) **Rectangle counting:** given a d-range q, compute the size of $\{r \in W \mid q \cap r \neq \emptyset\}$.
(6) **Rectangle reporting:** given a d-range q, report each element of $\{r \in W \mid q \cap r \neq \emptyset\}$.

In each case, q represents a query to which we expect a fast response. The idea is to do some preprocessing to accommodate incoming queries in a repetitive fashion.

Note that range counting (resp., reporting) is a subcase of rectangle counting (resp. reporting). To clarify the exposition (and keep up with tradition), however, we prefer to treat these problems separately. Other well-known problems falling under the umbrella of rectangle searching include *point enclosure* and *orthogonal segment intersection.* The former involves computing the number of d-ranges enclosing a given query point, while the latter, set in two dimensions, calls for computing how many horizontal segments from a given collection intersect a query vertical segment. In both cases, the reduction to rectangle counting is immediate.

The thrust of our results is to demonstrate the existence of efficient solutions to these problems that use minimum storage. One of the key ideas is to redesign range

* Received by the editors September 23, 1985; accepted for publication (in revised form) March 9, 1987. A preliminary version of this paper has appeared in the Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science, Portland, Oregon, October 1985, pp. 165-174.

† Department of Computer Science, Princeton University, Princeton, New Jersey 08544. This work was begun when the author was at Brown University, Providence, Rhode Island 02912. This work was supported in part by National Science Foundation grant MCS 83-03925.

trees (Bentley [B2]) and segment trees (Bentley [B1], Bentley and Wood [BW]) so as to require only linear storage. Since improvements in this area typically involve trimming off logarithmic factors, one must be clear about the models of computation to be used. It is all too easy to "improve" algorithms by encoding astronomical numbers in one computer word or allowing arbitrarily complex arithmetic. To guard us from such dubious tricks, we will assume that the registers and memory cells of our machines can only hold integers in the range $[0, n]$. In the following, the sign $\times$ refers to multiplication, $\div$ to division (truncated to the floor), and *shift* to the operation *shift* $(k) = 2^k$, defined for any k $(0 \leqq k \leqq \lfloor \log n \rfloor)$.[1] No operation is allowed if the result or any of the operands falls outside of the range $[0, n]$. This will make our model very weak and thus give all the more significance to our upper bounds.

Remark. As observed in Gabow et al. [GBT] the "orthogonal" nature of the problems listed above makes it possible to work in rank space, that is, to deal not with the coordinates themselves but with their ranks. This is precisely what we will be doing here. The conversion costs logarithmic time per coordinate, but it has the advantage of replacing real numbers by integers in the range $[1, n]$. It is important to keep in mind that although our data structures will use integers over $O(\log n)$ bits internally, no such restriction will be placed on the input and query coordinates. On the contrary these will be allowed to assume any real values.

The models of computation we will consider include variants of *pointer machines* (Tarjan [T]). Recall that the main characteristic of these machines is to forbid any kind of address calculation. New memory cells can be obtained from a free list and are delivered along with pointers to them. A pointer is just a symbolic name, that is, an address whose particular representation is transparent to the machine and on which no arithmetic operation is defined. Only pointers provided by the free list can be used. For the time being, let us assume that the only operations allowed are $=$ and $<$ along with the standard Booleans. We introduce our models of computation in order of increasing power. Of course, we always assume that the semigroup operations which might be defined (problems 3–4) can be performed in constant time.

(1) An *elementary pointer machine* (EPM) is a pointer machine endowed with $+$.
(2) A *semi-arithmetic pointer machine* (SAPM) is a pointer machine endowed with $+, -, \times, \div$.
(3) An *arithmetic pointer machine* (APM) is a pointer machine endowed with $+, -, \times, \div$, *shift*.
(4) A *random access machine* (RAM) is endowed with comparisons, and $+, -, \times, \div$. See Aho et al. [AHU] for details.

Our motivation for distinguishing between these models is twofold: one reason is to show that our basic techniques still work even on the barest machines. Another is to assess the sensitivity of the complexity of query-answering to the model of computation. Next, we briefly discuss these definitions. First of all, complexity is measured in all cases under the uniform cost criterion (Aho et al. [AHU]). Note that subtraction can be simulated in $O(\log n)$ time and $O(n)$ space on an EPM by binary search. Similarly, *shift* can be simulated on an SAPM in $O(\log \log n)$ time by binary search in a tree of size $O(\log n)$ (this can also be done with constant extra space by repeated squaring). For this reason, we will drop the SAPM model from consideration altogether. Any result mentioned in this paper with regard to an APM also holds on an SAPM, up to within a multiplicative factor of $\log \log n$ in the time complexity. All

[1] All logarithms are taken to the base 2. Throughout this paper, we will use the notation $\log^c n$ to designate $(\log n)^c$.

the operations mentioned above are in the instruction set of any modern computer, so our models are quite realistic. We have omitted *shift* from the RAM, because this operation can be simulated in constant time by table look-up. One final comment concerns the word-size. Suppose that for some application we need to use integers in the range $[-n^c, n^c]$, for some constant $c>0$. Any of the machines described above will work just as well. Indeed, we can accommodate any polynomial range by considering virtual words made of a constant number of actual machine words. The simulation will degrade the time performance by only a constant factor. See Knuth [K2] for details on multiple-precision arithmetic.

The complexity results. We have summarized our results for $\mathfrak{R}^2$ in Tables 1 and 2. The first concerns the static case. Each pair (x, y) indicates the storage $O(x)$ required by the data structure and the time $O(y)$ to answer a query; we use ε to denote an arbitrary positive real. In the second table one will find our results for the dynamic case. We give successively the storage requirement, the query time, and the time for an insertion or a deletion. In both cases, k indicates the number of objects to be reported plus one (we add a 1 to treat the no-output case uniformly). The time to construct each of these data structures is $O(n \log n)$.

TABLE 1
The static case.

Problem	RAM	APM	EPM
range/rectangle counting	$(n, \log n)$	$(n, \log n)$	$(n, \log^2 n)$
range/rectangle reporting	$\left(n, k\left(\log\frac{2n}{k}\right)^{\varepsilon}+\log n\right)$ $\left(n \log\log n, k \log\log\frac{4n}{k}+\log n\right)$ $(n \log^{\varepsilon} n, k+\log n)$	$\left(n, k \log\frac{2n}{k}\right)$	$\left(n, k\left(\log\frac{2n}{k}\right)^2\right)$
range search for max	$(n, \log^{1+\varepsilon} n)$ $(n \log\log n, \log n \log\log n)$ $(n \log^{\varepsilon} n, \log n)$	$(n, \log^2 n)$	$(n, \log^3 n)$
semigroup range search	$(n, \log^{2+\varepsilon} n)$ $(n \log\log n, \log^2 n \log\log n)$ $(n \log^{\varepsilon} n, \log^2 n)$	$(n, \log^3 n)$	$(n, \log^4 n)$

TABLE 2
The dynamic case on an EPM.

Problem	Storage	Query time	Update time
range/rectangle counting	$O(n)$	$O(\log^2 n)$	$O(\log^2 n)$
range reporting	$O(n)$	$O(k(\log 2n/k)^2)$	$O(\log^2 n)$
range search for max	$O(n)$	$O(\log^3 n \log\log n)$	$O(\log^3 n \log\log n)$
semigroup range search	$O(n)$	$O(\log^4 n)$	$O(\log^4 n)$
rectangle reporting	$O(n)$	$O(k(\log 2n/k)^2+\log^3 n)$	$O(\log^2 n)$

In the dynamic case, we have restricted ourselves to the EPM model. Our objective was only to show that the techniques of this paper could be dynamized even in the weakest model. It is likely that many of these bounds can be significantly lowered if we are ready to use a more powerful model such as a RAM or an APM. We leave these improvements as open problems. In the remainder of this paper we will mention upper bounds only in connection with the *weakest* models in which they hold. This is quite harmless as long as the reader keeps in mind that any result relative to an EPM holds on an APM or a RAM. Similarly, anything one can do on an APM can be done just as well on a RAM. Also for the sake of exposition, we restrict ourselves to two dimensions ($d=2$), but we recall a classical technique (Bentley [B2]) that allows us to extend all our data structures to $\Re^d$ ($d>2$). To obtain the complexity of the resulting algorithms, just multiply each expression in the original complexity by a factor of $\log^{d-2} n$ (note: the terms involving k remain unchanged, but a term $\log^{d-1} n$ is to be included in the query times).

Update times are best thought of as amortized bounds, that is, averaged out over a sequence of transactions. A general technique can be used in most cases, however, to turn these bounds into worst-case bounds (Willard and Lueker [WL]). Similarly, a method described in Overmars [O] can often be invoked to reduce deletion times by a factor of $\log n$. Finally, we can use a result of Mehlhorn [Me] and Willard [W1] to show that if we can afford an extra $\log^{\varepsilon} n$ factor in query time then the storage can often be reduced by a factor of $\log\log n$ with each increment in dimension. We will not consider these variants here for at least two reasons. The first is that the techniques have been already thoroughly exposed and it would be tedious but elementary to apply them to our data structures. The second is that these variants are usually too complex to be practical. We will strive in this paper to present data structures that are easy to implement. We have not succeeded in all cases, but in some we believe that we have. For example, our solutions to range counting are short, simple, and very efficient in practice. To illustrate this point we have included in the paper the code of a Pascal implementation of one of the solutions.

Comparison with previous work. Roughly speaking, our results constitute improvements of a logarithmic factor in storage over previous methods. In particular, we present the first linear-size data structures for range and rectangle counting in two dimensions with logarithmic query times. For these two problems our data structures are essentially memory compressed versions of the range tree of Bentley [B2], using new implementations of the idea of a downpointer introduced by Willard [W2]. As regards range and rectangle reporting on a RAM, we improve a method in Chazelle [C1] from $(n(\log n/\log\log n), k+\log n)$ to $(n\log^{\varepsilon} n, k+\log n)$. Interestingly, we have shown in Chazelle [C2] that the $(n(\log n/\log\log n), k+\log n)$ algorithm is optimal on a pointer machine. This constitutes a rare example (outside of hashing), where a pointer machine is provably less powerful than a RAM. Concerning range search for maximum, we improve over a data structure of Gabow et al. [GBT] from $(n\log n, \log n)$ to $(n\log^{\varepsilon} n, \log n)$. As regards semigroup range searching we present an improvement of a factor $\log^{1-\varepsilon} n$ space (again for any $\varepsilon>0$) over the algorithm for the same problem in Willard [W1]. In the group model (the special case of semigroup range searching where an inverse operation exists) we could not find any obvious way of taking advantage of the inverse operation to improve on the results in the table (except, of course, for the case of range counting). As a result, our $(n\log^{\varepsilon} n, \log^2 n)$ algorithm may compare favorably with Willard's $(n\log n, \log n)$ [W2] in storage requirement, but it is superseded in query time efficiency. Our other results for the static case represent tradeoffs and cannot be compared with previous work. In the dynamic case,

our upper bounds improve previous results by a factor of log n space but, except for range and rectangle counting, they also entail extra polylogarithmic costs in query and update times. Also, what makes comparisons even more difficult is that in order to prove the generality of our space-reduction techniques we have purposely chosen a very weak model of computation, i.e., the EPM.

2. Functional data structures. In trying to assess whether a particular data structure is optimal or not, it is natural to ask oneself: why is the data stored where it is and not elsewhere? The answer is usually "to facilitate the computation of some functions implicitly associated with the records of the data structure." For example, one will store keys in the nodes of a binary search tree to be able to branch left or right depending on the outcome of a comparison. An important observation is that nothing demands that a node should store its own key explicitly. All that is required is that whenever the function associated with that node is called, the node had better allow for its prompt evaluation. Having the key stored at the node at all times might be handy but it certainly is more than is strictly needed. There is a classical example of this fact: in their well-known data structure for planar point location [LP1], Lee and Preparata start out with a balanced tree whose nodes are associated with various lists. Then they make the key remark that many elements in these lists can be removed because whenever they are needed by the algorithm they will always have been encountered in other lists before. This simple transformation brings down the amount of storage required from quadratic to linear. However different from the previous one the resulting data structure might be (being much smaller, for one thing), it still has the same *functional* structure as before. In other words, the functions and arguments associated with each node, as well as their interconnections, have gone unchanged through the transformation. Only the assignment of data has been altered. This is no isolated case. Actually, many data structures have been discovered through a similar process. We propose to examine this phenomenon in all generality and see if some useful methodology can be derived from it.

There are several ways of looking at data structures from a design point of view. One might choose to treat them as structured mappings of data into memory. This compiler-level view addresses implementation issues and thus tends to be rigid and overspecifying in the early stage of the design process. Instead, one can take data structures a bit closer to the notation of abstract data type, and think of them as combinatorial structures that can be used and manipulated. For example, a data structure can be modeled as a graph with data stored at the nodes (Earley [Ea]). Semantic rules can be added to specify how the structure can be modified and constraints can be placed to enforce certain "shape" criteria (e.g., balance or degree conditions). If needed, formal definitions of data structures can be provided by means of grammars or operational specifications (Gonnet and Tompa [GoT]). Note that despite the added abstraction of this setting, a data structure is still far removed from an abstract data type (Guttag [G]). In particular, unlike the latter, the former specifies an *architecture* for communicating data and operating on it.

The framework above favors the treatment of data structures as combinatorial objects. It emphasizes *how* they are made and used rather than *why* they are the way they are. This is to be expected, of course, since a data structure may be used for many different purposes, and part of its interpretation is thus best left to the user. Balanced binary trees are a case in point: they can be used as search structures, priority queues, models of parallel architecture, etc. It is thus only natural to delay their interpretation so one can appreciate their versatility. This approach is sound, for it allows us to map

rich combinatorial constructions into useful data structures. It has one negative side-effect, however. Too much concern with the definition and construction aspects of a data structure makes one forget about why it is being used in the first place. What makes the problem all the more troublesome is that algorithms often run two distinct processes: one to build data structures, and another to use them. Whether these processes be distinct in time (e.g., as in a static search tree) or interleaved (e.g., as in dynamic or self-adjusting structures), they can most often be distinguished. Surprisingly, these processes often bear little relation to each other. Building or updating a range tree (Bentley [B2]), for example, and using it to answer a query are only remotely connected tasks. This makes our earlier question all the more difficult to answer: why is a particular structuring of data preferable to another?

This question goes to the heart of our discussion. Data structures are implementations of *paradigms*. The latter, unlike the former, tend to be few and far between, so most data structures appear as variants around certain themes. It is therefore important to facilitate the task of *data structure transformation*, since this is perhaps the most common method for discovering new data structures. To do so, it is useful to think of a data structure not only in terms of its own operational semantics but also of the semantics of the algorithms that use it, the *clients*. Unfortunately, the "combinatorial" view of data structures often lacks this flexibility. Their semantics often do reflect the clients' needs, but too indirectly to be readily modified around them. This is not to say that modeling data structures after nice, elegant combinatorial objects is not the best route to good design: we believe that it is. Occasionally, however, useful transformations will be obscured or discouraged, because they seemingly get in the way of the combinatorial identity of the objects in question.

We propose a design discipline that involves looking at the *functionality* of data structures from a client's viewpoint. Informally, we extend the notion of a data structure by associating with each node v of a graph, not a piece of data as is usually the case, but a function f_v (or several if needed) as well as a collection of data $S(v)$. To evaluate f_v, it is sufficient, yet not always necessary, to have available $S(v)$ and some values of the functions associated with nodes adjacent to v. For expressiveness, we allow f_v to be a higher-order function (i.e., a function that includes other functions among its arguments). This definition is incomplete but gives the gist of what we will call a *functional data structure* (FDS). In essence, it is a communication scheme for routing transfers of information among a collection of functions. Allowing stepwise refinement is an essential feature of an FDS. Refining an FDS is to replace nodes by other FDS's iteratively, and in the process, get closer and closer to a data structure in the traditional sense.

