SIAM J. COMPUT.
Vol. 17, No. 5, October 1988

003

DEFERRED DATA STRUCTURING*

RICHARD M. KARP†, RAJEEV MOTWANI†, AND PRABHAKAR RAGHAVAN‡

Abstract. We consider the problem of answering a series of on-line queries on a static data set. The conventional approach to such problems involves a preprocessing phase which constructs a data structure with good search behavior. The data structure representing the data set then remains fixed throughout the processing of the queries. Our approach involves dynamic or query-driven structuring of the data set; our algorithm processes the data set only when doing so is required for answering a query. A data structure constructed progressively in this fashion is called a *deferred data structure.*

We develop the notion of deferred data structures by solving the problem of answering membership queries on an ordered set. We obtain a randomized algorithm which achieves asymptotically optimal performance with high probability. We then present optimal deferred data structures for the following problems in the plane: testing convex-hull membership, half-plane intersection queries and fixed-constraint multi-objective linear programming. We also apply the deferred data structuring technique to multi-dimensional dominance query problems.

Key words. data structure, preprocessing, query processing, lower bound, randomized algorithm, computational geometry, convex hull, linear programming, half-space intersection, dominance counting

AMS(MOS) subject classifications. 68P05, 68P10, 68P20, 68Q20, 68U05

1. Introduction. We consider several search problems where we are given a set of n elements, which we call the *data set.* We are required to answer a sequence of queries about the data set.

The conventional approach to search problems consists of preprocessing the data set in time $p(n)$, thus building up a search structure that enables queries to be answered efficiently. Subsequently, each query can be answered in time $q(n)$. The time needed for answering r queries is thus $p(n)+r\cdot q(n)$. Very often, a single query can be answered without preprocessing in time $o(p(n))$. The preprocessing approach is thus uneconomical unless the number of queries r is sufficiently large.

We present here an alternative to preprocessing, in which the search structure is built up "on-the-fly" as queries are answered. Throughout this paper we assume that an adversary generates a stream of queries which can cease at any point. Each query must be answered *on-line*, before the next one is received. If the adversary generates sufficiently many queries, we will show that we build up the complete search structure in time $O(p(n))$ so that further queries can be answered in time $q(n)$. If on the other hand the adversary generates few queries, we will show that the total work we expend in the process of answering them (which includes building the search structure partially) is asymptotically smaller than $p(n)+r\cdot q(n)$. We thus perform at least as well as the preprocessing approach, and in fact better when r is small. We do so with no a priori knowledge of r. We call our approach *deferred data structuring* since we build up the search structure gradually as queries arrive, rather than all at once. In some cases we

* Received by the editors December 22, 1986, accepted for publication (in revised form) November 10, 1987.

† Computer Science Division, University of California, Berkeley, California 94720. The work of the first two authors was supported in part by the National Science Foundation under grant DCR-8411954. The results in § 4 first appeared in R. Motwani and P. Raghavan, *Deferred data structures: query-driven preprocessing for geometric search problems*, in Proc. 2nd Annual ACM Symposium on Computational Geometry, Yorktown Heights, NY, June 1986, pp. 303-312.

‡ IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. The work of this author was supported in part by an IBM Doctoral Fellowship while he was a graduate student at the Computer Science Division, University of California, Berkeley, California 94720.

show that our deferred data structuring algorithm is of nearly optimal efficiency, even in comparison with algorithms that know r, the number of queries, in advance.

In § 2 we exemplify our approach through the *membership query* problem. We determine the complexity of answering r queries on n elements under the comparison tree model. In § 3 we present a randomized algorithm for the membership query problem whose performance matches an information-theoretic lower bound (ignoring asymptotically smaller *additive* terms). We then proceed to exhibit deferred data structure for several geometry problems. In § 4 we show that deferred data structuring is optimal for the following two-dimensional geometric problems: (1) Given n points in the plane, to determine whether a query point lies inside their convex hull. (2) Given n half-planes, to determine whether a query point lies in their common intersection. (3) Given n linear constraints in two variables, to optimize a query objective function (also linear). Our algorithms are proved optimal by means of a tight lower bound (under the algebraic computation tree model) in § 4.4. In § 5 we consider *dominance problems* in d-space. We present theorems about the deferred construction of Bentley's ECDF *search tree* [2].

In this paper all logarithms are to the base two.

2. General principles of deferred data structuring. In this section we develop the basic ideas involved in deferred data structuring. Let $X=\{x_1, x_2, \cdots, x_n\}$ be a set of n elements drawn from a totally ordered set U. Consider a series of queries where each query q_j is an element of U; for each query, we must determine whether it is present in X.

If we had to answer just one query, we could simply compare the query q_1 to every member of X and answer the query in $O(n)$ comparison operations. This would be the preferred method for answering a small number of queries. On the other hand, if we knew that the number of queries r were large, we could first sort the elements of X in $p(n)=O(n \log n)$ operations, these building up a binary search tree T_X for the elements of X. We could then do a binary search costing $Q(n)=O(\log n)$ comparisons for each query; this takes $O((n+r)\cdot\log n)$ comparisons.

We proceed to determine the complexity (number of comparisons) of answering r queries on the set X; we do not know r a priori, and each query is to be answered before we know of the next one.

2.1. The lower bound. We first prove an information-theoretic lower bound for this problem.

THEOREM 1. *The number of comparisons needed to process r queries is at least* $(n+r)\cdot\log(\min\{n, r\})-O(\min\{n, r\})$ *in the worst case.*

Remark. Note that neither of the strategies mentioned above (linear search, or sorting followed by binary search) achieves this bound for all $r \leqq n$.

Proof. If we could collect the r queries and process them *off-line*, we would have an instance of the SET INTERSECTION problem where we have to find the elements common to the sets $X=\{x_1, x_2, \cdots, x_n\}$ and $Q=\{q_1, \cdots, q_r\}$. We will prove a lower bound of $\Omega((n+r)\cdot\log(\min\{n, r\}))$ comparisons for determining the intersection of two sets of cardinalities n and r. This off-line lower bound will hold a fortiori for the on-line case in which we are interested. We present the argument for the case $r \leqq n$; the other case is symmetrical.

Since we are interested in lower bounds on this problem, we can restrict our attention to only those cases where $X \cap Q=\emptyset$. In this case the algorithm has to determine the relation of each element in X to each element in Q. An adversary can ensure that for any two elements in Q there will be at least one in X whose value lies between them. In other words, the elements of Q will partition X into at least $r-1$ nonempty classes. Each such class will consist of all those members of X which lie between two consecutive values in the total ordering of Q. We shall give an information-theoretic lower bound by counting some ways of arranging X and Q to satisfy the above constraint.

There are $r!$ ways of ordering the elements in Q. Given a total order on Q, there are $(r-1)!$ ways of separating the elements in Q by some arbitrary $r-1$ elements from X. The remaining elements of X can be placed arbitrarily. There are $r+1$ available slots as determined by the r ordered elements of Q. This can be done in $(r+1)^{n-r+1}$ ways. Let I be the total number of interleavings (of X and Q) possible when $S \cap Q = \emptyset$. Then the number of possible arrangements specified above is a lower bound on the value of I:

$$I \geqq r! \cdot (r-1)! \cdot (r+1)^{n-r+1}.$$

Since the algorithm has to identify one out of (at least) this many possible arrangements the lower bound is given by $\log I$:

$$\log I \geqq (n+r) \cdot \log r - 2r \log e.$$

Here e represents the base of the natural logarithms. $\square$

2.2. Upper bounds. We now present two approaches to obtaining an upper bound which comes within a multiplicative constant factor of the lower bound. The first approach is based on merge-sort, while the second is based on recursively finding medians.

2.2.1. An approach based on merge-sort. The following algorithm comes within a constant factor of the lower bound. It uses a recursive merge-sort technique to totally order the elements in X. The merge-sort proceeds in $\log n$ stages. At the end of a stage the set X is partitioned into a number of equal-sized totally ordered subsets called *runs*. Each stage pairs off all the runs resulting from the previous stage and merges them to create longer runs. These stages are interleaved with the processing of a set of queries, until a single totally ordered run remains, whereafter no more comparisons between elements of X are required. To process a query implies a binary search through each of the existing runs. The number of queries processed between consecutive merging stages or, equivalently, the minimum length of a run before the ith query, are chosen appropriately.

