BENTLEY, J. L., AND SHAW, M. (1980), An Alphard specification of a correct and efficient transformation on data structures, *in IEEE Trans. Software Engrg.*, in press. (Preliminary version in "Proceedings, Specifications of Reliable Software Conference, April 1979, IEEE," pp. 222–237.

DOBKIN, D., AND LIPTON, R. J. (1976), Multidimensional searching problems, *SIAM J. Computing* 5, No. 2 (June), 181–186.

EDELSBRUNNER, H. (1979), "Optimizing the Dynamization of Decomposable Searching Problems," Report 35, Institut fuer Informationsverarbeitung, Technische Universitat Graz.

KNUTH, D. E. (1968). "The Art of Computer Programming," Vol. 1, "Fundamental Algorithms," Addison–Wesley, Reading, Mass.

KNUTH, D. E. (1973). "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison–Wesley, Reading, Mass.

VAN LEEUWEN, J., AND MAURER, H. A. (1980), "Dynamic Systems of Static Data-Structures," Report 42, Institut fuer Informationsverarbeitung, Technische Universitat Graz.

VAN LEEUWEN, J., AND WOOD, D. (1979), "Dynamization of Decomposable Searching Problems," University of Utrecht Vakgroep Informatica Report RUU-CS-79-6.

LIPTON, R. J., AND TARJAN, R. E. (1977), Applications of a planar separator theorem, *in* "Proceedings, Eighteenth Symposium on the Foundation of Computer Science, October 1977, IEEE," pp. 162–170.

LUEKER, G. (1978), A data structure for orthogonal range queries, *in* "Proceedings, Nineteenth Symposium on the Foundations of Computer Science, October 1978, IEEE," pp. 28–34.

LUEKER, G. (1979). "A Transformation for Adding Range Restriction Capability to Dynamic Data Structures for Decomposable Searching Problems," UCI Technical Report 129, February 1979.

MAURER, H. A., and OTTMANN, T. (1979), "Dynamic Solutions of Decomposable Searching Problems," Report 33, Institut fuer Informationsverarbeitung, Technische Universitat Graz, June 1979.

MUNRO, J. I., and SUWANDA, H. (1979), Implicit data structures, *in* "Proceedings, Eleventh Symposium on the Theory of Computing, April 1979, ACM," pp. 108–117.

OVERMARS, M. H., and VAN LEEUWEN, J. (1979), "Two general methods for dynamizing decomposable searching problems," University of Utrecht Vakgroep Informatica Report RUU-CS-79-9a.

PREPARATA, F. P. (1978), "A New Approach to Planar Point Location," University of Illinois Coordinated Science Laboratory Report R-829, September 1978.

REINGOLD, E. M., AND TARJAN, R. E. (1978), "On a Greedy Heuristic for Complete Matching," Technical Report, University of Illinois, September 1978.

RIVEST, R. L. (1976), Partial match retrieval algorithms, *SIAM J. Computing* 5, No. 1 (March), 19–50.

SHAMOS, M. I. (1978), "Computational Geometry," Ph.D. thesis, Yale University.

VUILLEMIN, J. (1978). A data structure for manipulating priority queues, *Comm. ACM* 21, No. 4 (April), 309–315.

WILLARD, D. (1978). "Predicate-Oriented Database Search Algorithms," Harvard Aiken Computation Laboratory Report TR-20-78.

c. For each point j in $R[x]$, calculate j's nearest neighbor in S (say, k), set $T[j] = k$, add j to $R[k]$, and modify Q to reflect the new value of $T[j]$. Do the same for each point in $R[y]$.

The running time of the above algorithm is $\theta(N^{3/2} \lg N)$. By applying Theorem 6.3 to the Lipton–Tarjan structure, the set S can be built in $\theta(N \lg N)$ time and both searches and deletions require $\theta(N^{1/2} \lg N)$ time. Steps 1a and 1b therefore require $\theta(N \lg N)$ time, and Step 1c requires linear time. Each execution of Step 2a requires $\theta(\lg N)$ operations, Step 2b requires $\theta(N^{1/2} \lg N)$, and Step 2c requires at most 10 nearest neighbor searches, for a total of $\theta(N^{1/2} \lg N)$ time per iteration. The total running time of Step 2 is therefore $\theta(N^{3/2} \lg N)$, and this establishes the total time required by the algorithm.

Acknowledgments

The helpful comments of Donna Brown, Kevin Brown, Michael Shamos, Herb Wilf, and Andrew and Frances Yao are gratefully acknowledged

Note added in proof. Three important developments occurred while this paper was in press. Professor K. Mehlhorn, in a paper entitled "Lower Bounds on the Efficiency of Static to Dynamic Data Structures," showed two new lower bound results. First, he showed the optimality of the k-binomial transforms, without the restriction to arboreal strategies—although his model is very general, his lower bounds are weaker than ours by a constant factor. Mehlhorn also showed the optimality of the dual 2-binomial transform, with the restriction to arboreal strategies. Finally, the present authors have shown that the dual k-binomial transforms are not optimal for $k > 2$ by demonstrating superior transforms.

References

Since this paper was originally circulated as a technical report, a number of papers have appeared that discuss additional aspects of static-to-dynamic transformations. Although they are not cited in the text, for completeness we have included references to many of those papers in the bibliography.

Bentley, J. L. (1976), "Divide and Conquer Algorithms for Closest-Point Problems in Multidimensional Space," Ph.D. thesis, University of North Carolina, December 1976.

Bentley, J. L. (1979). Decomposable searching problems, *Inform. Process. Lett.* **8**, No. 5 (June) 244–251.