An important consequence of this setting is to release the data from memory assignment. Indeed, the pairing $(v, S(v))$ is virtual and need not correspond to any physical reality. This should not be equated with, say, the independence of linked lists with regard to memory allocation. The abstraction provided by a functional data structure is much stronger. Indeed, it will not even be required that values be physically stored in the records with which they are associated. The only requirement is that these values should be available somehow whenever needed; in particular when the functions to which they are arguments must be evaluated. Whereas a data structure emphasizes data and communication between data, an FDS emphasizes functions and communication between parameters of these functions. It can be argued, of course, that an FDS can be emulated by almost any data structure, so of what good can it be? What makes the consideration of FDS's worthwhile is that they hint at data structures without *imposing* them. They key benefit of this setting is to allow the designer to bring upon

storage the same effect that *call by need* and *lazy evaluation* are known to have on execution time (Henderson [H]); namely to ensure that only the minimum amount of information needed to evaluate a function in some given amount of time be provided. This leads to a useful *folding* transformation, which we will describe shortly. Before going any further, let us take a simple example to illustrate our discussion. Let f be a *search* function mapping a key q to some element $f(S, q)$ of a set S:

$$f(S, q) \equiv \textbf{if } t(S) \textbf{ then } r(S, q)$$
$$\textbf{else if } g(S, q) \textbf{ then } f(p(S), q) \textbf{ else } f(S \backslash p(S), q).$$

If S is too small, i.e., $t(S)$ is true, then $r(S, q)$ provides the answer directly. Otherwise the function recurs with respect to either a subset $p(S)$ of S or its complement $S \backslash p(S)$, depending on the outcome of the predicate $g(S, q)$. This definition suggests an infinite binary tree $F(S)$ as an appropriate FDS for f. Each node v is associated with some set $S(v)$ obtained by applying to S compositions of $\lambda(s)p(s)$ and $\lambda(s)\{s \backslash p(s)\}$ intermixed in the obvious way.[2] For example, the root is associated with S, its left child to $p(S)$, and its right child to $S \backslash p(S)$. The tree can be made finite by ignoring all nodes associated with empty subsets. Next, one associates an FDS for $\lambda(q)g(S(v), q)$ with each node v of the tree. As is often the case in multidimensional searching, g may have a definition similar to f, e.g.,

$$g(V, q) \equiv \textbf{if } t'(V) \textbf{ then } r'(V, q)$$
$$\textbf{else if } h(V, q) \textbf{ then } g(p'(V), q) \textbf{ else } g(V \backslash p'(V), q).$$

This leads to an FDS, $G(V)$, defined for $V \subseteq S$ with respect to g similarly to $F(S)$. Putting things together, we refine $F(S)$ by associating $G(S(v))$ with each node v of $F(S)$. If p and p' are very different functions then it is not clear what benefits might be gained from this formalism. But suppose that it is possible to choose h so that p and p' are the same. Then f and g have identical "communication schemes." This allows us to perform a so-called *folding* transformation: the idea is to use the same FDS's for both f and g by taking each tree $G(S(v))$ and *folding* it over the subtree of $F(S)$ rooted at v. In effect, this is replacing $G(S(v))$ by an FDS for $\lambda(q)h(S(v), q)$. Once $F(S)$ has been refined in this manner, one will see a data structure with several layers of functional data structures superimposed on it. As a result, the storage used might be greatly inferior to what it would have been without folding the FDS. Indeed, each node v will thus only need an FDS for $\lambda(q)h(S(v), q)$ as opposed to an FDS for each $\lambda(q)h(V, q)$, where V is obtained by applying to $S(v)$ mixed compositions of $\lambda(s)p'(s)$ and $\lambda(s)\{s \backslash p'(s)\}$. To put things in perspective, we must recall that folding might entail redefining h: all benefits will be lost if the new implementation of g becomes much more complicated as a result. Although functional data structures have no complexity per se, they *hint* at the ultimate complexity of the data structure. Suppose that $g(S, q)$ can be evaluated very fast but at great cost in storage. Then folding offers the possibility of trade-off between space and time.

Searching in the past of mergesort. Here is a problem which, although a bit esoteric at first, is nevertheless fundamental. Run mergesort on a set of n distinct keys $S = \{y_1, \cdots, y_n\}$ and record the list produced after each merge. The resulting data structure can be modeled as a balanced binary tree T: the leaves are assigned the keys $y_1, \cdots, y_n$ from left to right, and each node v is associated with the sorted list $R(v)$ consisting

[2] We will use λ-expressions in the following either for sheer convenience as above or in order to distinguish between multivariate functions, such as $\lambda(x, y)f(x, y)$, and restrictions to univariate functions with constants, as in $\lambda(x)f(x, y)$.

of the keys at the leaves descending from v. For any q, define $s(v, q) = \min\{x \in R(v) \cup \{+\infty\} | q \leqq x\}$. What is an efficient data structure for computing $s(v, q)$, assuming that v is given by its inorder rank in the tree and that we are in a pointer machine model? Consider the data structure obtained by storing at v its inorder rank as well as a pointer to the root of a complete binary tree on $R(v)$. This is essentially Bentley's *range tree* [B2]: it allows us to compute $s(v, q)$ in $O(\log n)$ time and $O(n \log n)$ space (search for v in T, and then search for q in $R(v)$). Whether the storage can be reduced to linear and the query time kept polylogarithmic has been a long-standing open problem (Lee and Preparata [LP2]).

We settle this question by interpreting the range tree in functional terms. We have $s(v, q) \equiv f(v, \text{root}, q)$, with

$$f(v, z, q) \equiv \textbf{if } v = z \textbf{ then } g(R(v), q) \\ \textbf{else if } z < v \textbf{ then } f(v, z.r, q) \textbf{ else } f(v, z.l, q);$$

z is an ancestor of v or v itself, and $z.l$ (resp., $z.r$) denotes the left (resp., right) child of z. A natural FDS for f involves taking T and associating with each node v the function $\lambda(q)g(R(v), q)$ and the set $R(v)$. Since g is a *search* function it can be decomposed as follows (ignoring termination for simplicity):

$$g(V, q) \equiv \textbf{if } h(V, q) \textbf{ then } g(p(V), q) \textbf{ else } g(V \setminus p(V), q).$$

We can now refine the FDS for f by replacing each node v by an FDS for $\lambda(q)g(R(v), q)$. It is natural to think of p as a halving function taking a sorted set of numbers $\{p_1, \cdots, p_t\}$ as input and returning $\{p_1, \cdots, p_{\lfloor (t+1)/2 \rfloor}\}$ as output. This makes the implementation of h quite simple (one comparison), but unfortunately gives two widely different partitioning functions for f and g: one is based on the indices of the y_i's, the other on their values. So, one can try to make the two partitionings identical to allow *folding*, hoping that h will not become too complex as a result. To do that, we write $p(R(v)) \equiv R(v.l)$ (the function p becomes partial as a result, but it is all right). Instead of an FDS for $\lambda(q)g(R(v), q)$ at v, an FDS for $\lambda(q)h(R(v), q)$ now suffices.

Although by doing so, h increases in complexity, we must hope that h remains simpler to compute than g, otherwise the transformation would be useless. Since h is a predicate, it can be succinctly encoded as a bit vector $B(v)$. We have $B(v) = [b_0, \cdots, b_{|R(v)|-1}]$, where $b_i = 0$ (resp., 1) if the element at position $\neq i$ in $R(v)$ comes from the left (resp., right) child of v. Interestingly, $B(v)$ represents the transcript of pointer motions during the merge at node v. Note that if v is a leaf then $B(v)$ is not defined since that node, being associated with a single key, does not witness any merge. To compute $h(R(v), q)$ we proceed by binary search in $B(v)$. Unfortunately, we cannot do so with the bits of $B(v)$ alone, so we must find a way to produce the key $y^v(i) \in R(v)$ corresponding to a given bit position i in $B(v)$. This crucial operation is called *identifying* the bit b_i. Note that each bit of $B(v)$ has a distinct *identifier* in $R(v)$.

We can assume that v is not a leaf. If $b_i = 0$ (resp., $b_i = 1$) then we have $y^v(i) = y^{v.l}(j)$ (resp., $y^v(i) = y^{v.r}(k)$), for some j (resp., k). In other words, the key corresponding to b_i can be *traced* either at $v.l$ or $v.r$, depending on the value of b_i. We easily see that j (resp., k) is the number of 0's (resp., 1's) in $[b_0, \cdots, b_{i-1}]$. Suppose that

$$R(v.l) = [2, 3, 5, 7, 11, 13, 17, 19, 23, 27, 29, 31, 37, 41]$$

and

$$R(v.r) = [1, 6, 12, 14, 15, 20, 21, 24, 25, 26, 32, 33, 44, 46].$$

We have

$$R(v) = [1, 2, 3, 5, 6, 7, 11, 12, 13, 14, 15, 17, 19, 20, 21, 23,$$
$$24, 25, 26, 27, 29, 31, 32, 33, 37, 41, 44, 46]$$

and

$$B(v) = [1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1]$$

so, for example, 23 is the identifier of bit #15 in $B(v)$. Since this bit is 0, the key 23 can be found in $R(v.l)$. Its position is #8, which is also the number of 0's among the first 15 bits of $B(v)$, [1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1].

Iterating on this process until we reach a leaf of T allows us to identify any bit of $B(v)$. For this we assume that each leaf keeps its corresponding identifier alongside. Once identification is possible, we can compute $h(R(v), q)$ by binary search in $B(v)$. This will take $O(t(n) \log^2 n)$ time, where $t(n)$ is the time to find b_i and the sum $c_i = \sum_{0 \leq k < i} b_k$, given i. Computing $g(R(v), q)$ can then be done by binary search in T (following the definition of g and the fact that its FDS has been folded). We can take a shortcut, however, by observing that the value of $h(R(v), q)$ is given by a bit of $B(v)$ whose identifier, $g(R(v), q)$, is explicitly determined in the course of evaluating h. The computation of g will then also take $O(t(n) \log^2 n)$ operations.

Let us examine $t(n)$. To compute b_i and c_i, given i and v, we break up $B(v)$ into computer words $\beta_0, \cdots, \beta_{m-1}$ (filling each word up to $\lfloor \log n \rfloor$ bits) and we make the sequence $\beta_0, \cdots, \beta_{m-1}$ the leaves of a complete binary tree. At each internal node of the tree we indicate the number of 0's and the number of 1's among the bits of $B(v)$ stored at the leaves descending from its left child. With this data structure, it is easy to find in $O(\log n)$ time the word β_j that contains the bit b_i as well as the number of 1's or 0's among the bits of $\beta_0, \cdots, \beta_{j-1}$. At that point, it remains to examine each bit of β_j to complete the task. Bit enumeration on an EPM can be done very simply by searching for the number β_j in a perfectly balanced binary search tree of $2^{\lfloor \log n \rfloor}$ leaves and associating left turns with 0 and right turns with 1.

To conclude, $O(\log^3 n)$ time suffices to compute $s(v, q)$. To see that only linear storage is required, we remark that besides the input the number of bits used is proportional to the number of steps taken by running mergesort on n keys. This quantity is also proportional to the number of bits needed to store the input. Therefore, without any reference to the size of a computer word, we can conclude to the linearity of the data structure (in the sense that a data structure for a Turing machine is linear). This result holds on a pointer machine whose only operations are comparisons and additions. Note that whether the keys in S are integers or real numbers is irrelevant to our analysis. Also, one should observe that the use of bits need not be explicit in the algorithm. It will actually not appear at all in the implementation which we give in the next section.

It is easy to modify the data structure to produce an $O(n)$ space, $O(\log^4 n)$ time solution for range counting in two dimensions. But one can do much better. We must stop at this point to notice that the transformations which we have used so far are quite crude. They treat the recursive steps of the query-answering process as a sequence of independent computations. To enable the algorithm to use at any time information gathered at previous steps we must modify the refinement of the underlying functional data structure. The idea is to exploit the fact that the operations performed in the four nested loops of an $O(\log^4 n)$ time algorithm are unlikely to be independent. We feel that our point concerning the fruitfulness of functional data structures has been made, however, so we will not use the previous formalism any further. Rather, we will rely

on the intuition that it allowed us to develop and, in particular, we will keep the idea of folding in the back of our mind.

3. Range counting.

3.1. Range counting on a random access machine. We will describe a linear-size solution to range counting in two dimensions on a RAM with logarithmic query time. Although this result is not the strongest (the same holds on an APM as shown in the first table), it is very efficient in practice and illustrates most of the new ideas in simple terms. We have implemented the algorithm in Pascal (aiming for clarity rather than efficiency). The code is quite short, so we will give it in its entirety. For convenience, we assume that $n \geqq 2$ and that a word can store numbers not just in $[0, n]$ as we said earlier but in the range $[0, 4n \log^2 n]$. Then, rather amazingly, the data structure consists *solely* of four arrays X, Y, B, C: array $[0..n-1]$. The arrays X and Y contain, respectively, the x and y coordinates of the points in increasing order; B and C are a bit more mysterious. Here is an example. If the input is

$$\{(34, 3), (12, 1), (28, 23), (63, 15), (2, 35), (5, 17), (52, 43), (22, 13)\},$$

then we have

$$B = [64, 97, 81, 66, 33, 82, 34, 83]$$

and

$$C = [17, 35, 69, 6, 7, 9, 10, 12].$$

How do these numbers come about? We begin with an informal description of the algorithm.

To shorten the code given in this paper we make a number of simplifying assumptions, every one of which can be easily satisfied with a bit of care: (1) n is a power of two (pad with points at infinity, if needed); (2) all coordinates are distinct (go into rank space using presorting, if needed); (3) the query does not share coordinates with the point set and falls inside the smallest 2-range containing the input points.

It is convenient to leave aside the function $s(v, q)$ and use the related *ranking* function $r(v, q) = |\{y \in R(v) | y < q\}|$. To compute this function, we keep a similar data structure: each internal node v of T stores the bit vector $B(v)$, broken up into words $\beta_0, \cdots, \beta_{m-1}$, as usual. We also have an array $C(v)$, whose cell $\#i$ contains the number of 1's in $\beta_0, \cdots, \beta_i$. The problem is now to find the rank r of q in $R(w)$, assuming that we know its rank r' in $R(v)$ (without loss of generality, w is the right child of v). This rank is the number of 1's in $B(v)$ at positions $[0..r'-1]$. To find it, we must locate the corresponding bit in $B(v)$: first we find the word β_j that contains it, which is done in $O(1)$ time on a RAM, then we compute its relative position k in β_j. We also set r to the approximate count given by cell $\#(j-1)$ in $C(v)$. There now remains to shift β_j by k bits (done by division) and find the number of 1's in the remaining word by table look-up. Adding this number to r produces the desired result. So, after an initial binary search in the array $Y = R(\text{root})$, we find the rank of q in $R(\text{root})$, and percolate down to v. To carry out these computations, we need powers of two and tallies of ones for each integer in $[0, n-1]$. We will store all the $B(v)$'s and tallies in one array $B[0..n-1]$, and all the partial counts and powers of two in another array $C[0..n-1]$.

The data structure. The arrays $X[0..n-1]$ and $Y[0..n-1]$ give the x and y coordinates of the n points in increasing order. Let $\lambda = \log n$, $\mu = 2(1 + \lfloor \log \lambda \rfloor)$, and $M = 2^{\mu}$. In the code below, n, M, and λ (denoted lambda) will be global variables. Note that the word size w has been assumed to be at least as large as

$\lfloor \log n + 2 \log\log n + 2 \rfloor + 1 \geqq \lambda + \mu$. Let $P.\Lambda_i.M_i$ be the sequence of bits in $B[i]$, from left to right.