This algorithm ensures that the size of each run is at least $L(i)$ before the ith query. A suitable choice for $L(i)$ is $\Theta(i \log i)$. Since the length of a run must be a power of 2 we will choose

$$L(i) = 2^{\lceil \log (i \log i) \rceil}.$$

The processing cost of going from a stage with runs of length 1 to runs of length $L(i)$ is $O(n \log L(i))$. Thus the total cost of processing in answering r queries is $O(n \log r)$. The search cost for the ith query is upper bounded by $n \cdot \lceil \log (L(i)+1) \rceil / L(i)$. Summing over the first r queries, the search cost is bounded by

$$\sum_{i=1}^{r} \frac{n}{L(i)} \cdot \lceil \log (L(i)+1) \rceil = O(n \log r).$$

THEOREM 2. *For $r \leqq n$, the total cost of answering r queries is $O(n \log r)$.*

When $r > n$, we note that the set X will be completely ordered by our strategy. All queries are then answered in time $O(\log n)$ by binary search.

Proof. The processing cost and the search cost are each $O(n \log r)$, so that the total cost of answering the first r queries is $O(n \log r)$. □

2.2.2. An approach based on recursive median finding. We now describe an alternative approach based on median finding; a specification of the algorithm in "pseudo-pascal" follows. The algorithm builds the binary search tree T_X in a query-driven fashion. Each internal node v of T_X is viewed as representing a subset $X(v)$ of X—the root represents X, its left and right children represent the smallest $(n-1)/2$ and the biggest $(n-1)/2$ elements of X, respectively, and so on. Let LSon (v) and RSon (v) represent the left and right children of v, respectively. We can now think of building T_X as follows. For each internal node v, *expansion* consists of partitioning $X(v)$ into two subsets of equal size—elements smaller than the median of $X(v)$, which will constitute $X(\text{Lson}(v))$, and elements larger than the median, which will make up $X(\text{Rson}(v))$. We label v by the median of $X(v)$. Thus a node at level i represents at most $n/2^i$ elements of X.[1] Subsequently, LSon (v) and RSon (v) may be expanded. Since the median of $X(v)$ can be found in $3|X(v)|$ comparisons [12], the expansion of node v takes $3|X(v)|$ comparisons. If we begin by expanding the root of T_X (which represents the entire set X), and then expand every node created, T_X can be built up in $3n \log n$ comparisons.

The search for a query can be thought of as tracing a root-to-leaf path in T_X. The key observation is that for any given query q_j, we need only expand those nodes visited by the search for q_j; this is the query-driven tree construction referred to earlier. After each expansion, at most one of the resulting offspring will be visited. The first query q_1 is answered in $O(n+n/2+\cdots)=O(n)$ operations while building up one root-to-leaf path of T_X. The time taken to answer q_1 is thus within a constant factor of the time for a linear search. In the process of answering q_1, we have developed some structure that will be useful in answering subsequent queries; any future search that visits a node that is already expanded will only cost us a single comparison to proceed to the next level of the search; there is no further expansion cost at this node. Nodes that remain unexpanded will be expanded when other queries visit them. When n queries that visit all n leaves have been answered, T_X will have been completely built up. In essence, we are dispensing with an explicit preprocessing phase, i.e., we are doing "preprocessing" operations only when needed. The cost of building the data structure is distributed over several queries.

DETAILED DESCRIPTION OF THE ALGORITHM. With every node in the tree we associate a set of values and a label, both of which may at times be undefined.

Main body

Step 1. Initialize the tree, T_X, with the n data keys at the root.
Step 2. Get a query q.
Step 3. Result $\leftarrow$ SEARCH (*root*, q).
Step 4. Output the result.
Step 5. Goto Step 2

procedure SEARCH (v: node; q:query): boolean;

Step 1. If (v is not labeled) then EXPAND (v).
Step 2. If ($label(v)=q$) then return *true.*
Step 3. If (v is a leaf node) then return *false.*

[1] Actually it represents slightly fewer elements, since each node picks up one element of X as its label. This does not matter, as we are deriving an upper bound.

Step 4. If ($q < label(v)$) then return SEARCH ($left\_child(v)$, q).
Step 5. If ($q > label(v)$) then return SEARCH ($right\_child(v)$, q).

procedure EXPAND (v: node);
Step 1. $S \leftarrow set(v)$.
Step 2. $m \leftarrow$ MEDIAN_FIND (S).
Step 3. $label(v) \leftarrow m$.
Step 4. if ($|S| = 1$) then return.
Step 5. $S_l \leftarrow [x \mid x \in S$ and $x < m]$.
Step 6. $S_r \leftarrow [x \mid x \in S$ and $x > m]$.
Step 7. $set(leftchild(v)) \leftarrow S_l$.
Step 8. $set(rightchild(v)) \leftarrow S_r$.

It should be noted that the two subsets, S_l and S_r, are computed by the procedure MEDIAN_FIND as part of the process of finding the median. There is no extra work associated with determining these two sets once the median has been found.

In order to analyze our algorithm, let us define a function on n and r as follows:

$$\Lambda(n, r) = \begin{cases} 3n \log r + r \log n, & r \leqq n, \\ (3n + r) \cdot \log n, & r > n. \end{cases}$$

Note that $\Lambda(n, r) = \Theta((n + r) \cdot \log \min(n, r))$ since $r \cdot \log n \leqq n \cdot \log r$ for $r \leqq n$.

THEOREM 3. *The number of operations needed for processing r queries is no more than* $\Lambda(n, r)$.

Proof. Consider the case $r \leqq n$. No more than r nodes will be expanded at any level of T_X, after r queries. For nodes in the top $\log r$ levels, the total cost is thus less than $3n \log r$. This is because *all* nodes may be expanded at each of the first log r levels. The expansion of a node v entails finding the median of $X(v)$ and this requires at least $3|X(v)|$ comparisons in the worst case [12]. For $i > \lceil \log r \rceil$ the node-expansion cost at level i is $O(r \cdot n/2^i)$. This is because the cost of expanding a node at level i is at most $3 \cdot n/2^i$. Summing over all but the first $\lceil \log r \rceil$ levels, the cost of node expansion at these levels is $O(n)$. In addition to the expansion cost, we have to consider the cost associated with search; this is at most $\log n$ comparisons per query. The search component of the cost is thus always less than $r \log n$.

When r exceeds n, the expansion cost can never exceed the cost of constructing T_X completely; this cost is $3n \log n$. Again, note that the factor of 3 comes from the median-finding procedure. □

2.3. A general paradigm for deferred data structuring. We are now ready to state the general paradigm for deferred data structuring. This paradigm will isolate some features essential for a search problem to be amenable to this approach, and will simplify our description of the geometric search problems considered in §§ 4 and 5. It also enables us to identify some problems where this approach is not likely to work.

Let Π be a search problem with the following properties. (1) The search is on a set S of n data points (in the above example, $S = X$). (2) A query q can be answered in $O(n)$ time. (3) In time $O(n)$, we can partition S into two equal-sized subsets S_1 and S_2 such that (i) the answer to query q on set S is equal to the answer to q on either S_1 or S_2; (ii) in the course of partitioning S we can compute a function on S, $f(S)$, such that there is a constant time procedure, TEST ($f(S)$, q), which will determine whether the answer to q on S is to be found in S_1 or S_2. (In the above example $f(S) = $ MEDIAN (S) and TEST is a simple comparison operation.)

Under these conditions, we can adopt the deferred data structuring approach that builds the search tree gradually. We illustrate this paradigm by several geometric examples in §§ 4 and 5.

3. A randomized algorithm. In the last section we saw a deterministic algorithm to answer r queries in $O((n+r)\cdot\log\min\{n, r\})$ time using deferred data structures. The upper bound of Theorem 3 exceeds the information-theoretic lower bound by a factor of 3 if we use the median algorithm given in [12]. Finding the median of n elements takes $3n$ comparisons and this is what leads to the gap between the upper and lower bounds. A careful implementation would reduce the constant factor to 2.5 by passing down certain partial orders generated in the median-finding algorithm from parent to children nodes. More easily implemented algorithms given in [3] would yield even higher constant factors. There is an algorithm due to Floyd [7] which computes the median in $3n/2$ expected time. Its use would reduce our constant to 3/2. Here we present a randomized algorithm in which the number of comparisons will be optimal (with high probability).