(1) P is a sequence of $w - \lambda - \mu$ zeros (padding).

(2) Λ_i is a sequence of λ bits, called the λ-part of $B[i]$. Since n is a power of two, we can nicely characterize each node of T by a pair (a, b), where a is its height (0 for a leaf, λ for the root) and b is its rank at height a, counted from left to right starting at 0. Sort all the nodes in lexicographic order and concatenate into one string $b[0..l]$, in this order, all the bit vectors $B(v)$ (recall that these vectors are defined only from height 1 up). The string of bits $\Lambda_0.\Lambda_1 \cdots \Lambda_{n-1}$ is set to $b[0..l]$. Note that the count of bits is the same in both cases ($l = \lambda n - 1$) but that the boundaries of $B(v)$'s might not coincide at all with those of λ-parts. In the following, we will use the notation $b[i]$ to indicate the $(i+1)$st bit of $b[0..l]$.

(3) M_i, called the μ-part of $B[i]$, contains the number of 1's in the binary representation of i (note that μ bits are ample for that purpose).

(4) The λ-part of $C[i]$ is equal to 2^i for $i \in [0, \lambda - 1]$. For each $i \in [0, \lambda - 1]$, let us (only in thought) turn to 0 each bit of the λ-part of $C[i]$. Then for each $i \in [0, n-1]$, $C[i]$ is the total number of bits equal to 1 among the λ-parts of $B[0..i]$. Note that the powers of two stored in the λ-parts never conflict with the tallies of ones because the number of 1's among the first λ λ-parts of $B[0..\lambda - 1]$ can be encoded over $\lfloor \log(\lambda^2) \rfloor + 1 \leqq \mu$ bits.

We will illustrate these definitions with the first cells of B and C. The mergesort starts with the sequence of points x-sorted (2, 35), (5, 17), (12, 1), (22, 13), $\cdots$, and merges {35}, {17} (which gives $b[0] = 1$ and $b[1] = 0$) and then {1}, {13} (which gives $b[2] = 0$ and $b[3] = 1$). As a result, the λ-part of $B[0]$ is 100. Since $\mu = 4$, the μ-part of $B[0]$ is expressed over 4 bits. It is equal to the number of 1's in 0, that is, 0000. This gives $B[0] = 1000000$ in binary, i.e., 64 in decimal. The λ-part of $C[0]$ is equal to 1. This number, shifted by μ bits, becomes 16, to which the number of 1's in $B[0]$ ($=1$) must now be added. This gives the final value $C[0] = 17$.

The preprocessing. We use a nonrecursive mergesort to fill the λ-parts of B. To do so, we declare a temporary array $T[0..n]$. Variables *step* and *l* form the lexicographic pair in the sort. A counter *index* maintains the current position in B. A bit is inserted to the right after multiplying the current word by two to make room for it. When a λ-part is full ("**if** lambda = fill **then begin**") it is shifted by μ bits to its final position, and the μ-part is computed by enumerating the bits of *index*. The powers of two, $\{2^0, 2^1, \cdots, 2^{\lambda-1}\}$ stored in the λ-parts of C are computed in an extra pass ("**for** index := 0 **to** lambda − 1 **do**"). Initially, X contains the x-coordinates in increasing order and $Y[i]$ is the y-coordinate corresponding to $X[i]$; B and C are set to 0, and maxint stands for any integer larger than the coordinates. At the end Y contains the y-coordinates in increasing order.

```
procedure Preprocessing;
    var fill, index, i, j, k, l, r, u, cur, tmp, step: integer;
begin
    fill := 0; index := 0; step := 2;
    while step <= n do begin
      l := 0;
        while l < n do begin
            r := l + step − 1; u := (l + r) div 2;
            for k := l to r do T[k] := Y[k];
```

```
                T[r+1] := maxint; i := l; j := u+1; k := l-1;
                while (i <= u) or (j <= r) do begin
                    B[index] := 2 * B[index]; k := k+1;
                    if (T[j] < T[i]) or (i > u) then
                        begin
                            Y[k] := T[j]; j := j+1;
                            B[index] := B[index]+1;
                            C[index] := C[index]+1
                        end
                    else begin Y[k] := T[i]; i := i+1 end;
                    fill := fill+1;
                    if lambda = fill then begin
                        cur := index; B[index] := B[index] * M;
                        while cur > 0 do begin
                            tmp := cur; cur := cur div 2;
                            B[index] := B[index]+tmp-2 * cur
                        end;
                        index := index+1; fill := 0;
                        if index < n then C[index] := C[index-1]
                    end
                end;
                l := l+step
            end;
        step := 2 * step
        end;
        fill := M;
        for index := 0 to lamda-1 do
            begin C[index] := C[index]+fill;
            fill := 2 * fill end
    end;
```

The query-answering algorithm. Let $[x_1, x_2]\times[y_1, y_2]$ be the query range. The function RangeCount (query) decomposes $[x_1, x_2]$ into $O(\log n)$ canonical pieces, each represented by a node of T. The count of points inside the query is obtained by summing up the values $r(v, y_2)-r(v, y_1)$ for each v in the decomposition. Since we add up differences, we may redefine the function $r(v, q)$ as follows. Let i be the number of elements in $R(v)$ strictly less than q. By construction, the $(i+1)$st bit in the vector $B(v)$ (if defined) appears somewhere in $b[0..l]$, say as $b[j]$; then set $\rho(v, q)=j$. Consistently, all the bits of the $B(v)$'s will be addressed from now on by their position in $b[0..l]$.

Next, we describe the functions needed in a bottom-up fashion. The function One (pos) returns the number of 1's in $b[0, \text{pos}-1]$. First, we find the index i such that $B[i]$ contains the bit $b[\text{pos}]$. To compute the number of 1's in $B[i]$ left of the bit $b[\text{pos}]$, we shift and truncate $B[i]$ accordingly to obtain the integer j (we use a little trick to avoid shifts by λ). Then we use the μ-parts of B to find z, the number of 1's in j. This number is added to $C[i-1]$ in order to get the final result. We need a corrective term if $i-1$ is less than or equal to $\lambda-1$ (because of the powers of two stored in the first cells of C).

```
function One (pos: integer): integer;
    var i, j, z: integer;
```

```
begin
    i := pos div lambda;
    j := B[i] div (2 * M * (C[lambda * (1+i) - pos - 1] div M));
    z := B[j] - M * (B[j] div M);
    if 0 < i then z := z + C[i-1];
    if (0 < i) and (i <= lambda) then z := z - M * (C[i-1] div M);
    One := z
end;
```

The function Newpos (dir, block, pos, width) allows us to trace the position of a bit b[pos] from a node v to a child w. In other words, it computes the values $\rho(w, q)$, given $\rho(v, q)$. (The reader familiar with Willard [W2] might guess rightly that we are trying here to simulate the effect of *downpointers*). We allow the value of $\rho(w, q)$ to be negative. Why is that so? The first cells of B store the history of the first pass of mergesort. Therefore, these do not correspond to leaves of T but to parents of leaves. From the lexicographic ordering in B, we can see that the leaves of T, if they were to be stored in B, would appear in positions $-n, -n+1, \cdots, -1$. We do not need to store these leaves because no identification is required for range counting. The arguments of Newpos (dir, block, pos, width) denote, respectively,

(1) *dir*: which child of v is being considered (0 if left, 1 if right); we use an integer instead of a Boolean only for brevity.
(2) *block*: the position in $b[0..l]$ of the first bit of $B(v)$,
(3) *pos*: the position in $b[0..l]$ of the bit of interest in $B(v)$,
(4) *width*: the number of leaves in the subtree rooted at v.

Because of the lexicographic ordering of the nodes of T, the starting position of $B(w)$ in $b[0..l]$ is "block $-n$" (left child) or "block $-n+$width/2" (right child).

```
function Newpos (dir, block, pos, width: integer): integer;
begin
    if dir = 0 then Newpos := pos - n + One(block) - One(pos)
    else Newpos := block - n + One(pos) - One(block) + (width div 2)
end;
```

The function Path (A, q) searches for q in the array A of type "Tableau = array $[0..n-1]$ of integer", and returns min $(\{i \mid q \leqq A[i]\} \cup \{n-1\})$. It will be used for two purposes: first to locate y_1 and y_2 in Y, and then x_1 and x_2 in X.

```
function Path (A: Tableau; q: integer): integer;
    var l, k, r: integer;
begin
    l := 0; r := n - 1;
    while l < r do begin
        k := (l + r) div 2;
        if q <= A[k] then r := k end
        else l := k + 1 end;
    Path := l
end;
```

Let l_1 and l_2 be the two leaves of T returned by Path when applied, respectively, to x_1 and x_2 in X. Let l_0 be the leaf preceding l_1 in inorder (such a leaf is guaranteed to exist because of our assumption that the query falls entirely within the smallest box containing the input points). The nodes of the decomposition of $[x_1, x_2]$ are precisely those adjacent to the path from l_0 to l_2, but not on it, whose inorder ranks fall in

between the ranks of these two leaves. To find them, it suffices to determine the least common ancestor w of l_0 and l_2, and then collect the nodes N_1 (resp., N_2) hanging off the right (resp., left) of the path from l_0 (resp., l_2) to w. The number of points inside the query range is

$$\sum_{v \in N_1} \rho(v, y_2) + \sum_{v \in N_2} \rho(v, y_2) - \sum_{v \in N_1} \rho(v, y_1) - \sum_{v \in N_2} \rho(v, y_1).$$

All four computations are carried out by calling the function Cum (cut, init, path, dir). Let us take the case of $\sum_{v \in N_1} \rho(v, y_2)$, for example. Then "cut" is the number of leaves descending from w, "init" is equal to $\rho(\text{root}, y_2)$, "path" to l_0, and "dir" to 0 (it is 1 for N_2). The bits of "path" indicate the sequence of turns from the root to l_0. Along with "cut," this allows us to determine w and, hence, the nodes of N_1.

```
function Cum (cut, init, path, dir: integer): integer;
    var pos, z, bit, block, cur: integer;
begin
    pos := init; z := 0; block := (lambda - 1) * n; cut := n;
    while cur >= 2 do begin
        bit := (2 * path) div cur - 2 * (path div cur);
        if (cur < cut) and (bit = dir) then
            z := z + Newpos(1 - dir, block, pos, cur);
        pos := Newpos(bit, block, pos, cur);
        cur := cur div 2; block := block - n + bit * cur
    end;
    Cum := z
end;
```

For completeness, we also give the code of the mainline, which is quite straightforward.

```
function RangeCount (x1, x2, y1, y2: integer): integer;
    var left, right, low, high, cut, z: integer;
begin
    low := Path(Y, y1) + (lambda - 1) * n;
    high := Path(Y, y2) + (lambda - 1) * n;
    left := Path(X, x1) - 1; right := Path(X, x2); cut := n;
    while (2 * left) div cut = (2 * right) div cut do cut := cut div 2;
    z := Cum (cut, high, left, 0) + Cum (cut, high, right, 1);
    RangeCount := z - Cum (cut, low, left, 0) - Cum (cut, low, right, 1)
end;
```

The description of the algorithm is now complete. Again, observe that the coordinates can be assumed to be arbitrary reals, if needed. Note that the ratio between the storage needed and the input size is only 2. In the context of this algorithm, the λ-parts of B cannot be compressed. This is not true of C, however: this array can indeed be shrunken arbitrarily by skipping chunks at regular intervals. The blanks can be *made up* by further work in B at query-answering time. This simple remark shows that the ratio can be brought arbitrarily close to 1.5, while increasing the query time only by a constant factor.

One might wonder how much dependent on the RAM model the algorithm really is. We will see that, although it loses some of its simplicity in the process, the algorithm can still be ported to a pointer machine.

3.2. Range counting on a pointer machine. To avoid the need for address calculations, we implement the tree T with pointers. Each node v is associated with a list $W(v)$ (assumed to be doubly-linked for convenience). Aside from pointers for the list

itself, each record of $W(v)$ has eight fields, B, C, C', D, E, F, G, H, made of one word each. The key specification is that, concatenated together, the B-fields of $W(v)$ form the bit vector $B(v)$. This statement must be clarified and refined a little. First of all, only the $\lambda = \lfloor \log n \rfloor$ least significant bits of a B-field will contain bits of $B(v)$. These are called the *meaningful* bits of the B-field. We will ignore the others. With this scheme each B-field, except possibly the last one, has precisely λ meaningful bits. As regards the last one, we may assume that its meaningful bits are right-justified, and that an extra record is added to indicate how many bits of the B-field are indeed meaningful. Let the subscript k refer to the kth record of $W(v)$. Then,

(1) C_k stores the number of 1's in $B_0, \cdots, B_k$ and C'_k the number of 0's.
(2) D_k stores the identifier of the least significant bit of B_k. Recall that this is the value of $R(v)$ in one-to-one correspondence with that bit.
(3) Consider the smallest value of $R(v)$ greater than or equal to integer q that appears as a D-field of $W(v)$; we define $n(v, q)$ as the address of the record containing this D-field if it exists, or as a null pointer if it does not. Let w (resp., w') be the left (resp., right) child of v; then E_k (resp., F_k) contains the address $n(w, D_k)$ (resp., $n(w', D_k)$), or more precisely a pointer to the relevant record. These fields are left blank if v is a leaf.
(4) G_k and H_k are not immediately needed and, for the sake of clarity, will be introduced later on.

The root of the tree receives a special treatment: we augment W(root) with a balanced search tree for R(root). This will allow us to compute the rank of a y-coordinate and thus get the query-answering process started. All the fields should be well motivated from the previous discussion, except perhaps for E and F. These fields play a role somewhat similar to Willard's *downpointers* [W2]. They provide a mechanism to avoid repeated binary searches when examining sequences of lists $W(v)$. Unlike downpointers, however, they do not span the whole range of values in $R(v)$, but an evenly distributed sample. Before proceeding with a description of the query algorithm, we should make the obvious observation that, as before, the data structure is linear in size. Furthermore, it can be constructed in $O(n)$ space and $O(n \log n)$ time, using mergesort as a basis for the algorithm.

To answer a query, we simulate the process described in the previous section. We start with a binary search in R(root) with respect to y_1 and y_2 and follow E- and/or F-fields appropriately. Let v be an internal node of T and w be one of its children (say, the left one, without loss of generality). We need to compute $r(w, q)$, given $r(v, q)$. We will show how to find the pair $(n(w, q), r(w, q))$, given $(n(v, q), r(v, q))$. This is called computing a *transition*. Let B_k be the B-field of the record at address $n(v, q)$. We assume for now the availability of a primitive operation tally (k, t) which takes the number $B_k = 0 \cdots 0.x_0 \cdots x_{\lambda-1}$ in binary and an index t $(0 < t < \lambda)$, and returns $x_0 + x_1 + \cdots + x_{t-1}$. By definition, the first $r(v, q)$ bits of $B(v)$ are contained in the B-fields of $W(v)$ preceding $n(v, q)$, and perhaps in a prefix of B_k. Using C_k and C'_k we can find the length of this prefix using $+, -$. Next, we call the tally function to compute the number of 1's in the prefix, which gives us in turn the number of 0's among the first $r(v, q)$ bits of $B(v)$, that is, $r(w, q)$. To compute $n(w, q)$, we just follow the pointer in E_k. Since each B-field has at most λ bits, $n(w, q)$ will be either the address of the record where we land, or it will be the address of its predecessor in the list $W(w)$. To find out we can use q and D-fields, or if we prefer, we can use $r(w, q)$ and C, C'-fields. We mention the latter method for the following reason: it is possible to compute a transition from $(n(v, q), r(v, q))$ to $(n(w, q), r(w, q))$ *without* prior knowledge of q. This will be used in the next section.

To implement tally(k, t), we use the fields G_k and H_k as well as the primitive *shift* available on an APM. This, we should recall, takes $i \leqq \lfloor \log n \rfloor$ as input and returns 10^i (1 followed by i 0's). It is needed to simulate a RAM at the word level in constant time. To do so, let $\alpha = \lfloor \log \lambda \rfloor + 1$, $\beta = \lfloor \log \alpha \rfloor + 1$ and $\gamma = \lfloor \log \beta \rfloor + 1$ (these quantities are computed by binary search in preprocessing). For any integer u $(0 \leqq u < n)$, we define the function $h(u, i, j) = \lfloor u/2^{\lambda - j} \rfloor - 2^{j-i} \lfloor u/2^{\lambda - i} \rfloor$ if $0 \leqq i < j \leqq \lambda$, and 0 otherwise: if $u = u_0 \cdots u_{\lambda - 1}$ in binary, then $h(u, i, j) = u_i u_{i+1} \cdots u_{j-2} u_{j-1}$. Next, we define a word $\Theta = t_0 \cdots t_{\lambda - 1}$ as follows: for $i = 0, \cdots, 2^\beta$, let $t_{i\gamma} \cdots t_{(i+1)\gamma - 1}$ be the number of 1's in the binary representation of i. To set up the fields $G_k = g_0 \cdots g_{\lambda - 1}$ and $H_k = h_0 \cdots h_{\lambda - 1}$, let $p_0 \cdots p_{\lambda - 1}$ be the λ meaningful bits of B_k (Fig. 1). Recall that the last B-field of $W(v)$ might be short of λ meaningful bits; this special case can be treated easily, however, and will be omitted here.

(1) For each $i = 0, \cdots, \lfloor \lambda/\alpha \rfloor - 1$, we have $g_{i\alpha} \cdots g_{(i+1)\alpha - 1} = p_0 + \cdots + p_{(i+1)\alpha - 1}$.

(2) For each $i = 0, \cdots, \lfloor \lambda/\alpha \rfloor - 1$ and each $j = 0, \cdots, \lfloor \alpha/\beta \rfloor - 1$, we have $h_{i\alpha + j\beta} \cdots h_{i\alpha + (j+1)\beta - 1} = p_{i\alpha} + \cdots + p_{i\alpha + (j+1)\beta - 1}$. Since an integer x can be represented on $\lceil \log(x+1) \rceil$ bits, one will easily check that the space provided for Θ and the various blocks of G_k and H_k is sufficient, for n large enough.

We can now implement tally(k, t). To do so, we evaluate in sequence

$$i = \alpha \left\lfloor \frac{t}{\alpha} \right\rfloor,$$

$$j = i - \alpha,$$

$$l = i + \beta \left\lfloor \frac{t - i}{\beta} \right\rfloor,$$

$$m = l - \beta,$$

$$p = h(B_k, l, t),$$

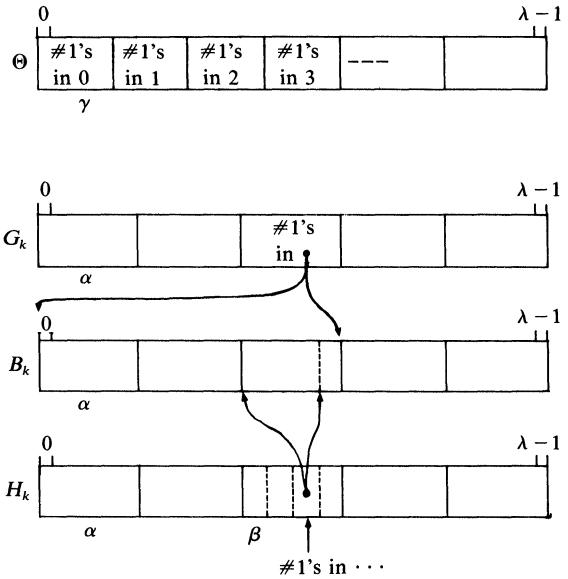


FIG. 1

then

$$\text{tally}(k, t) = h(G_k, j, i) + h(H_k, m, l) + h(\Theta, p\gamma, (p+1)\gamma).$$

The algorithm computes successive approximations of the answer. The vector $p_0 \cdots p_{t-1}$ is broken up into three blocks $p_0 \cdots p_{i-1}$, $p_i \cdots p_{l-1}$, and $p_l \cdots p_{t-1}$. The number of 1's in the first and second blocks are given directly by reading off one of the elementary pieces of G_k and H_k. For the last block, one turns to Θ for the answer.

To summarize, range counting can be done in $O(n)$ space and $O(\log n)$ query time on an APM. What happens if we only have an EPM? In that case, we may ignore the fields H and G altogether. Given any prefix of a B-field, we can count its number of 0's or 1's by examining each of its bits. To do so, we set up a binary search tree on the keys $\{0, \cdots, 2^\lambda - 1\}$ so as to identify the turns of a search with the binary representation of the search key. At most a constant number of subtractions might be needed for each node of T visited. These can be simulated in $O(\log n)$ time on an EPM. The query time will be $O(\log^2 n)$. This completes our discussion of the static case. The data structures described above for the various models form the skeleton of the remaining algorithms. We refer to them as *M-structures* (M for mergesort). They can all be built in $O(n \log n)$ time.

The dynamic case. We begin with a number of remarks which apply to all our subsequent treatments of dynamic problems. As we said earlier on, we will always assume that in the dynamic case the underlying model of computation is an EPM. One difficulty in dynamizing the previous data structure is trying to assign a value to λ. Since n changes constantly in a dynamic environment, so will λ if we set its value to $\lfloor \log n \rfloor$, as in the static case. This may imply that the entire data structure will need to be reconfigured at all times. To overcome this problem we relax the assignment of λ and only require that its value should lie between $\frac{1}{3} \log n$ and $\log n$ (for n larger than some appropriate constant). This is called the *compaction invariant.* The idea is that as soon as the compaction invariant ceases to be satisfied the entire data structure is to be rebuilt from scratch for a value of λ equal to $\lfloor \frac{2}{3} \log n \rfloor$. Clearly, no such reconstruction will be needed until after $\Omega(n)$ updates. Looking at a reconstruction as a sequence of insertions (at a polylogarithmic cost each), we will easily argue that the amortized cost of maintaining the compaction invariant is absorbed by the cost of individual updates. We now return to the specific problem of range counting in two dimensions.

A number of modifications to the M-structure are in order. To begin with, we replace fields C and C' by a dynamic search tree $C(v)$, e.g., a 2-3 tree (Aho et al. [AHU]), whose leaves are associated with the B-fields of $W(v)$. Each node (including the leaves) stores the number of meaningful 0's and the number of meaningful 1's in the B-fields of the leaves descending from it. This will essentially replace the use of C-fields while granting dynamic capabilities to the structure. Fields C, C', D, E, F, G, H are no longer needed. The most important modification is the emulation of a *variable* word-size. In general, B-fields will not be filled entirely with meaningful bits. A B-field β will have only a suffix of meaningful bits, that is, a contiguous sequence in right-justified position. The length of this suffix varies between 0 and λ. It can be determined by adding the counts of 0's and 1's associated with the leaf β in $C(v)$.

Invariant: If β is not the only B-field at node v, then it must have between $\lfloor \lambda/2 \rfloor$ and λ meaningful bits.

Without loss of generality, let w be the left child of v. We describe the computation of $r(w, q)$, given $r(v, q)$. Let β be the B-field that contains the bit of $B(v)$ at position $r(v, q)$. We can find β by using the counts of 0's and 1's stored at each node of $C(v)$

as partial ranks, and performing a simple binary search. This is similar to finding a key in a search tree, given its rank. Once β has been found, the position p of the desired bit in the B-field follows immediately. Note that this may require a subtraction, which we can afford since it takes only $O(\log n)$ time on an EPM. As is often the case, however, p will only be used for comparisons, so it can be represented as a pair $(r(v, q), p')$, where $p' = r(v, q) - p$ is the rank determined by the binary search. In this way, no subtraction is needed. While doing the search, we will collect partial counts of 0's: this is a special case of one-dimensional range search. If we visit a node z and branch to its right child, we update the partial count by adding to it the number of 0's provided with the left child of z. This is straightforward, so we omit the details. Finally, at the end of the search, we add up to this partial count the number of 0's among the first p bits of β. To enumerate these p bits we use binary search. Note that it is important to know the number of meaningful bits in β; also the compaction invariant ensures that the tree needed for the enumeration remains of linear size. This gives us exactly $r(w, q)$. Clearly, a single transition takes $O(\log n)$ steps. Answering a query can then be done in $O(\log^2 n)$ time.

Next, we look at the problem of inserting or deleting a bit in a B-field β. We insert a new bit by locating its position in the B-field. To make room for the newcomer, we shift to the left the prefix of β ending at that position. Conversely, deleting a bit causes a right shift of a prefix of β. Every node of $C(v)$ on the path from the current leaf to the root must have its partial counts updated, that is, incremented by -1, 0, or 1. Although an EPM does not allow subtraction in constant time, we can get around this difficulty by maintaining a doubly-linked list of length n dynamically, each record storing its rank in the list. Instead of storing counts in $C(v)$, we store pointers to the record with the appropriate rank, which can always be done since no count can exceed n. This allows us to update counts in constant time while adding only a linear term to the size of the data structure.

What happens if the shift causes an overflow or an underflow? In the first case, we insert a new record in $W(v)$ and use the B-field of that record to absorb the overflow. Since we then have $\lambda + 1$ bits at our disposal, we clear the old B-field and allocate $\lfloor \lambda/2 \rfloor$ of these bits to it, the remaining $\lambda + 1 - \lfloor \lambda/2 \rfloor$ going into the new B-field. This causes the insertion of one node in $C(v)$. In case of underflow, we pick any adjacent B-field (one is always to be found since, by definition, no underflow can occur if $W(v)$ has a single record) and evaluate their combined number of bits. If it exceeds λ, then we reassign the two B-fields with each at least $\lfloor \lambda/2 \rfloor$ bits. Otherwise, we use a single B-field and discard the other one. If the tree $C(v)$ falls out of balance, we apply the necessary rotations to reconfigure it into an acceptable shape. Rather than going into details, let us give a general argument, to be used again later on, showing why all counts can be updated in $O(\log n)$ time. The main observation is that the Steiner minimal tree of all nodes whose counts need to be updated has size $O(\log n)$. We can then (conceptually) orient the edges of this subtree upwards and update every count by pebbling the resulting dag. Indeed, the counts at a node can be obtained from the counts of its children. See Knuth [K1] for variants.

Let us recap the sequence of operations for inserting a new point (x, y) into the structure. We start with a binary search in T with respect to x. This gives us the root-to-leaf path $v_1, \cdots, v_l$ at whose nodes an insertion must take place. At the root of T we perform a binary search in $R(\text{root})$ to compute $r(\text{root}, y)$, and then descend in each of the relevant nodes by successive transitions. We are then ready to insert a new bit in each of the selected lists $B(v_i)$. Note that the value of this bit is determined by whether v_{i+1} is the right or left child of v_i. Deletions are handled in a similar fashion.

Of course, we must deal with the possibility of the tree T falling out of balance. For this purpose, we use a weight-balanced tree, also known as a BB[α] tree. The idea, due to Lueker and Willard [L], [WL], [W2], is to rebuild the entire subtree rooted at the highest node going out of balance. This rebuilding is only expensive high up in the tree, but by chance weight-balanced trees go out of balance mostly at lower levels. With a proper choice of α, it can be shown (Willard and Lueker [WL]) that a node v cannot go out of balance twice before a number of updates at least proportional to the size of the subtree rooted at v has taken place among its leaves.

A simple counting argument reveals the performance of this scheme. We create on the order of $\log^2 n$ credits for every update, which we distribute evenly to each node v_i visited during the update. This should be enough to pay for the updates in each $C(v_i)$ and still leave on the order of $\log n$ credits to each node v_i. These remaining credits are stored in the *node-bank* of v_i. Suppose now that v_i falls out of balance. As we said earlier, after v_i last fell out of balance, at least $\Omega(|H|)$ updates must have occurred at the leaves of the subtree H rooted at v_i. For this reason, the node-bank of v_i will have available at least on the order $|H| \log n$ credits. Since constructing a data structure of size p takes $O(p \log p)$ time, these credits can pay for the entire rebuilding of H and of all the structures in $\{C(v) | v \in H\}$. The node-bank at v_i is thus emptied, but ready to be refilled with future updates taking place below it. This proves that insertions and deletions can be performed in $O(\log^2 n)$ amortized time. A more complex strategy (Willard and Lueker [WL]) can be invoked to spread the rebalancing over all the updates in an even manner and obtain an $O(\log^2 n)$ worst-case update time.

We close this section with the problem of maintaining the compaction invariant. Our discussion will apply to range counting as well as to all the other dynamic problems considered later on. For this reason, we make the general assumption that any update among n active input elements requires $O(\log^c n)$ amortized time (c is an arbitrary nonnegative constant). Between two consecutive reconstructions, assume that each update generates a number of credits equal to $\log^c n$, where n is the input size at the time of the update. We will show that these credits can cover the cost of the second reconstruction (up to within a constant factor). Let p (resp., n) be the size of the input during the first (resp., second) reconstruction. We have either $\lfloor \frac{2}{3} \log p \rfloor < \frac{1}{3} \log n$ or $\lfloor \frac{2}{3} \log p \rfloor > \log n$. In the former case, we derive that $n > p^2/8$, therefore at least $n - p = \Omega(n)$ insertions must have taken place between the two reconstructions. The credits generated thus amount to $\Omega(n \log^c n)$, which covers the cost of the second reconstruction. In the latter case, $\lfloor \frac{2}{3} \log p \rfloor > \log n$, we have $p > n^{3/2}$, which implies that at least $p - n = \Omega(n^{3/2})$ deletions occurred. The credits released by these deletions trivially cover the $O(n \log^c n)$ cost of reconstruction. Note that in both cases the credits released also cover the cost of maintaining auxiliary structures, such as the tree of size $O(2^\lambda)$ used for bit enumeration.

THEOREM 1. *Range counting in two dimensions can be done in $O(n)$ space and $O(n \log n)$ preprocessing time. In the static case, the query time is $O(\log n)$ on an APM. On a RAM the storage used is at most twice the number of words required to store the input. In the dynamic case, query and update times are $O(\log^2 n)$ on an EPM.*

4. On the process of identification. As we will quickly find out, identifying the point associated with a given bit in $B(v)$ is the computational bottleneck of most of the problems to be considered later on. For this reason, let us look at this process in some detail. Recall that computing a transition from $(n(v, y), r(v, y))$ to $(n(w, y), r(w, y))$ can be done without knowledge of y. This is convenient because it allows us to characterize any bit of $B(v)$ by a pair (n_v, r_v), knowing that the correspond-

ing bit in $B(w)$, denoted (n_w, r_w), can be computed directly from (n_v, r_v). This is the essential step for identification on a pointer machine.

LEMMA 1. *Let v be a node of T and let b denote an arbitrary bit of $B(v)$, specified by a pair of the form (n_v, r_v). Computing a transition from b can be done in constant time on an APM. In the dynamic case, it can be done in $O(\log n)$ time on an EPM.*

Proof. We review the basic steps of the algorithms, most of which we have already seen. To begin with, we determine the value of the bit b of $B(v)$ corresponding to (n_v, r_v). On an APM, we can find the position of the bit in the appropriate B-field in constant time, using C, C'-fields. Then we call on *shift* and $\div$ to find b. Recall that the ith most significant bit of a λ-bit word x is $h(x, i-1, i) = \lfloor x/2^{\lambda-i} \rfloor - 2\lfloor x/2^{\lambda-i+1} \rfloor$. Next, we set w to be the left child of v if $b = 0$, or the right child if $b = 1$. We are now ready to compute the transition from b, which is done exactly as described in the previous section. In the dynamic case, we use the tree $C(v)$ to locate b in $B(v)$ and then compute its location in $B(w)$. A single transition takes $O(\log n)$ steps. □

The power or random access allows us to do much better on a RAM. The idea is to store a little more information at scattered nodes of T in order to compute repeated transitions in one fell swoop.