The randomized algorithm differs from the one in § 2 in just one respect. The median of the set of values stored at a node was used earlier to get a partition for the purposes of node expansion. Here we will use a *mediocre* element for the same purpose. The mediocre element will be chosen to be quite close to the median. More precisely, the rank of a mediocre element from a set of size t will lie in the range $t/2\pm t^{2/3}$. We will use randomized techniques to compute a mediocre element efficiently. First, a random subset of size $O(t^{5/6})$ is chosen from the t elements. The median of this random subset is a good candidate for being a mediocre element. It takes $t+O(t^{5/6})$ comparisons to pick a random sample and test its median element for "mediocrity" (see Step 5 below). This sampling is repeated until a mediocre element is found. The call to the procedure MEDIAN_FIND, in the algorithm outlined in § 2, should be replaced by a call to the procedure MEDIOCRE_FIND outlined below.

procedure MEDIOCRE_FIND (T: set of values): value;

Step 1 Let $t=|T|$.
Step 2 Pick a random sample S of size $2\cdot\lceil t^{5/6}\rceil+1$ from T.
Step 3 $m\leftarrow$ MEDIAN_FIND (S).
Step 4 Compute rank (m) by comparing with each element of $T-S$.
Step 5 If rank (m) is not in the range $(t/2)\pm t^{2/3}$ then goto Step 2.
Step 6 Return m.

Note that in Step 4 we need not compare m with elements of S since we assume that the procedure MEDIAN_FIND implicitly gives us the partition of S with respect to m. At the last few levels we will revert to deterministic median finding since the node sizes will be too small to justify randomization. A good choice is to use procedure MEDIOCRE_FIND for the first $\log n-5$ levels and procedure MEDIAN_FIND thereafter. The randomized algorithm leads to the following theorem.

THEOREM 4. *Let $T(n, r)$ be the total number of comparisons made by the randomized algorithm in answering r queries on n elements. Then the following holds with probability greater than* $1-\log r/\beta\cdot n$,

$$T(n, r)\leqq\begin{cases}(1+\alpha)(n\log r+r\log n), & r\leqq n,\\ (1+\alpha)(n+r)\log n, & r>n,\end{cases}$$

where β depends on the value of the constant α.

The remainder of this section is devoted to the proof of this theorem. The proof will be organized into five lemmas.

The use of mediocre elements (instead of the median) may result in uneven splits, causing an imbalance in the binary search tree being created. Nevertheless, the following lemma shows that the higher of the new binary search tree cannot be much worse than $\log n$. We also show that the number of elements associated with each node at level i is close to $n/2^i$. Let the size of a node in the search tree be the number of elements associated with that node.

LEMMA 1. Let s_i be the size of some node at level i. Then,

$$n_i\left(1-\frac{20}{n_i^{1/3}}\right) \leqq s_i \leqq n_i\left(1+\frac{20}{n_i^{1/3}}\right),$$

provided $n_i \geqq 22$, where $n_i = n/2^i$.

Proof. We will prove one side of the inequality by means of induction on the levels. The inequality is clearly true at the root ($i=0$) since $s_0 = n$. Suppose the inequality holds up to level $i-1$, i.e., $s_{i-1} \leqq n_{i-1}\cdot(1+20/n_{i-1}^{1/3})$. We now partition the s_{i-1} elements about the mediocre element. Let s_i denote the larger of the two partition sets. By the definition of the mediocre element we have $s_i \leqq s_{i-1}/2 + s_{i-1}^{2/3}$. Using the fact that $(1+x)^a \leqq 1 + a\cdot x$, $0 \leqq a \leqq 1$ we get,

$$s_i \leqq n_i\left(1+\frac{1}{n_i^{1/3}}\left(11\cdot 2^{2/3}+\frac{40}{3}\cdot\frac{2^{1/3}}{n_i^{1/3}}\right)\right).$$

For $n_i \geqq 22$, we note that,

$$\left(11\cdot 2^{2/3}+\frac{40}{3}\cdot\frac{2^{1/3}}{n_i^{1/3}}\right) \leqq 20.$$

This implies the desired result,

$$s_i \leqq n_i\left(1+\frac{20}{n_i^{1/3}}\right)$$

provided $n_i \geqq 22$. □

LEMMA 2. *The height of the binary search tree in the randomized algorithm will be* $\log n + O(1)$.

Proof. At level $k = \log n - 5$ we will no longer be using mediocre elements to expand a node. Instead, we use the median of the set of elements stored at a node to partition those elements. At this point Lemma 1 is still applicable since $n_k = 32 \geqq 22$ and we have,

$$s_k \leqq n_k\left(1+\frac{20}{n_k^{1/3}}\right) \leqq 2^8.$$

Thus, the total number of levels is no more than $k+8$. The height of the tree is bounded by $\log n + 3$. □

From Lemma 2 it follows that the cost of searching in the randomized binary search tree will be close to optimal. Let us now consider the cost of constructing the tree, in particular the total cost of node expansions. The following result shows that the median of the random sample is a mediocre element for the entire set with very high probability.

LEMMA 3. *Let $p(t)$ be the probability that a single iteration of the random sampling does not come up with a mediocre element for a set of size t. Then,*

$$p(t) \leqq 2\cdot t^{1/2}\cdot \exp(-4\cdot t^{1/6}) \leqq \frac{1}{4t}.$$

Proof. Let T be a set of size t to which a single iteration of the random sampling process has been applied. First, a random subset S of size $s(t)=2\cdot f(t)+1$ is chosen, where $f(t)=\lceil t^{5/6}\rceil$. The median of S is tested for being a mediocre element of T. In other words, the rank of the median of S should be in the range $t/2\pm t^{2/3}$ in T. Let $P(x_r)$ be the event that the element x_r (the element of rank r in T) is the median of S:

$$P(x_r)=\binom{r-1}{f(t)}\cdot\binom{t-r}{f(t)}\bigg/\binom{t}{s(t)},\qquad f(t)<r\leqq t-f(t).$$

Let $d(t)=t^{2/3}$. We will refer to $f(t)$ and $d(t)$ as f and d, respectively, to simplify the following description. Clearly,

$$p(t)=\sum_{r=f+1}^{t/2-d}P(x_r)+\sum_{r=t/2+d}^{t-f}P(x_r)=\frac{2s}{t-2f}\sum_{r=f+1}^{t/2-d}\binom{r-f}{r}\cdot\binom{r}{f}\cdot\binom{t-r}{f}\bigg/\binom{t}{2\cdot f}.$$

We make use of Stirling's formula:

$$n!=(2\pi n)^{1/2}\left(\frac{n}{e}\right)^n e^{k_n},\qquad \frac{1}{12n+1}<k_n<\frac{1}{12n}$$

to derive the following inequality upon considerable simplification:

$$p(t)<2\cdot f^{1/2}\cdot\exp\left(\frac{-4fd^2}{t^2}\right).$$

Given the choices for $f(t)$ and $d(t)$ the bound on $p(t)$ follows immediately. The second part of the inequality given below is also easy to verify:

$$p(t)<2\cdot t^{1/2}\cdot\exp(-4t^{1/6})<\frac{1}{4t}.$$ □

Consider now the overall cost of expanding the nodes in the randomized algorithm. First, there is the cost of finding the medians of the small random samples. Lemma 5 will show that the cost of finding the medians of the small random samples is small even when summed over the entire tree. More important is the cost of deciding whether the median for the sample is a mediocre element for the entire set. There is no cost associated with the actual partitioning since the testing for "mediocrity" implicitly determines the precise partition (see Step 5 of the procedure MEDIOCRE_FIND). Consider the ith level in the tree being constructed. Let $m=2^i$ denote the maximum number of nodes at this level. The sizes of the sets associated with the nodes at this level must lie in the range $(n_i/2)\pm 20\cdot n^{2/3}$, where $n_i=n/m$ is the average size of these sets. Supppose each application of the random sampling yielded a mediocre element. This would imply that the total cost of testing for mediocrity is n. However, there will be some bad instances where we do not generate a mediocre element. Let the number of such instances be s at the ith level. The next lemma shows that with high probability s is bounded by εm, where ε is an appropriately small constant. Let c_i denote the cost of testing for mediocrity at level i. When $s\leqq\varepsilon\cdot m$ we have

$$c_i\leqq n+n\cdot\varepsilon\left(1+\frac{20}{n_i^{1/3}}\right)=(1+\alpha)\cdot n.$$

Since $n_i>1$ at all levels it is clear the $\alpha\leqq 21\cdot\varepsilon$.

LEMMA 4. *Let C denote the sum of c_i over all but the last $O(1/\varepsilon)$ levels, $P(C\geqq(1+\alpha)\cdot n\cdot\log r)\leqq\log r/k^2\cdot n$.*

Proof. Let the random variable ζ_i denote the number of bad instances in $l=(1+\varepsilon)\cdot m$ iterations of the random sampling at level i. We already have bounds on

$p(t)$, the probability of a single iteration on a node of size t being bad. The l iterations at level i do not use equal-sized sets. Therefore let p denote the largest value taken by $p(t)$ at the nodes of that level. Let $E(\zeta)$ and $D(\zeta)$ denote the mean and deviation of some random variable ζ. The Chebyshev inequality states that $P(|\zeta - E(\zeta)| \geqq \lambda \cdot D(\zeta)) \leqq 1/\lambda^2$. Since $E(\zeta_i) = l \cdot p$ and $D(\zeta_i) = (l \cdot p \cdot (1-p))^{1/2}$ we have the following:

$$P(\zeta_i \geqq l - m) \leqq \frac{1 \cdot p \cdot (1-p)}{m \cdot \varepsilon^2} \quad \text{when } p \leqq \frac{\varepsilon}{2 \cdot (1+\varepsilon)}.$$

Using the bounds on $p(t)$ and the lower bound on the size of a node at level i we get, $P(s > \varepsilon m) = P(c_i \geqq (1+\alpha) \cdot n) \leqq k/\varepsilon^2 \cdot n$ for all but the last $O(\log 1/\varepsilon)$ levels, k is a small constant. Choosing $\beta = \varepsilon^2/k$ and summing the probability over the first $\log r$ levels yields the required bound. □

LEMMA 5. *When $r < n$, the total cost of finding the medians of the random samples is $O(n^{5/6} \cdot r^{1/6})$ with probability $1 - \log r/\beta \cdot n$.*

Proof. The cost of finding the median at a node of size t is $3t$. Let the sizes of the two children of this node be $k \cdot t$ and $(1-k) \cdot t$, where k lies between $\frac{1}{2}$ and 1. The cost of finding the medians for the children will be proportional to $C(k) \cdot t^{5/6}$, where $C(k) = (k^{5/6} + (1-k)^{5/6})$. Clearly, $C(k)$ is maximized at $k = \frac{1}{2}$. Define $C = C(\frac{1}{2}) = 2^{1/6}$. Thus, the cost of finding the medians at a single level increases by at most a factor of C in going from level i to $i+1$. We know that the cost of median finding at the first level is $3 \cdot n^{5/6}$. Hence, the total median-finding cost for the first $\log r$ levels is

$$3 \cdot n^{5/6} \cdot (1 + C + C^2 \cdots C^{\log r - 1}).$$

This sums to $O(n^{5/6} \cdot r^{1/6})$ since $C \leqq 2^{1/6}$. When $r > n$ the bound on the median-finding cost becomes $O(n)$. In our analysis so far we have ignored the repetitions in the median finding for a given node. This will be necessary since not every median of the random sample will be a mediocre element for the entire set. However, the analysis in Lemma 4 also applies to the median-finding cost since it just bounds the number of repetitions of the mediocre finding process at a level. □

Theorem 4 follows immediately from Lemmas 2, 4, and 5.

4. Planar convex hull and linear programming problems.

4.1. Point membership in a convex hull. In this section we consider the following problem. We are given a set $P = \{p_1, p_2, \cdots, p_n\}$ of n data points in the plane. Data point p_i is specified by its two coordinates $p_i = (p_{ix}, p_{iy})$. The convex hull of P will be denoted by CH (P). We are required to answer a series of queries: "Is the query point $q_j = (q_{jx}, q_{jy})$ included in CH (P)?"

We first present two solutions based on the preprocessing approach. Neither of these is optimal for all values of r. Let BCH (P) denote those points of P which lie on the boundary of CH (P). A single query can be answered in $O(n)$ time as follows. Compute the polar angles from q_j to all the data points. The query point is included in CH (P) if and only if the range of angles $\geqq 180°$. Alternatively, we can answer r queries by first constructing CH (P) in time $O(n \log h)$, where h is the number of points in BCH (P) [6], [11]. Now choose a point, O, in the interior of CH (P) and divide the plane into h *wedges* by means of h semi-infinite lines originating at O and going through each of the h vertices of CH (P). Each wedge contains exactly one edge from the boundary of CH (P). In any wedge, all points on the same side of this edge as O must lie inside CH (P). To answer a query we first determine the wedge in which it lies in $O(\log h)$ time by doing a binary search with respect to the angles subtended at O. We can now test the query point with respect to the edge of the CH (P) which lies in that particular wedge to decide the membership in CH (P). This requires a total of $O((n+r)\cdot\log h)$ operations to answer r queries.

Our approach to solving the point membership problem using deferred data structuring is based on the Kirkpatrick–Seidel top-down convex-hull algorithm [6]. The edges on the boundary of CH (P) consist of an *upper chain* and a *lower chain.* Each of these is a sequence of edges going from the leftmost to the rightmost point in P. Consider a vertical line which partitions P into two nonempty subsets. Such a line will intersect with exactly one edge of each chain; these edges will be referred to as the *upper tangent* and the *lower tangent* corresponding to the line. The tangents corresponding to a vertical line which partitions P into subsets of equal size (which we call the *median line*) are called the tangents of P. Kirkpatrick and Seidel show that a tangent can be computed in $O(|P|)$ operations.

We now describe our deferred data structure. In the following description we only refer to the upper chain and tangents; analogous reasoning applies to the lower chain and tangents. The data structure consists of a binary search tree T_P in which each internal node v represents a subset $P(v)$ of P (where $P(root)=P$). Associated with v is an x-interval $R_v=[x_L(v), x_R(v)]$; $P(v)$ consists of exactly those data points whose x-coordinates lie in R_v. We expand a node by computing the median line of $P(v)$. The members of $P(v)$ are partitioned into two subsets: points lying to the left of the median line and points lying to its right. These are associated with the two children of v. The tangent for $P(v)$ can now be computed in $O(|P(v)|)$ operations. It is possible that the tangents corresponding to the two vertical lines demarcating R_v may be adjacent in the chain. In fact, the two tangents may be the same. In these degenerate cases we do not need to compute the tangent of $P(v)$. Such degeneracies can be identified from the tangents corresponding to the vertical lines bounding R_v (these tangents will have been computed by ancestors of v). If at a node we find that both the upper and the lower tangent are degenerate, we will not expand the node; such a node is a leaf of T_P. Since at least one new tangent is discovered each time we expand a node, the number of internal nodes of T_P (and hence the number of leaves of T_P) will never exceed h.

The search for a query traverses a root-to-leaf path in the search tree. A node is expanded when it is first visited. At any node v the search progresses to its left or right child depending on the x-coordinate of the query point. In addition, we test whether the query point lies below the upper tangent (extended to infinity in both directions) of $P(v)$. If this test fails at any node along the search path we know that the query point lies outside CH (P). Similar tests apply to the lower chain/tangent.

Figure 1 shows an example in which two queries $q1$ and $q2$ have resulted in the expansion of the root and its two children. The query $q1$ lay to the left of the median line of P, and above the lower tangent of P (extended to the left by dotted lines). This caused LSon (*root*) to be expanded; at this point we find that $q1$ lies below the lower tangent of the left child and is thus outside CH (P). Note that the lower tangents of *root* and LSon (*root*) meet at a point of P; this means that we will never again compute a lower tangent in the right-subtree of LSon (*root*). Similarly, $q2$ expands the right child of the root node; it is found to lie between the upper and lower tangents of RSon (*root*), and is thus in CH (P).

THEOREM 5. *The number of operations for processing r hull-membership queries is* $O(\Lambda(n, r))$.