LEMMA 2. *Let v be a node of T and b denote an arbitrary bit of $B(v)$ (at a known location), and let ε be any positive real. It is possible to modify an M-structure on a RAM so that identifying b can be accomplished in* (1) *$O(n)$ space and $O(\log^{\varepsilon} n)$ time; or* (2) *$O(n \log\log n)$ space and $O(\log\log n)$ time; or* (3) *$O(n \log^{\varepsilon} n)$ space and constant time. In all cases, the transformation of the M-structure can be accomplished in $O(n \log n)$ time.*

Proof. As usual in the static case we set $\lambda = \lfloor \log n \rfloor$. We define the *level* of a node of T as its number of ancestors. Let w be a descendent of v, and let b denote the bit of $B(v)$ corresponding to (n_v, r_v). How can we compute (n_w, r_w) from (n_v, r_v) in constant time? First note that w cannot be just any descendent of v, if $B(w)$ is to have a bit in correspondence with b. At a given level in T, there is a unique node w that satisfies the criterion. Furthermore, recall that on a RAM the consideration of n_v and n_w is redundant, since r_v and r_w supply all the needed information in useful and efficient form. We can fully characterize the bit b by a pair of the form (v, r). In practice, such a pair can be readily translated into an absolute address by adding r to the starting position of $B(v)$. Recall that this addressing at the bit level is virtual, but that it can be emulated in constant time on a RAM. By abuse of language, we will refer to b as "bit (v, r)." The problem at hand is that of computing a batch of consecutive transitions, and more precisely, the function $\theta(v, r, l) = (w, s)$, where (w, s) is the bit at level l in correspondence with (v, r). Whenever l is understood, we will say that bit $\theta(v, r, l)$ is the *companion* of bit (v, r).

We show how to compute $\theta(v, r, l)$ in constant time, adding only a small amount of storage to the M-structure. We consider that l and v are both fixed and we assume that l is larger than the level of v. Let h be the difference between l and the level of v, and let the sequence $w_0, \cdots, w_{2^h-1}$ denote the descendents of v at level l (without loss of generality, we will assume that n is a power of two, so that v has exactly 2^h descendents at level l). As usual, $B(v)$ is broken up into words $\beta_0, \cdots, \beta_{m-1}$, each made of λ meaningful bits (except possibly for β_{m-1}, a special case that we will ignore in the remainder of this discussion). We cannot afford to attach with each bit of $B(v)$ the address of its companion, but we might still want to do so for a sample of $B(v)$; for example, to follow a known pattern, for the least significant bit of each β_i. There is a catch, however. Indeed, this could lead to a situation where, say, all the companions land in the same node at level l, thus providing useless information for most of the other bits of $B(v)$. To circumvent this difficulty, we do not link the bits to their

companions but to *neighboring bits* chosen in a round-robin fashion among the nodes at level l. We augment each β_i with a pointer γ_i and a bit vector Γ_i, defined as follows:

(1) The pointers γ_i allow us to jump from selected places in $B(v)$ to related places at level l. For any j such that $0 \leqq j < 2^h$, we say that bit (w_j, s) is the *j-companion* of bit (v, r) if its identifier is the smallest value in $R(w_j)$ that is at least as large as the identifier of (v, r). Note that there is a unique j for which the j-companion of (v, r) is also its companion. For each i such that $0 \leqq i < m$, we define γ_i as the address of the word in $B(w_{i \bmod 2^h})$ that contains the $(i \bmod 2^h)$-companion of the least significant bit in β_i. If there is no such companion, γ_i need not be defined.

(2) The bit vectors Γ_i allow us to correct the errors introduced by the γ_i's. Let (v, r) be as usual an arbitrary bit of β_i. Let $\theta(v, r, l) = (w_j, s)$, and let k be the largest integer less than i such that γ_k points to a word of $B(w_j)$. Suppose for the time being that k is well defined. Then it cannot be too far away from i because of the round-robin scheme. We easily verify that k is always at least as large as $i - 2^h$. The address in γ_k allows us to jump from v to w_j within some reasonable accuracy. To complete the computation, we need information to (1) compute k, given (v, r), and (2) go from γ_k to the actual companion of (v, r). If (v, r) is the dth most significant bit of β_i, we define P_d as an $(h+1)$-long bit vector containing the binary representation of $i-k$ in right-justified position. This allows us to complete the first task. For the second one, we define Q_d as a $(\lfloor \log \lambda \rfloor + h + 3)$-bit vector. Let $d_0, d_1, d_2 \cdots$ be the bits of $B(w_j)$, broken down into words, δ_0, δ_1, δ_2, etc. Let δ_a be the word pointed to by γ_k, let d_b be the most significant meaningful bit of δ_a, and let d_c be the bit $\theta(v, r, l)$; we define Q_d as the $(\lfloor \log \lambda \rfloor + h + 3)$-bit vector containing the binary representation of $c - b$ in right-justified position (note that $c \geqq b$). We can verify that the number of bits provided by Q_d is sufficient for that purpose ($\#$ bits needed $= \lceil \log (c - b + 1) \rceil < \lfloor \log \lambda \rfloor + h + 3$). If k is not defined then P_d and Q_d store the values of j and c, respectively, with a flag to indicate so; again $\lfloor \log \lambda \rfloor + 2h + 4$ are more than sufficient for that purpose. Finally, we define Γ_i as the concatenated string $P_1.Q_1.P_2.Q_2 \cdots P_\lambda.Q_\lambda$. Note that Γ_i is $\lambda(\lfloor \log \lambda \rfloor + 2h + 4)$ bits long.

Computing $\theta(v, r, l)$ is now straightforward. To begin with, we find the index i of the word β_i that contains bit (v, r). This gives us the position d of (v, r) in β_i, which in turn leads to the two bit vectors P_d and Q_d. Note that both vectors can always be read in constant time. With P_d in hand, we can retrieve k as well as the address stored in γ_k. Using now Q_d as a relative address, we automatically gain access to $\theta(v, r, l)$. All these operations take constant time on a RAM. So, we have devised a technique for computing repeated transitions in constant time. But at what cost in storage? With the notation used above, the overhead amounts to one word γ_i and one bit vector Γ_i for each word β_i of $B(v)$. This gives an average of $O((h + \log \lambda)/\lambda)$ extra words per bit of $B(v)$, hence a grand total of $O(((h + \log \lambda)/\lambda)n + 2^{l-h})$ space, summing up over every node v at level $l - h$. Recall that there are exactly n bits in all the $B(v)$'s combined, at any given level. Of course, Γ_i will be organized as an array of words. Because each Γ_i is of the same length, these can be packed into one large array in order to simulate a two-dimensional matrix. To summarize, we have a scheme for computing h consecutive transitions in constant time from level j, using $O(((h + \log \lambda)/\lambda)n + 2^j)$ added space.

What is the time needed to construct the set of γ_i and Γ_i? With the random access features of a RAM, it is not difficult to carry out the preprocessing in $O(n)$ time. We

briefly describe the procedure. Assuming that for each v at level $l-h$ the array $R(v)$ is available, we can compute the companion of each bit of $B(v)$ in constant amortized time. To do so, we compute $R(w_0), \cdots, R(w_{2^h-1})$, using bucket sort to place each element in its right set. This is possible because, if we work in rank space, the points of $R(v)$ can be partitioned by vertical slabs of the same width, each corresponding to a distinct $R(w_j)$. Knowing the companions of the bits of $B(v)$ serves two purposes: one is to provide the position of each companion in its list $R(w_j)$, and the other is to reveal the name of the node in question. With the first piece of information we compute Q_d, and with the latter we derive P_d. To do so, we compute the round-robin assignment of the bits of $B(v)$ and form $O(2^h)$ sorted lists of y-values, which we merge with $R(w_0), \cdots, R(w_{2^h-1})$, in turn. This gives us all the γ_i's in linear time. To compute the P_d's, it suffices to scan $B(v)$ and for each bit deduce P_d from the index j of the node w_j associated with its companion. Finally, Q_d can be found by simulating the computation of $\theta(v, r, l)$ for each bit of $B(v)$. We can obtain Q_d as soon as it is needed in the computation, since we already know the value of $\theta(v, r, l)$. These explanations are more intuitive than formal, but we do not judge it necessary to dwell on this technical but simple aspect of the algorithm. For convenience, we denote the data structure which we have just described, $\Theta(l-h, l)$. In general, $\Theta(a, b)$ is a structure that allows us to compute transitions from level a to level b in constant time, using $O(((b-a+\log\lambda)/\lambda)n+2^a)$ extra space. Incidentally, there is an obvious discrepancy with regard to this bound. If we set $b-a=1$, then we have just described a method for computing constant time transitions, while using an extra $\log\log n$ bits for each bit of $B(v)$! Of course, this result is useless since we already know how to compute constant time transitions with only constant overhead in space. We will therefore save the more complex method only as a way to batch many consecutive transitions together.

We are now ready to attack the main question: how to identify (v, r)? Let m be the level of v ($m<\lambda-1$); the idea is to express m in some appropriate number system. We will build several data structures on the same model. Let $D(x, y)$ denote a data structure for jumping from any level l_1 to any level l_2, with $\lambda-y\leqq l_1<l_2\leqq\lambda-x$. Identification can be done by using $D(0, \lambda)$. We define $D(x, y)$ in a recursive manner (for notational convenience we assume, without loss of generality, that $x=0$). Let $\alpha(y)$ be an integer function of y such that $0\leqq\alpha(y)\leqq y/2$. The data structure $D(0, y)$ is made of

$$\Theta(\lambda-y, \lambda)\cup\left\{D(0, \alpha(y)-1), \cdots, D(j\alpha(y), (j+1)\alpha(y)-1), \cdots, D\left(\left\lfloor\frac{y}{\alpha(y)}\right\rfloor\alpha(y), y\right)\right\}.$$

We stop the recursion at some threshold ν, which is to say that $D(a, b)$ is null for $b-a\leqq\nu$. How do we physically store these structures? We want to emulate a situation where each $\Theta(i, j)$ is accessible as defined earlier, that is, as though it were the only one attached to level i. Unfortunately, many $\Theta(i, j)$'s are bound to be assigned to the same level i. We solve this difficulty by storing the $\Theta(i, j)$'s as two-dimensional matrices (as before), and keeping an entry table $E[0..\lambda+1, 0..\lambda+1]$ defined as follows: $E[i, j]$ stores the address of the first entry in $\Theta(i, j)$, if $\Theta(i, j)$ appears in the full expansion of $D(0, \lambda)$; it stores 0 otherwise. The storage $S(n, y)$ occupied by $D(0, y)$ follows the recurrence

$$S(n, y)=\left\lfloor\frac{y}{\alpha(y)}\right\rfloor S(n, \alpha(y))+S\left(n, y+1-\left\lfloor\frac{y}{\alpha(y)}\right\rfloor\alpha(y)\right)+O\left(\frac{y+\log\lambda}{\lambda}n\right),$$

and $S(n, y)=0$, for $y\leqq\nu$ (threshold below which we stop the recursion). We have

omitted the term $2^{\lambda-y}$ to be able to treat the recurrence uniformly. We will see later on the effect of this omission. Let $\alpha^{(0)}(y)=y$ and, for any $i>0$, let $\alpha^{(i)}(y)=\alpha(\alpha^{(i-1)}(y))$. Finally, let $\alpha^*(y)=\max\{i|\alpha^{(i)}(y)\geqq\nu\}$. We derive the relation

$$S(n,\lambda)=O\left(\left(\alpha^*(\lambda)+1+r\frac{\log\lambda}{\nu}\right)n\right). \tag{1}$$

We can easily show that taking into account the terms of the form $2^{\lambda-y}$ adds another term $O((\alpha^*(\lambda)+1)n)$, which asymptotically is immaterial. We briefly describe the preprocessing. We define the first *recursive layer* L_1 as the construction of $\Theta(0,\lambda)$. The second recursive layer L_2 involves building in this order

$$\Theta\left(0,\lambda-\left\lfloor\frac{\lambda}{\alpha(\lambda)}\right\rfloor\alpha(\lambda)\right),\cdots,\Theta(\lambda-(j+1)\alpha(\lambda)+1,\lambda-j\alpha(\lambda)),\cdots,$$
$$\Theta(\lambda-\alpha(\lambda)+1,\lambda),$$

and so forth, every time expanding each term with its recursive definition. If we look at the definition of $D(0,\lambda)$ as inducing a tree of height roughly $\alpha^*(\lambda)$, then L_i represents the work to be accomplished to build each node of the tree at level i. Starting with $R(\text{root})$, we can construct each layer in time $O(n|L_i|)$, where $|L_i|$ is the number of terms involved in the expression of L_i. This derives from our previous observation on the preprocessing of $\Theta(a,b)$. Note that when computing a Θ-structure the set of lists $R(v)$ needed for the computation can be obtained from the previous one in linear time. Since $\alpha(y)\leqq y/2$, the sum $\sum n|L_i|$ is a geometric series whose value is on the order of $(\lambda/\nu)n$. This gives a preprocessing time of $O(n\log n)$.

To identify (v,r), we begin by checking to see if m, the level of v, is equal to 0. If this is the case, we can then use $\Theta(0,\lambda)$ and conclude in constant time. If $m>0$, we compute the *starting interval* of the form $[j\alpha(\lambda),(j+1)\alpha(\lambda)-1]$ (or $[\lfloor\lambda/\alpha(\lambda)\rfloor\alpha(\lambda),\lambda]$) that contains $\lambda-m$. We leap from v to the companion of (v,r) at level $\lambda-j\alpha(\lambda)$ (with $j\leqq\lfloor\lambda/\alpha(\lambda)\rfloor$) by calling the algorithm recursively, using $D(j\alpha(\lambda),(j+1)\alpha(\lambda)-1)$ (or $D(\lfloor\lambda/\alpha(\lambda)\rfloor\alpha(\lambda),\lambda)$). When the recursion passes the cutoff point (i.e., is about to jump across fewer than ν levels) we complete the computation by taking transitions one level at a time. Of course, we assume that we have the appropriate data structure for doing so. When this is done, we jump from level $\lambda-j\alpha(\lambda)$ to level λ by taking intermediate steps. Let k be initially set to j.

(1) Take a regular transition from $\lambda-k\alpha(\lambda)$ to $\lambda-(k\alpha(\lambda)-1)$;
(2) Jump from $\lambda-(k\alpha(\lambda)-1)$ to $\lambda-(k-1)\alpha(\lambda)$, using $D((k-1)\alpha(\lambda),k\alpha(\lambda)-1)$;
(3) Decrement k by 1. If $k>0$ then go to step (1), else stop.

Let $Q(n,m)$ be the time to identify (v,r), that is, to jump across $\lambda-m$ levels. We immediately find that

$$Q(n,m)=O\left(\nu+\sum_{\nu\leqq\alpha^{(i)}(\lambda)\leqq\lambda}\frac{\alpha^{(i)}(\lambda)}{\alpha^{(i+1)}(\lambda)}\right). \tag{2}$$

Setting $\alpha(y)=\lfloor y/2\rfloor$ and $\nu=1$, from (1) and (2) we obtain $S(n,\lambda)=O(n\log\log n)$ and $Q(n,m)=O(\log\lambda)=O(\log\log n)$. This completes the proof of the second part of Lemma 2. Let us now set $\alpha(y)=\lfloor y/\lambda^{\varepsilon}\rfloor$ and $\nu=\log\lambda$. We derive $S(n,\lambda)=O(n)$ and $Q(n,m)=O(\log^{\varepsilon}n)$, which gives us the first part of Lemma 2.