Proof. The depth of T_P never exceeds $\log n$. Moreover, a node at level i can be expanded in time $O(n/2^i)$. This fits our paradigm. An analysis similar to the proof of Theorem 2 establishes the result. □

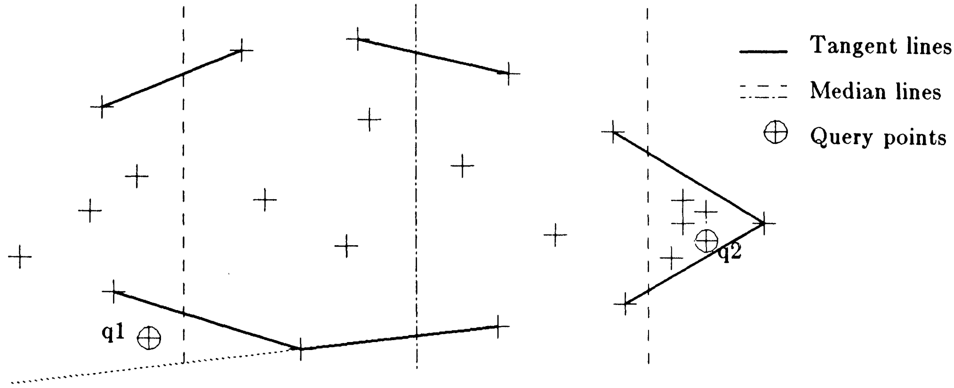

FIG. 1. *Membership in a hull; two queries and the resulting development of T_P.*

4.2. Intersection of half-spaces. We consider the problem of determining whether a query point $q_j = (q_{jx}, q_{jy})$ lies in the intersection of n half-planes. Let $H = \{h_1, h_2, \cdots h_n\}$ denote the set of lines which bound the half-planes. We assume that each half-plane contains the origin. If not, we can apply a suitable linear transformation in $O(n)$ time to bring the origin into the common intersection (provided the intersection of the h_i is nonempty). This can be done by finding a point in the interior of the intersection [8] and mapping the origin onto this feasible point. We can also test in linear time whether the intersection is empty [8]. Let H_i denote the half-plane (containing the origin) which is bounded by the line h_i. We assume in this section that the intersection of the H_i is bounded—in § 4.3 we will show that the case of an unbounded intersection region is easily handled.

The notion of geometric *duality* (or *polarity*) [4], [11] will prove extremely useful in the solution of the next two problems. In the plane this reduces to a transformation between points and lines. The dual of a point $p = (a, b)$ is the line l_p whose equation is $ax + by + 1 = 0$, and vice versa. A more intuitive definition is illustrated in Fig. 2.

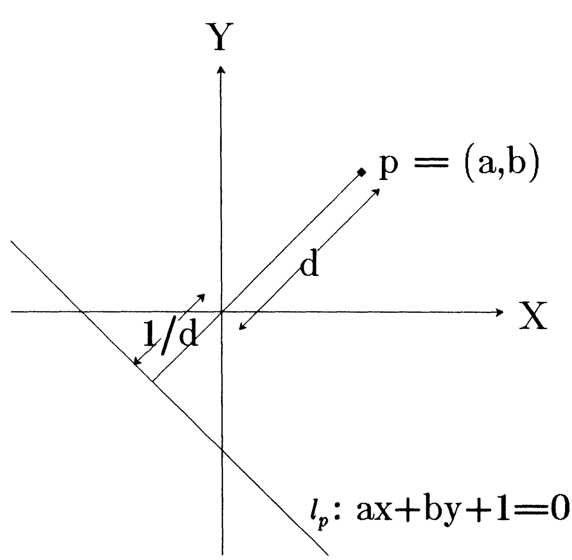

FIG. 2. *Duality of points and lines.*

The line l_p is perpendicular to the line joining the origin to the point p. If the distance between p and the origin is d, then the dual line l_p lies at a distance $1/d$ from the origin in the opposite direction.

We will now apply the duality transformation to the intersection of the half-planes under consideration. The dual of the line h_i is a point, which we will denote by p_i; we denote by P the set of these points. The dual of the intersection of the H_i is the set of all points in $\mathbf{R}^2$ not in CH (P). The dual of q_j is a line L_j. The query point q_j is in the intersection of the H_i if and only if L_j does not intersect CH (P). Thus our problem reduces to determining whether each of a series of query lines intersects the convex hull of a set of points.

The search tree and the node-expansion process are exactly the same as in § 4.1. At each node v, we compute the intersection of L_j with the median line of $P(v)$. We know that L_j must intersect CH (P) if one of the following holds: (1) the intersection point lies between the upper and lower tangents of $P(v)$; (2) L_j intersects one of the tangents of the current node. If not, we must continue the search in the left or right child of v, depending on the slopes of L_j and the tangent. These three possibilities are illustrated in Fig. 3 by lines $L1$, $L2$, and $L3$, respectively. In the case of $L3$, we see that any intersection of $L3$ with CH (P) must lie to the left of the median line; we therefore continue the search in LSon (v).

The following theorem results.

THEOREM 6. *The number of operations for processing r half-plane intersection queries is* $O(\Lambda(n, r))$.

4.3. Two-variable linear programming. Let $L(f)$ be a two-variable linear programming problem with n constraints and the objective function f, which is to be minimized subject to these constraints. The algorithms of Dyer [5] and Megiddo [8] can find the optimum for a single objective function in time $O(n)$. We consider a query version of the linear programming problem. Each query is an objective function f_i, and we are asked to solve $L(f_i)$.

The preprocessing approach to this problem consists of finding the intersection of the half-planes defined by the constraints. This can be done in $O(n \log n)$ time by divide-and-conquer. The set of half-planes is partitioned into two sets of almost equal

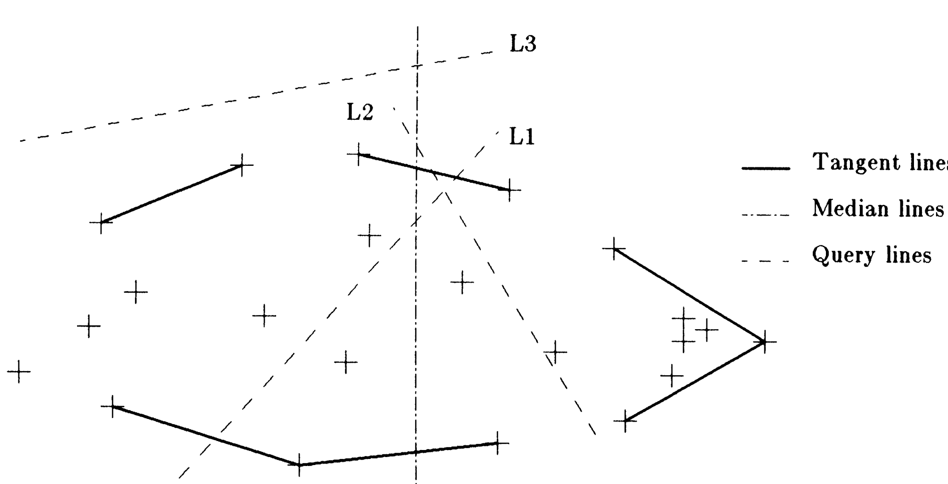

FIG. 3. *Example for testing line intersection with a hull.*

sizes. The intersection of half-planes in each subproblem can be found recursively; the two intersections can then be merged in linear time [11]. A binary search for the slope of the objective function then answers each query in $O(\log n)$ time.

As before, we resort to the geometric dual to solve the problem. We may again assume without loss of generality that the feasible region R_L is nonempty and contains the origin. Each of the n constraints defines a half-plane H_i; R_L is the intersection of these half-planes. Using the notation of § 4.1, the dual of R_L is the exterior of CH (P).

To begin with, we will assume that R_L is bounded. This implies that the origin in the dual plane lies in CH (P). The objective function f_i can be looked upon as a family of parallel lines in the primal. Depending on the slope of f_i, we need only consider the set of parallel lines above or below the origin. This set of lines dualizes to a semi-infinite straight line with the origin as one endpoint. We call this the objective line g_i, and note that it intersects the boundary of CH (P) at one point which corresponds to the optimum solution.

The search tree and node expansion are as in § 4.2. While searching at a node v, we compute the intersection, if any, of g_i with the median line of $P(v)$. If there is no intersection or if the point of intersection does not lie between the tangents, the search proceeds to the left (right) child of v if the origin lies to the left (right) of the median line. Otherwise, we proceed in the opposite direction. The search terminates if g_i intersects a tangent of $P(v)$.

When R_L is unbounded, the origin in the dual plane does not lie in CH (P). If g_i does not intersect CH (P), the solution to the problem is unbounded. This can be detected by computing in $O(n)$ time the polar angle from the origin to all points in P; this is done once, at the beginning. If g_i lies outside the cone defined by this range of angles, it does not intersect CH (P). If g_i intersects CH (P), we use the same search procedure as in the bounded case. The two points in BCH (P) which subtend the extreme angles at the origin are joined by a tangent. Intersection with this tangent is ignored for the termination criterion above.

Figure 4 shows an unbounded feasible region, and the corresponding convex hull in the dual. Two objective functions f_1 and f_2 and their dual objective lines are shown. The arc in the dual indicates the locus of objective lines (e.g., g_2) that do not intersect CH (P), and hence have unbounded optima.