To prove the last part of Lemma 2, we must modify the overall design of the data structure: $D(0,y)$ will now be made not only of

$$\left\{D(0,\alpha(y)-1),\cdots,D(j\alpha(y),(j+1)\alpha(y)-1),\cdots,D\left(\left\lfloor\frac{y}{\alpha(y)}\right\rfloor\alpha(y),y\right)\right\},$$

but also of

$$\left\{\Theta(\lambda-\alpha(y),\lambda),\cdots,\Theta(\lambda-j\alpha(y),\lambda),\cdots,\Theta\left(\lambda-\left\lfloor\frac{y}{\alpha(y)}\right\rfloor\alpha(y),\lambda\right),\Theta(\lambda-y,\lambda)\right\}.$$

As usual, $D(a, b)$ is null if $b-a\leqq\nu$. We assume that $\alpha(y)=\lfloor y/\lambda^{\varepsilon/2}\rfloor$ and $\nu=1$. The storage needed follows the relation

$$S(n,y)=\left\lfloor\frac{y}{\alpha(y)}\right\rfloor S(n,\alpha(y))+S\left(n,y+1-\left\lfloor\frac{y}{\alpha(y)}\right\rfloor\alpha(y)\right)+O\left(\frac{y(y+\log\lambda)}{\alpha(y)\lambda}n\right),$$

and $S(n, y)=0$, for $y\leqq\nu$. This gives $S(n,\lambda)=O(n\log^{\varepsilon/2}n\log\log n)=O(n\log^{\varepsilon}n)$. Once again, it is easily verified that the extra terms 2^a omitted from the recurrence are absorbed asymptotically. The construction time is $O(n\log n)$ since there are only a constant number of layers. To identify (v, r), we proceed recursively within the starting interval. The key difference now is that at a given recursion level, only a single Θ-structure need be used. The time thus becomes proportional to the number of recursive calls, which is constant. The proof of Lemma 2 is now complete. □

5. Range reporting. For this problem we use an M-structure with the various modifications described in the previous section. It is clear that to start out we might as well mimic the algorithm for range counting until all the bits in the appropriate $B(v)$'s have been located. These are the bits whose identifiers fall in the query range. They appear as $O(\log n)$ sequences of consecutive bits, each sequence being associated with a distinct node of T. We call these sequences the *decomposition vectors* of the query. To complete the computation, it suffices to identify each bit of the decomposition vectors. This can be done by applying the techniques of the previous section to each bit individually.

How do we dynamize the data structure? On an EPM, there is no difficulty adapting to the problem at hand our solution to range counting. Identification proceeds by applying repeated transitions, using the methods of Lemma 1.

This technique leads to the results stated in the tables of § 1, except for one minor difference: we have just proven a slightly weaker set of results obtained by replacing each occurrence of n/k by n in the tables. Obviously, to obtain the results claimed it suffices to treat, say, the case $k\geqq n^{2/3}$ separately. Indeed, if $k<n^{2/3}$ then $\log n$ and $\log(n/k)$ are within constant factors of each other. We will describe a dynamic algorithm on an EPM with a complexity of $O(n)$ space, $O(n\log n)$ preprocessing time, $O(\log n)$ update time, and for $k\geqq n^{2/3}$, $O(k)$ query time. Since k is not known beforehand the idea will be to dovetail between this method and the others (i.e., run the two competing algorithms in parallel). The first process to terminate will thus guarantee the upper bounds claimed in the tables. We define the alternative algorithm by combining standard data structuring techniques. Assuming for simplicity that all x-coordinates are distinct, we partition the set of n points by means of vertical slabs into groups of size between $p(n)$ and $4p(n)$, where $p(n)=\lfloor 2^{2\lfloor\log n\rfloor/3}\rfloor$. For each group of points we maintain a dynamic list containing the indices of the points sorted by y-coordinates. Given two query values y and y' we will use the list to report all the points in the group with y-coordinates between y and y'. Using, say, a 2-3 tree, it is easy to guarantee $O(k+\log n)$ query time (where k is the number of points to be reported) and $O(\log n)$ update time. The storage requirement is trivially linear. The utility of this data structure for the previous problem should be obvious. Given a query rectangle, our first task is to use the slabs to partition it into subrectangles, which is easily done in $O(\log n)$ time. If the query rectangle falls entirely within a single slab,

then we naively check each point in the slab. If the partition is effective, however, then each subrectangle can be handled by using the appropriate list (checking each element in the list in the case of the leftmost and rightmost subrectangles, and using the tree structures for the other subrectangles, if any).

The query time is easily shown to be $O(k+n^{1/3}\log n+n^{2/3})$, which is $O(k)$ if $k \geqq n^{2/3}$. Insertions and deletions are performed in logarithmic time by updating the relevant lists. To handle overflows and underflows we will assume for the time being that although n varies, $\lfloor \log n \rfloor$ and hence $p(n)$ remain the same. Let m be the size of a group where an insertion has just taken place. If $m > 4p(n)$ then break up the group into two subgroups of size $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$. If a deletion has taken place and $m < p(n)$ then let m' be the size of one of its adjacent groups (guaranteed to exist if n is larger than some constant). If $m+m' \leqq 3p(n)$ then merge the two groups into one, otherwise form two groups of size $\lfloor (m+m')/2 \rfloor$ and $\lceil (m+m')/2 \rceil$, respectively. In all cases the relevant lists are reconstructed from scratch. We easily check that each new group has its size $\Omega(p(n))$ away from the two limits $p(n)$ and $4p(n)$. A simple accounting argument can then be used to prove that an insertion/deletion has an $O(\log n)$ amortized complexity. Note that once $p(n)$ is known the only arithmetic needed is addition and subtraction by 1, which can be implemented in $O(\log n)$ time on an EPM. (Or even in $O(1)$, if we maintain a linked list from 1 to n and pointers to the values of the various group sizes). If now $\lfloor \log n \rfloor$ increases or decreases by 1, we simply rebuild the entire data structure from scratch, which is easily done in $O(n \log n)$ time (or even $O(n)$ using presorting). Note that to update the value of $p(n)$ at each update can be easily done in constant amortized time (on an EPM). When it is decided that a reconstruction is needed, prior to it we replace $p(n)$ by $q(n) = \lfloor 2^{2\lfloor \log (3n) \rfloor /3} \rfloor$, and always alternate back and forth between $p(n)$ and $q(n)$. In this way, we are guaranteed to have $\Omega(n)$ updates between two consecutive reconstructions. Obviously, everything we said earlier about $p(n)$ applies to $q(n)$ just as well. Since $p(n)$ and $q(n)$ can be computed in $O(n)$ time, using only additions, reconstructions still preserve the logarithmic amortized complexity of an update.

THEOREM 2. *Data structures for range reporting in two dimensions can be constructed in $O(n \log n)$ time in all cases. Let $k-1$ be the number of points to be reported, and let ε be any real > 0. On a RAM, a tradeoff is possible: in particular, we can achieve* (1) *$O(k(\log (2n/k))^{\varepsilon} + \log n)$ query time and $O(n)$ space*; (2) *$O(k \log\log(4n/k) + \log n)$ query time and $O(n \log\log n)$ space*; (3) *$O(k+\log n)$ query time and $O(n \log^{\varepsilon} n)$ space. On an APM, it is possible to achieve a query time of $O(k \log (2n/k))$, using $O(n)$ space. In the dynamic case on an EPM, it is possible to achieve a query time of $O(k(\log (2n/k))^2)$ and an update time of $O(\log^2 n)$ time, using $O(n)$ space.*

6. Range searching for maximum. As usual, we will make use of an M-structure, and we will start the algorithm by computing the decomposition vectors of the query. In general, these vectors will fail to fill whole sequences of B-fields. However, they can be broken up into three (or fewer) blocks, one of which corresponds to a sequence of B-fields, and the others to subparts. Discovering the identifier with maximum value in each block will readily lead to the final answer. Let v be an arbitrary node of T. To handle the first case, we add an M-field to each record of $W(v)$ to indicate the maximum value attained by the identifiers of bits in the corresponding B-field. Then we set up a data structure M_1 for doing one-dimensional range search for maximum with respect to the M-fields of $W(v)$. For the other blocks we need a data structure M_2 for computing maxval (β, k, l), a function that takes as input a B-field $\beta = x_0 \cdots x_{\lambda-1}$ and two bit positions $0 \leqq k \leqq l < \lambda$, and returns the position i of the bit x_i

whose identifier has maximum value among $\{x_k, x_{k+1}, \cdots, x_l\}$. We next show how to implement M_1 and M_2 in our various models of computation.

Implementation of M_1.

(1) *On a RAM.* We use a technique of Gabow, Bentley, and Tarjan [GBT] to reduce one-dimensional range search for maximum to the computation of nearest common ancestors in some special tree. To begin with, we construct a *Cartesian tree* with respect to the M-fields of each $W(v)$. This tree, discovered by Vuillemin [V], is defined as follows: pick the maximum M-field and store it in the root of the tree. Remove this maximum from the sequence of M-fields, and define the left (resp., right) subtree recursively with respect to the left (resp., right) remaining subsequence. Two important facts about Cartesian trees state that: (a) they can be built in linear time; (b) one-dimensional range search for maximum is reducible to the computation of the nearest common ancestor (nca) of two nodes in a Cartesian tree. Harel and Tarjan [HT] have given an efficient method for computing nca's. Their method allows us to implement M_1 in linear space and time, and obtain any answer in constant time.

(2) *On an APM or an EPM.* More simply, we use a complete binary tree with each node v holding the maximum value in the range spanned by the subtree rooted at v. This heap structure is of standard use for one-dimensional range search, so we need not elaborate on it. Query time is $O(\log n)$ in both models.

Implementation of M_2.

(1) *On a RAM.* We still use Cartesian trees, but in an indirect manner. The key observation is that for our purposes Cartesian trees do not need to be labelled. Therefore they can be represented by a canonical index over a number of bits $= O(\log \#\text{Cartesian trees of size } \lambda)$. We can get a rough estimate on this number by representing a p-node Cartesian tree as a string of matching parentheses for internal nodes, 1 for leaves and 0 for missing leaves, as in $(((10)((01)0))((01)1))$, for example. Missing leaves are the nonexistent brothers of single children (poor things!). There are p nodes and at most $p-1$ missing leaves, so $6p$ bits are sufficient for the encoding. Since the number of nodes we are dealing with is quite small, we can precompute all possible Cartesian trees and preprocess them for efficient nca computation. It will then suffice to store a pointer from each record in $W(v)$ to the appropriate Cartesian tree. Let αp be the number of words required to store a p-node tree preprocessed for constant time nca computation (Harel and Tarjan [HT]). Storing all Cartesian trees of size λ will thus require $4^{3\lambda}\alpha\lambda = \Theta(n^6 \log n)$ words! This is a bit too much, so we must resize the B-fields. We divide each B-field into subwords of at most $\lfloor \lambda/7 \rfloor$ bits each, and for each piece we keep a pointer to its appropriate $\lfloor \lambda/7 \rfloor$-node Cartesian tree. This will multiply the query time by a constant factor, but it will also bring down the space requirement to a more acceptable $O(n^{6/7} \log n) = O(n)$ bound. The time to compute maxval (β, k, l) is constant. But to identify the corresponding bit, we must resort to the techniques of Lemma 2. Since there are $O(\log n)$ decomposition vectors, we can achieve a query time of $O(\log^{1+\varepsilon} n)$ with $O(n)$ space; a query time of $O(\log n \log\log n)$ with $O(n \log\log n)$ space; or a query time of $O(\log n)$ with $O(n \log^{\varepsilon} n)$ space.

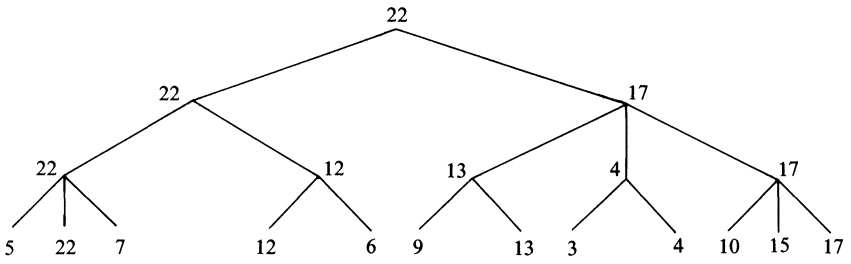
(2) *On an APM or an EPM.* We use the same technique, except for the preprocessing of Cartesian trees, now no longer needed. We compute an nca naively by examining the entire tree. This allows us to compute maxval (β, k, l) in

$O(\log n)$ time. The total cost of the computation will be $O(\log^2 n)$ on an APM and $O(\log^3 n)$ on an EPM.

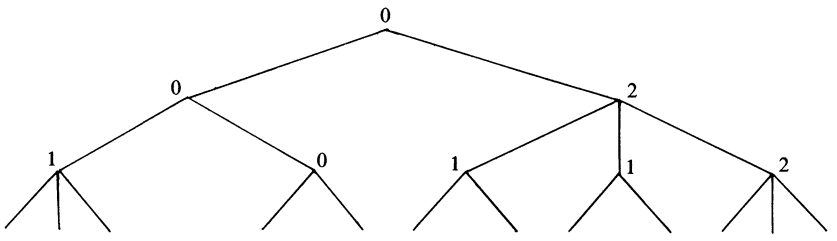
In all cases (including on a RAM with constant-time nca computation), the preprocessing can be done in linear time per level of T, that is, in a total of $O(n \log n)$ steps. This concludes our treatment of the static case.

The dynamic case. How can we dynamize these data structures on a pointer machine? The problem is quite similar to range counting, after all, so the proof of Theorem 1 can serve us as a guide. Certainly, the treatment of B-fields and the management of the structures M_1 can follow the same lines used for range counting. Difficulties arise with M_2, however. Consider the insertion or deletion of a bit in some B-field β. What is wrong with the following scheme? Find which subpart of β houses the bit in question and follow its pointer to the appropriate Cartesian tree. Then update the tree naively by considering each of its nodes. There are two basic problems: one is to update a Cartesian tree without knowing the values of the keys. Another problem, though less serious, is that these trees are shared among many pointers, so pleasing one pointer with an update would most likely upset many others. Once again, a functional approach will take us out of this dilemma.

Instead of a Cartesian tree, we use a dynamic one-dimensional range tree (similar to M_1); for example a 2-3 tree (Aho et al. [AHU]). Ideally, each node would store the maximum value associated with its leaves below (Fig. 2(a)). But this is as costly as having labelled trees, so we must seek a different solution. The key remark is that we need not store values in the nodes, but simply bits indicating where these values come from. These are called the *direction flags* of the tree. If the value at node z originates from a leaf of its leftmost subtree, the direction flag of z is 0; if it comes from the second subtree from the left, it is 1; else the direction flag is 2 (Fig. 2(b)).



(a)



(b)

FIG. 2

We can represent such a tree by means of balanced parentheses. Each leaf is a 1 and each internal node is a pair $(\cdots)$ immediately followed by its direction flag. For example, the tree of Fig. 2(b) is encoded as

$$(((111)1(11)0)0((11)1(11)1(111)2)2)0.$$

We represent each character on two bits: 0, 1, 2 as they are, "(" as 2, and ")" as 3, for example. Note that although 2 is used for different purposes there is no source of ambiguity, since 2 is always preceded by ")" and ")(" can never occur. Encoding and decoding any such tree can easily be done in time linear in its size. A leaf takes up 1 character and an internal node 3, so the tree associated with a B-field of λ meaningful bits can be encoded over $8\lambda - 6$ bits. Since every description starts with 10 there is no need for a boundary delimiter if the bits are right-justified. We will therefore assign 8 additional words for each B-field.