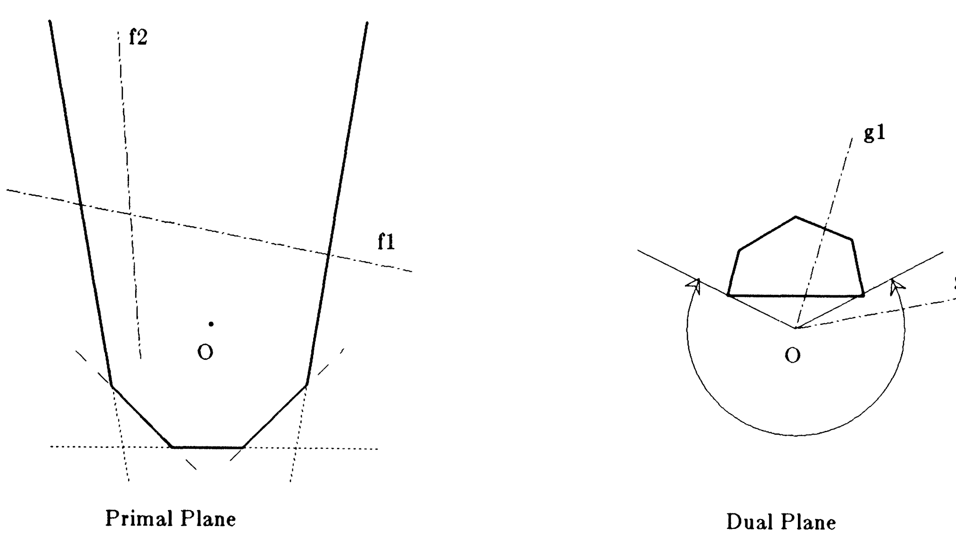

FIG. 4. *Unbounded linear-programming search example.*

THEOREM 7. *The number of operations for processing r two-variable linear programming queries is* $O(\Lambda(n, r))$.

4.4. Lower bounds under the algebraic tree model. The information-theoretic lower bound of § 2 is not valid for the geometric problems we have been considering in this section. In § 2 we were working with the comparison-tree model of computation, whereas we are allowing arithmetic operations here. We therefore use the *algebraic tree* model of computation [1].

An algebraic computation tree is an algorithm to decicide whether an input vector, a point in $\mathbf{R}^n$, lies in a point set $W \subseteq \mathbf{R}^n$. The nodes in the tree are of three types: *computation* nodes, *branching* nodes, and *leaves.* A computation node has exactly one child and it can perform one of the usual arithmetic operations or compute a square root. A branching node behaves like a node in a comparison tree, i.e., it can perform comparisons with previously computed values. It has exactly two children corresponding to the possible outcomes of the comparison. A leaf is labeled either "Accept" or "Reject," and it has no children. Each addition operation, subtraction operation or multiplication by a constant costs zero. Every other operation or comparison has a unit cost. The complexity of an algebraic computation tree is the maximum sum of costs along a root-leaf path in the tree. If $W \subseteq \mathbf{R}^n$, then $C(W)$, the complexity of W, is the minimum complexity of a tree that accepts precisely the set W. For any point set $S \subseteq \mathbf{R}^n$, let $\#(S)$ denote the number of connected components of W. It was shown in [1] that $C(W) = \Omega(\log \#(W))$.

We now show a lower bound of $\Omega((n+r) \cdot \log \min\{n, r\})$ algebraic operations for processing r hull-membership queries on n data points. We will in fact show that this bound holds when the r queries are processed *off-line.* The bound is obtained through a reduction from the SET DISJOINTNESS problem, defined as follows. Given two sets $X = \{x_1, x_2 \cdots x_n\}$ and $Q = \{q_1, q_2 \cdots q_r\}$, determine whether their intersection is nonempty. This problem is a simpler version of the SET INTERSECTION problem mentioned in § 2. We first prove a lower bound on SET DISJOINTNESS.

THEOREM 8. *Any algebraic computation tree that solves* SET DISJOINTNESS *must have a complexity of* $\Omega((n+r) \cdot \log \min\{n, r\})$.

Proof. Assume without loss of generality that $r \leq n$. Every instance of SET DISJOINTNESS can be represented as a point $\beta = (x_1, \cdots, x_n, q_1, \cdots, q_r)$ in $\mathbf{R}^{n+r}$. Let $W \subseteq \mathbf{R}^{n+r}$ be the set of all points representing disjoint sets. The complexity of the problem is $\Omega(\log \#(W))$, where $\#(W)$ is the number of connected components of W [1]. Consider instances for which the q_i are distinct. The elements of Q can be ordered as $\{q_{(1)} < q_{(2)} < \cdots < q_{(r)}\}$, where (i) represents the index of the ith smallest value in $\{q_1, \cdots, q_r\}$. Let $S_\beta(i) = \{x_k : q_{(i)} < x_k < q_{(i+1)}\}$, for $1 \leqq i \leqq r-1$. Define $W^* = \{\beta : |S_\beta(i) = \lfloor n/(r-1) \rfloor, 1 \leqq i \leqq r-1\}$, $W^* \subseteq W$. The subsets of W^* corresponding to different choices of S_β's are separated by hyperplanes of the form $x_i = q_j$. These hyperplanes are entirely disjoint from W. This means that if two points in W^* are separated by these hyperplanes then they must also be separated in W. Hence, the number of components of W is at least as large as the number of ways of partitioning $\{x_1, x_2, \cdots, x_n\}$ into the S_β's as per the definition of W^*. A counting argument shows that this is at least as large as

$$r! \frac{n!}{(\lfloor (n/r-1) \rfloor !)^{r-1}}.$$

From this it follows that the complexity is $\Omega((n+r) \cdot \log r)$. □

THEOREM 9. *The complexity of processing r hull-membership queries is* $\Omega((n+r) \cdot \log \min\{n, r\})$.

Proof. By reduction from SET DISJOINTNESS in $O(n+r)$ time. Without loss of generality, assume that the elements of both sets lie in the interval $[0, 2\pi)$. Each element x_i maps onto a point p_i on the unit circle with polar coordinates $(1, x_i)$. This constitutes our data set P; note that BCH $(P) = P$. Each element q_j of Q maps onto a point r_j with polar coordinates $(1, q_j)$. The point r_j lies in CH (P) if and only if $q_j \in X$. Thus SET DISJOINTNESS α_{n+r} HULL_MEMBERSHIP. □

The lower bound extends to the problems in §§ 4.2 and 4.3.

4.5. Effect of the number of points on the convex hull. In this section we return to the problem of determining whether a query point lies within the convex hull of n given data points. We show that a substantial improvement is possible when h, the number of data points on the boundary of the convex hull, is much smaller than n. It is clear that the guarantees of Theorem 4 are too weak in such a case, since it is possible to find CH (P) in $O(n \log h)$ operations by the Kirkpatrick-Seidel algorithm; subsequently, queries can be answered in time $O(\log h)$ each. This gives a time bound of $O((n+r) \log h)$ for answering r queries. This may seem to contradict the lower bound of Theorem 9 but recall that in the lower bound reduction all n data points were on the boundary of the convex hull. When r exceeds h, the algorithm of § 4.1 achieves a time bound of $O(n \log h + r \log n)$, since node expansion costs add up to only $O(n \log h)$. The cost of searching, however, unfortunately grows as $r \log n$ because the depth of T_P may grow as $\log n$ even though the number of leaves is only h.

To get around this difficulty we construct, in a dovetailed fashion, two binary search trees T_P and T_D. Let T be the fully expanded version of the search tree constructed by the algorithm of § 4.1. It has h leaves and can be constructed in $O(n \log h)$ time. The two trees T_P and T_D will be partially expanded versions of T. T_P is the version obtained by processing queries according to the algorithm of § 4.1. The other tree T_D is obtained by partially constructing T through a deferred depth-first traversal.

The depth-first traversal of a tree with l leaves can be looked upon as consisting of l phases, each of which ends when a new leaf is reached. Similarly, the depth-first construction of T_D can be broken down into h phases. These h phases are interleaves with the processing of the first h queries on the search tree T_P. Each phase can also be looked upon as the processing of a judiciously chosen query on the tree T_D. Thus the cost of the deferred construction of T_D has the same upper bound as that for T_P.

When r exceeds h, the tree T_D will be fully constructed after the first h queries have been processed on T_P. At this point T_P itself may not be fully expanded; in fact only one leaf may have been exposed in it. Since the CH (P) is now completely determined by T_D we can do away with the two search trees for further query processing. We now resort to the *wedge* method to answer each query in time $O(\log h)$ (see § 4.1). Since the cost of constructing T_D is $O(n \log h)$ the following theorem results.

THEOREM 10. *The cost of processing r hull-membership queries is* $O(\Lambda'(n, r, h))$, *where*

$$\Lambda'(n, r, h) = \begin{cases} n \log r, & r \leqq h, \\ (n+r) \cdot \log h, & r > h. \end{cases}$$

Analogous results hold for the problems in §§ 4.2 and 4.3.

5. Domination problems. In this section we investigate a problem related to *point domination* in k-dimensional space. This problem does not fit directly into the paradigm presented at the end of § 2. However, a higher-dimensional analogue of divide-and-conquer enables us to adapt our technique to such problems.

Let p_i denote the ith coordinate of a point p in k-space. We say that p *dominates* q if and only if $p_i \geqq q_i$ for all i, $1 \leqq i \leqq k$. Bentley [2] considers the *dominance counting* problem which is also called the ECDF *Searching Problem.* In this problem we are given a set $P = \{p_1, p_2 \cdots p_n\}$ of n points in k-space. For each query point q, we are asked to report the number of points of P dominated by q.

Bentley uses a *multidimensional divide-and-conquer* strategy to solve this problem. He constructs a data structure, the ECDF *tree*, which answers each query in $O(\log^k n)$ time following a preprocessing phase requiring $O(n \log^{k-1} n)$ time. This result holds for fixed number of dimensions (k) and for n a power of 2. However, a more detailed analysis due to Monier [9] shows the validity of this result for arbitrary n and k. In fact, Monier shows that the constant implicit in the O result is $1/(k-1)!$. In the following analysis we too will assume that the number of dimensions is fixed and that n is a power of 2. Our results can be generalized to allow for arbitrary k and n by invoking the results due to Monier.

The basic paradigm of multidimensional divide-and-conquer is as follows: given a problem involving n points in k-space, first divide into (and recursively solve) two subproblems each of $n/2$ points in k-space, and then recursively solve one problem of at most n points in $(k-1)$-space. When applied to the dominance counting problem, this paradigm yields the following search or counting strategy:

(1) Find a $(k-1)$-dimensional hyperplane M dividing P into two subsets P_1 and P_2, each of cardinality $n/2$. We will assume that M is of the form $x_k = c$. Hence, all points in P_1 have their kth coordinate less than c, while those in P_2 have their kth coordinate greater than c.

(2) If the query point q lies on the same side of M as P_1 (i.e., $q_k < c$) then recursively search in P_1 only. It is clear that the query point cannot dominate any point in P_2.

(3) Otherwise, q lies on the same side of M as P_2 (i.e., $q_k > c$) and we know that q dominates every point of P_1 in the kth-coordinate. Now we project P_1 and q onto M and recursively search in $(k-1)$-space. We also search P_2 in k-space.

In Fig. 5 we illustrate this strategy for two-dimensional space.

In one-dimensional space the ECDF searching problem reduces to finding the rank of a query value in the given data-set. The one-dimensional ECDF search tree is an optimal binary search tree on the n points in P. The k-dimensional ECDF tree for

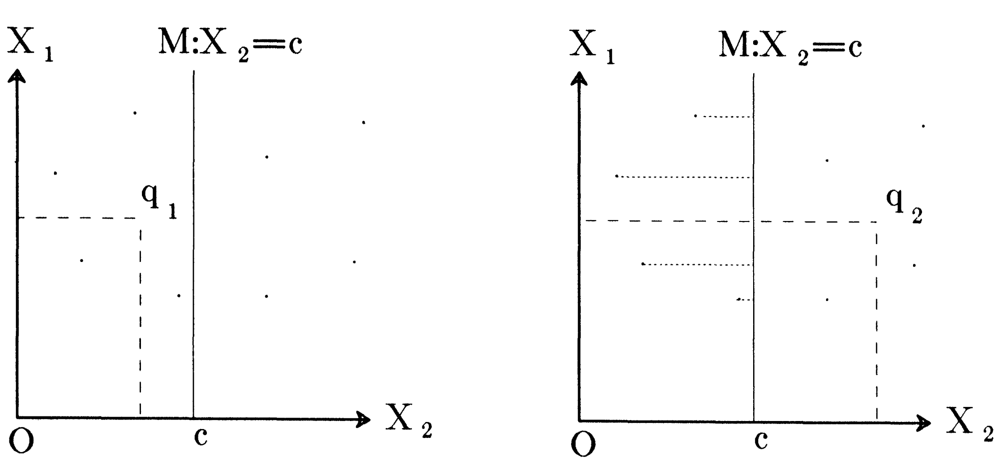

FIG. 5. *The two cases for dominance counting in 2-space.*

the n points in P is a recursively built data structure. The root of this tree contains M, the median hyperplane for the kth dimension. The left subtree is a k-dimensional ECDF tree for the $n/2$ points in P_1, the points in P which lie below M. Similarly, the right subtree is a k-dimensional ECDF tree for the $n/2$ points in P_2, the points in P which lie above M. The root also contains a $(k-1)$-dimensional ECDF tree representing the points in the P_1 projected onto M.

To answer a query q, the search algorithm compares q_k to c, the value defining the median plane M stored at the root. If q_k is less than c then the search is restricted to the points in P_1 only. The algorithm then recursively searches in the left subtree. If, on the other hand, q_k is greater than c then the algorithm recursively searches in the right subtree as well as the $(k-1)$-dimensional ECDF tree stored at the root. For the one-dimensional ECDF tree the algorithm is the standard binary tree search. For fixed k, the preprocessing time to build the k-dimensional ECDF tree is $p(n) = O(n \log^k n)$, and the time required to answer a single query is $q(n) = O(\log^k n)$.

We now apply the deferred data structuring technique to the k-dimensional ECDF tree. As before, we do not perform any preprocessing to construct the search tree. The ECDF tree is constructed on-the-fly in the process of answering the queries. Initially, all the points are stored at the root of the k-dimensional ECDF tree. In general, when a query search reaches an unexpanded node v we compute the median hyperplane, M_v, and partition the data points around M_v. The two sets are then passed down to the two descendant nodes of v. We also initialize the $(k-1)$-dimensional ECDF tree which is to be created at v. Even these lower-dimensional trees are created in a deferred fashion depending upon the queries being answered. The application of deferred data structuring to the ECDF tree results in the following theorem.

THEOREM 11. *The cost of answering r dominance search queries in k-space is $O(F(n, r, k))$, where*

$$F(n, r, k) = \begin{cases} n \log^k r + r \log^k n, & r \leqq n, \\ n \log^k n + r \log^k n, & r > n. \end{cases}$$

Proof. The proof will be by induction over both k and n. It is easy to see that the time required to answer a query remains unchanged by the process of deferring the construction of the ECDF tree. This proof will concentrate on the node-expansion component of the processing cost. Clearly, we need not consider the case where $r > n$ since the node-expansion cost cannot exceed the total preprocessing cost of the nondeferred ECDF tree. Let $f(n, r, k)$ denote the *worst-case* node-expansion cost for answering r queries over n data points in k dimensions using a k-dimensional ECDF tree. When r exceeds n we have $f(n, r, k) = O(n \cdot \log^k n)$ since n queries, each leading to a different leaf, are sufficient to fully expand the ECDF tree. We will now prove that $f(n, r, k) = O(n \cdot \log^k r)$ when $r \leqq n$.

The basis of this induction is the case where $k = 1$. Consider the one-dimensional ECDF tree. It is an optimal binary search tree and we can invoke Theorem 3 to show the validity of this theorem. This establishes the base case of our induction over k, in other words, $f(n, r, 1) = O(n \cdot \log r)$ when $r \leqq n$. The induction hypothesis is that the above result is valid for up to $k-1$ dimensions, i.e., $f(n, r, k-1) = O(n \cdot \log^{k-1} r)$ when $r \leqq n$. We now prove that it must be valid for k dimensions also. At the second level of our nested induction we concentrate on the k-dimensional ECDF tree and use induction over n. It is clear that the k-dimensional ECDF tree for $n = 1$ points will satisfy the above theorem for $r \leqq n$. We now assume that the result is valid for up to $n-1$ points in k dimensions. To complete the proof we show that, under the given assumptions, the result can be extended to n points in k dimensions.