How can we use these trees to compute maxval (β, k, l)? To begin with, we decode the tree T_β associated with β. For (conceptual) simplicity, we might want to request $O(\lambda)$ temporary storage to construct a "useful" copy of T_β, that is, using pointers, one-word fields for direction flags, etc. We can now compute maxval (β, k, l) by performing one-dimensional range search in T_β. The canonical decomposition of $[k, l]$ can be computed in $O(\log \lambda)$ operations. This reduces the investigation to $O(\log \lambda)$ nodes of T_β. For each such node, we determine the leaf from which its associated value originates, and we identify the corresponding bit. This will take a total of $O(\log \lambda \log^2 n)$ time on an EPM.

Let us now look at insertions and deletions. We proceed as in the proof of Theorem 1. Suppose that no underflow or overflow occurs within the B-field β. In that case, we carry out the update in T_β alone. Intuitively, we can look at the tree in its entirety, but we should to try to minimize the number of bits that must be identified since these are the costly operations. The advantage of using a balanced tree is to limit the number of necessary identifications to $O(\log \lambda)$ (as opposed to $O(\lambda)$). For each update, we apply the standard insert or delete procedure for 2-3 trees. This can only upset the direction flags of the nodes that are effectively manipulated by the update as well as the direction flags of their ancestors. We promptly repair the damage caused to the direction flags by identifying the bits associated with the children of each of these nodes. This will take time $O(\log \lambda \log^2 n)$ on an EPM.

How do we deal with overflow or underflow? Following the proof of Theorem 1, the reconfiguration of the B-fields necessitates splitting or concatenating one or two trees. This can be done in $O(\log \lambda)$ steps [AHU]. Again we must ask: how many direction flags must be recomputed? A careful examination of the split and concatenate procedures on a 2-3 tree shows that after each application of these operations the number of nodes that either have been manipulated or are ancestors of nodes that have been manipulated is at most logarithmic in the size of the tree. Since these are the only ones whose direction flags might need updating, we conclude that the time spent to handle underflow and overflow is asymptotically the same as the time to insert or delete a new element.

As soon as a tree has been updated, we re-encode it as a bit vector and re-attach the result to the relevant B-field, discarding the old tree. Once M_2 has been computed, updating M_1 can be done in $O(\log n)$ operations on all models, assuming that it is implemented as a dynamic one-dimensional range search tree. The procedure is similar to the updating of $C(v)$ in the case of range counting (§ 3.2). It is also possible to unify the treatment of M_1 and M_2 and consider a single master tree with two modes

of representation: with pointers for M_1 (the top part) and with bit vectors for M_2 (the lower levels).

To summarize, it takes $O(\log^2 n \log\log n)$ time to carry an update with respect to a given node of T. Reconfiguring a subtree of T of size p can easily be done in $O(p \log p)$ steps: play a knock-out tournament at each level of T being reconstructed. The usual counting argument shows that the amortized cost of rebuilding T when it goes out of balance is $O(\log^2 n)$. Unfortunately this is largely dominated by the $O(\log^3 n \log\log n)$ steps incurred at the time of insertion or deletion.

THEOREM 3. *The data structures for range searching for maximum in two dimensions can be constructed in $O(n \log n)$ time in all cases. On a RAM, a tradeoff is possible: in particular, we can achieve* (1) *$O(\log^{1+\varepsilon} n)$ query time and $O(n)$ space*; (2) *$O(\log n \log\log n)$ query time and $O(n \log\log n)$ space*; (3) *$O(\log n)$ query time and $O(n \log^{\varepsilon} n)$ space. On an APM (resp., EPM) it is possible to achieve a query time of $O(\log^2 n)$ (resp., $O(\log^3 n)$), using $O(n)$ space. In the dynamic case on an EPM, it is possible to achieve query and update times of $O(\log^3 n \log\log n)$, using $O(n)$ space.*

7. Semigroup range searching. Of course, we assume that any value in the semigroup can be stored within one or, say, a constant number of computer words. We modify the M-structures by providing each node v with a complete binary tree whose leaves are associated with the B-fields of $W(v)$. Each leaf stores the semigroup sum of the values associated with the bits of its corresponding B-field. Similarly each internal node stores the sum of the values stored in its descending leaves. As in the case of range searching for maximum the algorithm computes the decomposition vectors of the query in a first stage. As usual, these are broken up into three (or fewer) blocks, one of which corresponds to sequences of B-fields, and the others to subparts. The semigroup sum associated with the identifiers of the first block can be readily obtained with two binary searches in the relevant auxiliary tree (this is a standard one-dimensional range search). For the other blocks, one must identify each of their bits to complete the computation, which will take a total of $O(t(n) \log^2 n)$, where $t(n)$ is the time taken to identify one bit. We can then use Lemmas 1 and 2 to derive the performance of the algorithms.

Executing updates is similar to the dynamic treatment of range search for maximum, so no further elaboration is necessary. The M_2-structures can be eliminated altogether, therefore updating the data structure will require $O(\log^2 n)$ identifications. Note that as usual the preprocessing requires $O(n \log n)$ time.

THEOREM 4. *The data structures for semigroup range searching in two dimensions can be constructed in $O(n \log n)$ time in all cases. On a RAM, a tradeoff is possible: in particular, we can achieve* (1) *$O(\log^{2+\varepsilon} n)$ query time and $O(n)$ space*; (2) *$O(\log^2 n \log\log n)$ query time and $O(n \log\log n)$ space*; (3) *$O(\log^2 n)$ query time and $O(n \log^{\varepsilon} n)$ space. On an APM it is possible to achieve a query time of $O(\log^3 n)$, using $O(n)$ space. In the dynamic case on an EPM, it is possible to achieve query and update times of $O(\log^4 n)$, using $O(n)$ space.*

8. Rectangle searching: counting and reporting. Using an equivalence result of Edelsbrunner and Overmars [EO], rectangle counting can be reduced to range counting. The basic idea is to subtract the number of rectangles which do *not* intersect the query from the total number of rectangles. Using inclusion–exclusion relations the problem reduces to a constant number of range counting problems. We immediately derive a result similar to Theorem 1. Note that the extra subtractions needed are inconsequential in the asymptotic complexity of the algorithms on an EPM.

THEOREM 5. *Rectangle counting in two dimensions can be done in $O(n)$ space and $O(n \log n)$ preprocessing time. In the static case, the query time is $O(\log n)$ on an APM. In the dynamic case, query and update times are $O(\log^2 n)$ on an EPM.*

As is well known (Edelsbrunner [Ed]), rectangle reporting in two dimensions can be reduced to three subproblems:

(1) *Range reporting.* See above.

(2) *Orthogonal segment intersection.* Given a set of horizontal segments in the plane and a (query) vertical segment q, report all intersections between q and the horizontal segments.

(3) *Point enclosure.* Given a set of 2-ranges in the plane and a query point q, report all the 2-ranges that contain q.

Given a set V of 2-ranges and a query rectangle q, rectangle reporting can be performed by doing (1) range reporting with respect to the lower left corners of the rectangles in V and the query q, (2) orthogonal segment intersection with respect to the bottom (resp., left) sides of the rectangles and the left (resp., bottom) side of q; (3) point enclosure with respect to the rectangles of V and the lower left corner of q.

Multiple reports caused by singular cases can be easily avoided using extra care. To complete our treatment of the static case, we simply observe that optimal data structures already exist for both the orthogonal segment intersection and the point enclosure problem (Chazelle [C1]). The size of these data structures is $O(n)$ and their construction takes $O(n \log n)$ time on an EPM. Queries can be answered in time $O(k+\log n)$, where k is the size of the output. Using our previous solutions to range reporting, we are then equipped to solve the problem at hand. The complexity is dominated by the cost of range reporting.

To deal with the dynamic version of rectangle reporting, we use a different approach. The previous sections involved a redesign of the *range tree* in linear space. We will now undertake a similar transformation with respect to the *segment tree*, a data structure due to Bentley [B1]. The idea again is to identify the functional components of the data structure and, on that basis, re-implement it completely differently.

The dynamic case. (1) *Orthogonal segment intersection.* For explanatory purposes, we make various simplifying assumptions. To remove them is tedious but does not affect the validity of our results. Our main assumption is that the coordinates of the segments, including the query, are all distinct. The problem to solve can be formulated as follows: let V be a set of n horizontal segments, each of the form (x_i, x_i', y_i), with $x_i < x_i'$. Given a vertical segment q compute all intersections between q and the horizontal segments.

We begin with the static version of the problem, and as a starter we introduce a little notation. The integer i is called the *index* of segment (x_i, x_i', y_i). For any j $(1 \leqq j \leqq 2n)$ let m_j denote the unique x-coordinate of rank j among the $2n$ endpoints. We are now ready to define the segment tree of V. For convenience, we use a declarative definition, as opposed to a more standard procedural definition (Bentley and Wood [BW]). Let T be a $(4n+1)$-node complete binary tree. For $i = 2, \cdots, 2n$, we put the ith leaf from the left in correspondence with both the endpoint m_i and the interval $(m_{i-1}, m_i]$. The leftmost leaf is in correspondence with m_1. Each internal node is associated with the union of the intervals at the leaves descending from it. We label each node v of T as follows: if v is the root, $b(v)$ is the null string; else $b(v)$ is the string of 0's and 1's given by the left (0) and right (1) turns of the path from the root to v.

Each node v is associated with a *node-set* $L(v)$. The segment (x_i, x_i', y_i) is represented in the tree T by including the index i into the node-sets of a collection of nodes

with label-set C_i. The set of indices associated with node v constitutes $L(v)$. Let l_i and r_i be, respectively, the leaf corresponding to x_i and the leaf immediately to the right of the one associated with x_i'. Let p be the longest common prefix of $b(l_i)$ and $b(r_i)$. We can always write $b(l_i)$ as a string $p0a_1 \cdots a_l$ of 1's and 0's, and $b(r_i)$ as a string $p1b_1 \cdots b_r$. We define

$$C_i = \{p0a_1 \cdots a_{j-1}1 | a_j = 0\} \cup \{p1b_1 \cdots b_{k-1}0 | b_k = 1\}.$$

Each interval $(x_i, x_i']$ is in this way *canonically* partitioned into intervals associated with nodes of T. Where our data structures for rectangle searching, which we call *S-structures*, differ from segment trees is in the encoding of node-sets. To begin with, we construct an M-structure (EPM version) with respect to the $2n$ endpoints of segments in V: to break ties between the y-coordinates of the endpoints of a segment, we may agree for consistency that the left endpoint precedes its right counterpart along the y-axis. We use T as the underlying tree of the M-structure.

Let us examine the relationship between $B(v)$ and $L(v)$. Recall that each bit of $B(v)$ is associated with a unique endpoint x_j or x_j'. In either case we say that the bit has *index j*. Without loss of generality, assume that v is the right child of node z. Then, by construction, for each i in $L(v)$ the endpoint x_i is associated with a leaf descending from z and therefore has a *trace* in $B(z)$. This allows us to encode $L(v)$ as a bit vector $A(v)$ of the same length as $B(z)$. The kth bit of $A(v)$ is 1 if and only if the index of the kth bit of $B(z)$ appears in $L(v)$. In this manner for every index in $L(v)$ there is a unique 1 in $A(v)$. All the other bits of $A(v)$ are set to 0.

Recall that $B(z)$ appears in compact form in $W(z)$ as a list of B-fields. We represent $A(v)$ exactly like $B(z)$ ($A(v)$ and $B(z)$ are isomorphic). Furthermore, we add pointers between the two lists to make it possible to jump directly between any B-field in $B(z)$ into its associated A-field in $A(v)$. To answer a query (x, y, y') is now straightforward. We perform a binary search to identify the leaf of T whose associated interval contains x (if any). Next, we search for y and y' in the virtual list $R(z)$ of each ancestor z of the leaf. Proceeding down from the root to the leaf, this can be readily done with the M-structure. After these preliminaries, we can locate both y and y' in each virtual list $L(v)$ by jumping directly from its virtual counterpart $R(z)$ (z is the parent of v). Let $\alpha_0, \alpha_1, \cdots$ be the A-fields of $A(v)$. The previous search leads to two bits, one in α_i and the other in α_j ($i \leqq j$). The problem is now to identify all the bits equal to 1 in each field of $L = \{\alpha_{i+1}, \alpha_{i+2}, \cdots, \alpha_{j-1}\}$, as well as in a certain suffix of α_i and prefix α_j. The latter part of the computation can be accomplished in $O(\log^2 n)$ time per report with $O(\log n)$ overhead. Assuming that L is not empty, retrieving the 1's among the fields of L is more difficult. We will say that α_l is *vacuous* if each of its bits is set to 0. We cannot just examine the fields $\alpha_{i+1}, \cdots, \alpha_{j-1}$, in search of all the 1's, since we would run the risk of visiting many vacuous fields. Instead, we link together the nonvacuous A-fields. For each relevant bit found we can return to its companion in $B(z)$ in $O(\log n)$ time, and then proceed to identify it, which takes another $O(\log^2 n)$ time.

We must now dynamize this static data structure. First, we describe the data structure and its invariants, and then we give an overview of the method used to insert a new horizontal segment (the case of a deletion is similar, so it will be omitted). The data structure is built from a dynamic M-structure (as in range reporting) augmented with the A-fields defined above. Recall that for each internal node z of T, we have a search tree $C(z)$, whose leaves are associated with the B-fields of $W(z)$. Let v be a child of z; we define $\mathcal{T}(v)$ to be a tree isomorphic to $C(z)$. The leaves of $\mathcal{T}(v)$ are in

one-to-one correspondence with the A-fields of $A(v)$ and, as before, these A-fields are isomorphic to the B-fields of $B(z)$ and connected to them via pointers. Instead of linking together nonvacuous A-fields, it suffices to keep flags at each node of $\mathcal{T}(v)$ which is the ancestor of at least one leaf corresponding to a nonvacuous A-field. With this scheme the position of each relevant bit in the nonvacuous fields at node v can be found in $O(\log n)$ time. As before, the identification of these bits will dominate the query time, bringing it up to $O(k \log^2 n)$.

How do we insert a new segment? To begin with, we take the two endpoints of the segment and insert them in the underlying M-structure. This involves the addition of two new leaves in T and the updating of $C(v)$ for each node v on the path from these leaves to the root. Note that for each such internal node and their left ($v.l$) and right ($v.r$) children we must update $\mathcal{T}(v.l)$ and $\mathcal{T}(v.r)$ to maintain the invariant of isomorphism with $C(v)$. Of course, the update of the flags and the A-fields will be different depending on whether $v.l$ or $v.r$ are nodes of the canonical decomposition of the new segment. It is a simple exercise to show that the updating can be performed in $O(\log n)$ time per node v. Thus updating includes checking with neighboring nodes of the new leaves if previously inserted segments may have to undergo changes in their canonical decomposition because of the new insertion. Deletions are symmetric to insertions and therefore quite similar. For details on dynamic operations on segment trees, consult Edelsbrunner [Ed] and Mehlhorn [Me].

The last part of any dynamic operation is to check whether the tree T has fallen out of balance. If that is the case, we apply the method used for the M-structure. This implies finding the highest node v of T that falls out of balance and reconstructing the entire subtree below v. The key remark is that all the segments involved in the updating must have at least one endpoint identified by a leaf descending from the parent of v. Therefore the cost of rebuilding the structure is once again $O(|H| \log n)$, where H is the subtree of T rooted at v. The usual counting argument shows that $O(\log^2 n)$ is an amortized upper bound on the update time.