Consider the root node, say V, of the k-dimensional ECDF tree for the n points in P. It contains a median hyperplane, say M_V, which partitions the n points in P into two equal subsets, P_1 and P_2. Recall that P_1 is the set of all those points in P which lie below M_V; P_2 is the set of those points in P which lie above M_V. The left and right subtrees of V are the k-dimensional ECDF trees for P_1 and P_2, respectively. We also store at V a $(k-1)$-dimensional ECDF tree, say T_1, for the projections of the points in P_1 onto M_V. This lower-dimension tree creates a kind of asymmetry between P_1 and P_2. This asymmetry can complicate our proof considerably. Therefore, for the purposes of this proof only, we will make a simplifying assumption about the structure of the ECDF tree. We assume that V also contains a $(k-1)$-dimensional ECDF tree, say T_2, for the projections of the points in P_2 onto M_V.

The search procedure for the ECDF tree is also modified to introduce symmetry. Given a query q, we first test it with respect to the median hyperplane M_V. If it lies above M_V the search continues in the right subtree of V and in T_1. On the other hand, if q lies below M_V we continue the search in the left subtree of V as well as T_2. The search in T_2 is redundant because q, lying below M_V, cannot dominate any point in P_2. These modifications are made not just at the root but at all nodes in an ECDF tree. It is not very hard to see that these modifications can only increase the running times of our node-expansion algorithm. Moreover, these changes entail performing redundant operations which do not change the outcome of our algorithm. It is clear, therefore, that any upper bounds on the node-expansion costs for the modified ECDF tree also apply to the original deferred data structure.

We now proceed to complete the induction proof for $r \leqq n$. Let r_1 denote the number of queries which lie below the median hyperplane M_V. These queries continue the search down the left subtree of the root. Let $r_2 = r - r_1$ denote the remaining queries which continue the search down the right subtree as they lie above the median hyperplane M_V. Consider the node-expansion costs involved in processing these queries. Finding the median hyperplane M_V requires $O(n)$ operations. The r_1 queries lying below M_V are processed in the left subtree of V (a k-dimensional ECDF tree on $n/2$ points) and in T_2 (a $(k-1)$-dimensional ECDF tree on $n/2$ points). The remaining r_2 queries are processed in the right subtree of V (a k-dimensional ECDF tree on $n/2$ points) and in T_1 (a $(k-1)$-dimensional ECDF tree on $n/2$ points). This gives us the following bound on the total node-expansion cost entailed by processing r queries:

$$f(n, r, k) = \max_{r_1+r_2=r} \left\{ f\left(\frac{n}{2}, r_1, k\right) + f\left(\frac{n}{2}, r_2, k\right) + f\left(\frac{n}{2}, r_1, k-1\right) \right.$$
$$\left. + f\left(\frac{n}{2}, r_2, k-1\right) + O(n) \right\}.$$

Using the induction hypotheses we know the exact form of the functions on the right-hand side of the inequality. In particular, we know that these functions are convex. This implies that the right-hand side of the inequality is maximized when $r_1 = r_2 = r/2$. Putting together all this we have the desired result

$$f(n, r, k) = O(n \cdot \log^k r), \qquad r \leqq n.$$

Again, note that this result is valid only for fixed k and n a power of 2. The constant implicit in the O will, in general, depend on k. Monier's detailed analyses [9] of Bentley's algorithm also extends our result to arbitrary n and k. □

Bentley [2] actually has a slightly better bound on the preprocessing time for constructing ECDF trees. He makes use of a *presorting* technique to improve the bound to $O(n \cdot \log^{k-1} n)$ for k-dimensional ECDF trees on n points. He first sorts all n points by the first coordinate in $O(n \cdot \log n)$ time. This ordering is maintained at every step, especially when dividing the points into two sets about a median hyperplane for some other coordinate. Consider the two-dimensional ECDF tree. Initially, all n points are stored at the root in *order by the first coordinate.* After the first query, these n points are partitioned about a median hyperplane and passed down to the children nodes. The ordering by the first coordinate is maintained during this partition. Let P_1 denote the points being passed down to the left subtree, P_2 denotes the points passed down to the right subtree. In the original ECDF tree we would have constructed a one-dimensional ECDF tree for the points in P_1 and stored it at the root. Instead, we now just store the points of P_1, in order by the first coordinate, at the root. This process is repeated at every node in the two-dimensional ECDF tree. We now use the two-dimensional ECDF tree as the basic data structure in our recursive construction of a k-dimensional ECDF tree. In effect, we have done away with the one-dimensional ECDF tree. The preprocessing cost for constructing the presorted k-dimensional ECDF tree becomes $O(n \cdot \log^{k-1} n) + O(n \cdot \log n)$. The new data structure is as easily deferred as the previous one and we have the following result.

THEOREM 12. *The cost of answering r dominance search queries in k-space is* $O(G(n, r, k))$, *where*

$$G(n, r, k) = \begin{cases} n \log n + n \log^{k-1} r + r \log^k n, & r \leqq n, \\ n \log^{k-1} n + r \log^k n, & r > n. \end{cases}$$

Proof. The proof follows from a straightforward modification of the proof for Theorem 11. Note that cost of presorting is subsumed by the node-expansion cost when $r > n$. □

6. Conclusion. The paradigm of deferred data structuring has been applied to some search problems. In all cases, we considered on-line queries and developed the search tree as queries were processed. For the problems studied, our method improves on existing strategies involving a preprocessing phase followed by a search phase. An interesting open problem is to design deferred data structures for dynamic data sets in which insertions and deletions are allowed concurrently with query processing.

The *nearest-neighbor problem* [13] asks for the nearest of n data points to a query point. The problem is solved using *Voronoi diagrams* in $O(\log n)$ search time; the Voronoi diagram can be constructed in $O(n \log n)$ time. There is no known top-down divide-and-conquer algorithm for constructing the Voronoi diagram optimally. The obvious top-down method of constructing the bisector of the left and the right $n/2$ points (see [14] for a definition of the bisector of two sets of points) fails, since sorting reduces to computing this bisector. It remains an interesting open problem whether a deferred data structure can be devised for the nearest-neighbor search problem. Note that the techniques of § 2 can be used to solve the one-dimensional nearest-neighbor problem.

REFERENCES

[1] M. BEN-OR, *Lower bounds for algebraic computation trees*, in Proc. 15th Annual ACM Symposium on Theory of Computing, May 1983, pp. 80–86.

[2] J. L. BENTLEY, *Multidimensional divide and conquer*, Comm. ACM, 23 (1980), pp. 214–229.

[3] M. BLUM, R. FLOYD, V. PRATT, R. RIVEST, AND R. TARJAN, *Time bounds for selection*, J. Comput. System. Sci., 7 (1973), pp. 448–461.

[4] B. M. CHAZELLE, L. J. GUIBAS, AND D. T. LEE, *The power of geometric duality*, in Proc. 24th Annual IEEE Annual Symposium on Foundations of Computer Science, November 1983, pp. 217–225.

[5] M. E. DYER, *Linear time algorithms for two- and three-variable linear programs*, SIAM J. Comput., 13 (1984), pp. 31–45.

[6] D. G. KIRKPATRICK AND R. SEIDEL, *The ultimate planar convex hull algorithm?*, SIAM J. Comput., 15 (1986), pp. 287–299.

[7] D. E. KNUTH, *The Art of Computer Programming: Sorting and Searching*, 3, Addison-Wesley, New York, 1973, pp. 217–219.

[8] N. MEGIDDO, *Linear time algorithm for linear programming in* R^3 *and related problems*, SIAM J. Comput., 12 (1983), pp. 759–776.

[9] L. MONIER, *Combinatorial solutions of multidimensional divide-and-conquer recurrences*, J. Algorithms, 1 (1986), pp. 60–74.

[10] R. MOTWANI AND P. RAGHAVAN, *Deferred data structures: query-driven preprocessing for geometric search problems*, in Proc. 2nd Annual ACM Symposium on Computational Geometry, Yorktown Heights, NY, June 1986, pp. 303–312.

[11] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin, New York, 1985.

[12] A. SCHÖNHAGE, M. PATERSON, AND N. PIPPENGER, *Finding the median*, J. Comput. System Sci., 13 (1981), pp. 184–199.

[13] M. I. SHAMOS AND D. HOEY, *Closest-point problems*, in Proc. 16th Annual IEEE Annual Symposium on Foundations of Computer Science, October 1975, pp. 151–162.

[14] M. I. SHAMOS, *Computational geometry*, Ph.D. thesis, Yale University, New Haven, CT, 1977.