As in the case of range reporting, we can cut down the query time to $O(k(\log (2n/k))^2)$ by dovetailing between the method above and a slab-based technique (§ 5). Instead of a dynamic list one will now use a dynamic *interval tree* (Edelsbrunner [Ed])—see also Mehlhorn [Me]. This data structure stores n intervals on the real axis in linear storage, so that all k intervals containing an arbitrary query value can be reported in $O(k+\log n)$ time. Edelsbrunner has shown, moreover, that insertions and deletions can be executed in $O(\log n)$ amortized time. Using the technique of slabs developed in § 5 we immediately obtain a dynamic algorithm for orthogonal segment intersection with a complexity of $O(n)$ space, $O(n \log n)$ preprocessing time, $O(\log n)$ update time and, for $k \geqq n^{2/3}$, $O(k)$ query time.

The dynamic case. (2) *Point enclosure.* This section will not make use of the compaction technique described earlier. The basic approach will be mostly borrowed from McCreight [Mc] and Edelsbrunner [Ed]. For this reason, we will only sketch the data structures, and refer to the aforementioned sources the reader who wishes to reconstruct all the proofs in full detail.

McCreight [Mc] has given a dynamic algorithm for point enclosure with the following performance: the storage needed is $O(n)$ and the query time is $O(k+\log^3 n)$, where k is the size of the output. We must extend his algorithm a little, however, because it handles only updates over a fixed universe. We assume that the reader is familiar with the concept of a *priority search tree* (McCreight [Mc]). For this reason we only briefly sketch the main features of this data structure. Given a set of n points in the plane and a fixed line L, a priority search tree allows us to report all the points

inside a query rectangle constrained to have one of its sides collinear with L. The complexity of the data structure is $O(n)$ space, $O(n \log n)$ preprocessing time and $O(k+\log n)$ query time, where k is the number of points to be reported. Furthermore any point can be inserted or deleted in $O(\log n)$ time.

Let V be a set of n 2-ranges in $\Re^2$. Consider a vertical line L that separates the collection of $2n$ vertical sides into two equal-size sets. We associate with the root of a binary tree the set of 2-ranges that intersect L, and to create the two subtrees below the root we iterate on this process with respect to the 2-ranges falling strictly to the left and strictly to the right of L. This defines a binary tree T of height $O(\log n)$. The reader familiar with Edelsbrunner [Ed] will recognize in T the interval tree of the projection of V on the horizontal axis. For each node v of T, clip the 2-ranges associated with v, using the dividing line at that node. This gives us a set of left ranges $R_l(v)$ and a set of right ranges $R_r(v)$. We construct two similar structures with respect to these collections. We restrict our description to $R_l(v)$. Apply exactly the procedure just described to $R_l(v)$ with respect to the horizontal (and not the vertical) direction. This leads to another interval tree-like structure, $T_l(v)$, whose nodes store lists of 2-ranges that are adjacent to both a horizontal and a vertical line. Given any point, finding the 2-ranges that contain it is equivalent to *domination searching* (Fig. 3). This problem involves preprocessing a set of points in the plane so that, for any query point q, finding which of the given points are dominated by q in both x and y coordinates can be done efficiently. But this is right up the alley of McCreight's priority search tree [Mc]. Putting all these simple observations together leads to a data structure of linear size for solving point enclosure reporting in time $O(k+\log^3 n)$, where k is the size of the output. Note that the cubic term comes from the fact that McCreight's structure is called upon on the order of $\log n$ times for each level of T.

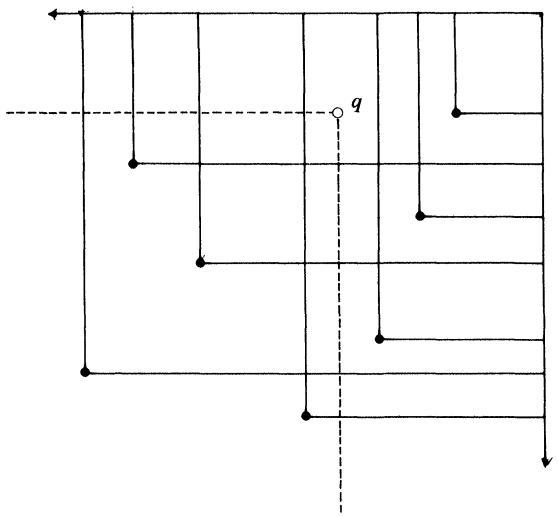


FIG. 3

Edelsbrunner [Ed] has shown that interval trees can be updated in $O(\log n)$ amortized time. Using the fact that a priority search tree on presorted points can be computed in linear time, we can use Edelsbrunner's method almost verbatim and derive

an algorithm for updating each $T_l(v)$ and $T_r(v)$ in $O(\log n)$ amortized time. A similar argument applied to T leads to an overall $O(\log^2 n)$ update time. We omit the details. Putting all these results together we have a linear-size data structure for dynamic reporting on an EPM with $O(\log^2 n)$ update time and $O(k \log^2 n)$ query time. Dovetailing with the alternative algorithms will cut down this query time to the minimum of $O(k \log^2 n + \log^3 n)$ and $O(k + n^{2/3})$, which gives $O(k \log^2 (2n/k) + \log^3 n)$. Note that a similar observation can be made in the static case. Theorem 6 summarizes our results on rectangle reporting.

THEOREM 6. *Data structures for rectangle reporting in two dimensions can be constructed in $O(n \log n)$ time in all cases. Let $k-1$ be the number of points to be reported, and let ε be any real >0. On a RAM, a tradeoff is possible: in particular, we can achieve* (1) *$O(k(\log (2n/k))^{\varepsilon} + \log n)$ query time and $O(n)$* space; (2) *$O(k \log\log (4n/k) + \log n)$ query time and $O(n \log\log n)$ space*; (3) *$O(k + \log n)$ query time and $O(n \log^{\varepsilon} n)$ space. On an APM (resp., EPM), it is possible to achieve a query time of $O(k \log (2n/k))$ (resp., $O(k(\log (2n/k))^2)$), using $O(n)$ space. In the dynamic case on an EPM, it is possible to achieve a query time of $O(k(\log (2n/k))^2 + \log^3 n)$ and an update time of $O(\log^2 n)$ time, using $O(n)$ space.*

9. Going into higher dimensions. Generalizing our result to any fixed dimension is straightforward. For range problems we use a classical technique of Bentley [B2]. Let $(x_1, \cdots, x_d)$ be a system of Cartesian coordinates in $\mathfrak{R}^d$. We build a complete binary tree with respect to the x_d-coordinates of the points. Each query range is thus decomposed into $O(\log n)$ canonical ranges, each of which can be handled by solving a range problem in $\mathfrak{R}^{d-1}$. The technique generalizes readily for the dynamic case (see Lueker [L], Willard [W2], Willard and Lueker [WL], for example).

As regards rectangle searching, we also follow a standard technique (Edelsbrunner and Maurer [EM]). See also Mehlhorn [Me]. The *left* coordinate of a d-range refers to its smaller coordinate in dimension x_d; its *x_d-interval* denotes the interval formed by projecting the d-range on the x_d-axis. Let R be a d-range and let x be its left coordinate. We define the *left* face of R to be the $(d-1)$-range obtained by projecting R on the hyperplane $x_d = x$. A d-range intersects another if and only if one intersects the other's left face. Since the two conditions are mutually exclusive, barring equalities among coordinates, we can use this criterion for both reporting and counting purposes. The data structure consists of a range tree and segment tree built respectively with respect to the left coordinates and x_d-intervals of the input. Each node has a pointer to a $(d-1)$-dimensional structure defined recursively. Once again, this generalization is sufficiently well known to avoid further elaboration.

Note that a problem in dimension d is in this way reduced to $O(\log^{d-2} n)$ problems in two dimensions. Analyzing the complexity in the nonreporting cases is straightforward. In the reporting problems considered earlier we obtained query times of the form $O(kf(n, k) + \log n)$, where $f(n, k)$ is decreasing in k and asymptotically equivalent to $f(n, 1)$ for $k = O(n^{3/4})$. To be more accurate, the query times could be expressed as the minimum of $O(kf(n, 1) + \log n)$ and $O(k + n^{2/3})$. This implies that in dimension d the query times $Q(n, k)$ are of the form

$$O\left(\log^{d-2} n + \sum_{1 \leq i \leq m} \min (k_i f(n, 1) + \log n, k_i + n^{2/3})\right),$$

where $\sum_{1 \leq i \leq m} k_i = k$ and $m = O(\log^{d-2} n)$. Let $k = A + B$, where $A = \sum_{k_i < n^{2/3}} k_i$ and

$B = \sum_{k_i \geq n^{2/3}} k_i$. We have

$$Q(n, k) = O(\log^{d-1} n + Af(n, 1) + B).$$

Note that $A < mn^{2/3}$. If $B \geq n^{3/4}$, then since m and $f(n, 1)$ are at most polylogarithmic we have $Q(n, k) = O(B) = O(k)$. If now $B < n^{3/4}$ we have $k = O(n^{3/4})$, and therefore $Q(n, k) = O(\log^{d-1} n + kf(n, k))$. In all cases, we have shown that $Q(n, k) = O(kf(n, k) + \log^{d-1} n)$.

THEOREM 7. *Each of the previous data structures (Theorems 1-6) can be extended to* $\mathfrak{R}^d$ $(d > 2)$. *The complexity of the resulting data structures can be obtained by multiplying each expression in the complexity for two dimensions by a factor of* $\log^{d-2} n$. (*Note: the terms involving k remain unchanged, but an extra term* $\log^{d-1} n$ *must be included in the query time*).

10. Discussion. We conclude with a few remarks and open problems.

The M-structures are essentially transcripts of sorting runs. We have used mergesort as our base sort but we could have turned to other methods as well. If we visualize the preprocessing played backwards and rotate the point set by 90 degrees, what we will see in action is no longer mergesort but quicksort (with median splitting). If we use radix-sort, we will see appear the possibility of tradeoffs between space and query time in higher dimensions, depending on the size of the radix. These simple observations show that the relationship between range search and sorting is rich, indeed, and deserves further investigation. An interesting open question is to determine whether there exist linear-size data structures with polylogarithmic response-time for solving range searching in *any* fixed dimension.

Our data structure for range counting in two dimensions is quite simple. It is very fast and surprisingly economical in its use of storage. Last but not least, it can be implemented with little effort. Unfortunately, this cannot be said of all the data structures in this paper. Some of our upper bounds are obtained at the price of a fairly heavy machinery. Can the algorithms be significantly simplified while keeping the same asymptotic complexity? Can their asymptotic performance be improved?

We believe that many of our bounds for dynamic multidimensional searching can be lowered if one has access to a more powerful machine than an EPM. For example, what improvements can the added power of a RAM buy us?

All the data structures in this paper rely crucially on the use of bit vectors. These have sometimes been used in previous data structures to encode sets—often as addresses in a RAM, like in (Gabow and Tarjan [GT]). Our use of bit vectors is quite different. The nodes of the M-structures, for example, store bit vectors that encode the history of a computation. This scheme is a departure from previous solutions and raises interesting questions concerning models of computation. It is evident from this work that even seemingly *minimal* models such as pointer machines with only comparison and addition can't avoid but allow exotic encodings of the sort. Indeed, it seems that only by placing very strong semantic limitations on the models of computation one might manage to invalidate these data structures. Let us observe that with a few notable exceptions (Tarjan [T], Chazelle [C2], for example) little is known on the computational power of pointer machines. Our results suggest that these might be, indeed, more powerful than expected.

Acknowledgments. I wish to thank the referee for several useful comments about this manuscript.

REFERENCES

[AHU] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[B1] J. L. BENTLEY, *Algorithms for Klee's rectangle problems*, Dept. of Comp. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1977, unpublished notes.

[B2] ———, *Multidimensional divide and conquer*, Comm. ACM, 23 (1980), pp. 214–229.

[BW] J. L. BENTLEY AND D. WOOD, *An optimal worst-case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., C-29 (1980), pp. 571–577.

[C1] B. CHAZELLE, *Filtering search: a new approach to query-answering*, this Journal, 15 (1986), pp. 703–724.

[C2] ———, *Lower bounds on the complexity of multidimensional searching*, Proc. 27th Annual IEEE Symposium on the Foundations of Computer Science, 1986, pp. 87–96.

[Ea] J. EARLEY, *Toward an understanding of data structures*, Comm. ACM, 14 (1971), pp. 617–627.

[Ed] H. EDELSBRUNNER, *Intersection problems in computational geometry*, Ph.D. thesis, Tech. Rep. F-93, Technical Univ. Graz, Graz, Austria 1982.

[EM] H. EDELSBRUNNER AND H. A. MAURER, *On the intersection of orthogonal objects*, Inform. Process. Lett. 13 (1981), pp. 177–181.

[EO] H. EDELSBRUNNER AND M. H. OVERMARS, *On the equivalence of some rectangle problems*, Inform. Process. Lett. 14 (1982), pp. 124–127.

[EOS] H. EDELSBRUNNER, M. H. OVERMARS, AND R. SEIDEL, *Some methods of computational geometry applied to computer graphics*, Computer Vision, Graphics, and Image Processing, 28 (1984), pp. 92–108.

[GBT] H. N. GABOW, J. L. BENTLEY, AND R. E. TARJAN, *Scaling and related techniques for geometry problems*, Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 135–143.

[GT] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 246–251.

[GoT] G. H. GONNET AND F. W. TOMPA, *A constructive approach to the design of algorithms and their data structures*, Comm. ACM, 26 (1983), pp. 912–920.

[G] J. GUTTAG, *Abstract data types and the development of data structures*, Comm. ACM, 20 (1977), pp. 396–404.

[HT] D. HAREL AND R. E. TARJAN, *Fast algorithms for finding nearest common ancestors*, this Journal, 13 (1984), pp. 338–355.

[H] P. HENDERSON, *Functional Programming: Application and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[K1] D. E. KNUTH, *The Art of Computer Programming, Sorting and Searching*, Vol. 3, Addison-Wesley, Reading, MA, 1973.

[K2] ———, *The Art of Computer Programming, Seminumerical Algorithms*, Vol. 2, Addison-Wesley, Reading, MA, 1981.

[L] G. S. LUEKER, *A data structure for orthogonal range queries*, Proc. 19th Annual IEEE Symposium on the Foundations of Computer Science, 1978, pp. 28–34.

[LP1] D. T. LEE AND F. P. PREPARATA, *Location of a point in a planar subdivision and its applications*, this Journal, (1977), pp. 594–606.

[LP2] ———, *Computational geometry—a survey*, IEEE Trans. Comput., C-33 (1984), pp. 1072–1101.

[Mc] E. M. MCCREIGHT, *Priority search trees*, this Journal, 14 (1985), pp. 257–276.

[Me] K. MEHLHORN, *Data Structures and Algorithms: Vol. 3, Multidimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[O] M. OVERMARS, *The Design of Dynamic Data Structures*, Springer Lecture Notes in Computer Science 156, Springer-Verlag, Berlin, New York, 1983.

[T] R. E. TARJAN, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

[V] J. VUILLEMIN, *A unifying look at data structures*, Comm. ACM, 23 (1980), pp. 229–239.

[W1] D. E. WILLARD, *Reduced memory space for multi-dimensional search trees*, 2nd Annual STACS, LNCS, Springer-Verlag, Berlin, New York, 1985, pp. 363–374.

[W2] ———, *New data structures for orthogonal range queries*, this Journal, 14 (1985), pp. 232–253.

[WL] D. E. WILLARD AND G. S. LUEKER, *Adding range restriction capability to dynamic data structures*, J. Assoc. Comput. Mach., 32 (1985), pp. 597–617.