Bentley, J. L., Detig, D., Guibas, L., and Saxe, J. B. (1978). "An Optimal Data Structure for Minimal-Storage Dynamic Member Searching," Carnegie–Mellon University, 1978.

Bentley, J. L., and Maurer, H. A. (1980), Efficient worst-case data structures for range searching, *Acta Informatica* **13**, No. 2, 155–168.

Bentley, J. L., and Shamos, M. I. (1977), A problem in multivariate statistics: Algorithm, data structure, and applications, *in* "Proceedings, Fifteenth Allerton Conference on Communication, Control and Computing, September 1977," pp. 193–201.

We must now cite the following fact, which will be crucial later in our discussion.

> In a set of points in the plane, any given point can be the nearest neighbor of at most six other points in the set.

This fact is a consequence of the fact that at most six unit circles can be made to touch a given unit circle without overlap; a precise proof can be found in Bentley (1976). This fact has two pleasant implications for our algorithm. First, the cardinality of each list $R[i]$ can be at most 6. Second, the number of points whose nearest neighbors must be found in any iteration is bounded above by 10. (Because x was the nearest neighbor of at most five points besides the one just deleted from S—that is, y; the same holds for y.)

We will now describe precisely the algorithm we informally sketched above. It employs the following data structures.

—P, the input array of *points* to be matched.

—Q, a priority *queue* representing the objects in T, ordered by distance to nearest neighbor, and implemented as a heap.

—R, an array that is the *reverse* of structure T. The element $R[i]$ contains the list of (at most six) points whose nearest neighbor is point I.

—S, the *set* of currently unmatched points. The set is implemented by transforming the Lipton–Tarjan nearest neighbor structure into a dynamic structure supporting deletions and queries by the transform of Theorem 6.3.

—T, an array *telling* the nearest unmatched neighbor of each unmatched point.

Our implementation of Reingold and Tarjan's approximate matching method can now be described precisely as follows.

1. Initialize the structures as follows.
   a. Build S from the points contained in P.
   b. Build T by searching S a total of N times to find the nearest neighbor of each point. As each entry is made in T, record the corresponding "reverse" entry in R.
   c. Insert the elements of T into the priority queue Q.
2. Repeat the following operations $N/2$ times.
   a. Use Q to find a pair of points realizing the minimum interpoint distance among points in S; call the two points x and y. Report x and y as a pair in the matching.
   b. Delete x and y from S.

bility of the transforms: the identification of any searching problem P such that (1) P may be made decomposable by having the query provide some extra information, and (2) known static algorithms for P can be altered to yield that extra information at low cost. The identification of other such "pseudodecomposable" problems (and other decomposable problems in general) remains an open problem.

APPENDIX II: AN ALGORITHM FOR APPROXIMATE MATCHINGS

In this appendix we will investigate a computational problem that was examined by Reingold and Tarjan (1978). They studied the following method for finding low-cost matchings among a set of N points in the plane.

> Select a pair of points in the set that realize the minimal interpoint distance, report the points as being a pair in the matching, and remove them from the set. Repeat the above process $N/2$ times, at which time all points in the set have been matched.

Reingold and Tarjan described algorithms by which the above method could be implemented in $\theta(N^2 \lg N)$ worst-case time or $\theta(N^2)$ expected time. We will now investigate an algorithm based on the transform of Theorem 6.3 (structures with deletion only) that operates in $\theta(N^{3/2} \lg N)$ worst-case time.

Our first description of the algorithm will be fairly informal. The algorithm's primary data structures are the set, S, of all unmatched points (organized to facilitate nearest neighbor searching), and an array, T, recording for each unmatched point the unmatched point nearest it. The algorithm initializes the structure by building S as the set of all unmatched points and performing N nearest neighbor searches to compute T. We will now describe the iterative step. We find the closest pair (say, points x and y) by choosing x as an unmatched point with minimum distance to its nearest neighbor, and y as x's nearest neighbor (this can be accomplished in logarithmic time by a priority queue, Q, representing the distances in T). We then delete both x and y from S. To maintain T we must see if any other points in S have x or y as their nearest neighbor. To do this we keep a "reverse set", R, as an array of lists in which $R[i]$ records for each point i the set of all points that have i as their nearest neighbor (note that R is the inverse of T). We can now find all points with x or y as nearest neighbors by examining $R[x]$ and $R[y]$. We then use S to find the nearest neighbors of those points, record those nearest neighbors in T and R, and modify Q to reflect the new state of T. This completes the iterative step of the algorithm.

of all records in that subset (thus asking for the intersection of the query space and the record set) [$\cup$]. Finally, Best Match queries specify an "ideal" record and a distance function (often the Hamming distance), and ask for the record in the set closest to the ideal [min]. These queries and data structures for answering them are discussed by Rivest (1976).

We saw in the body of the paper two decomposable searching problems that arise in statistics. Both of the problems are defined in terms of vector domination (one vector is said to dominate another if it is greater in all coordinates). A Maxima query asks whether the query vector is dominated by any in the set [$\vee$]. The Empirical Cumulative Distribution Function (ECDF) query asks how many vectors a given vector dominates [+].

Examples of decomposable searching problems abound in computational geometry. Many queries are asked of sets of points in the plane or Euclidean k-space, including Nearest Neighbor (which point in the set is nearest the query point?) [min], Furthest Neighbor [max], and Near Neighbor (list all points within distance d of the query point) [$\cup$] queries. Other queries deal with more complicated objects. For example, we might wish to know whether a given point is in the intersection of a set of half-planes (this problem arises in linear programming)—Feasible Region queries are decomposable [with the $\wedge$ operator]. Other queries include Rectangle Intersection (what rectangles in the set does this rectangle intersect?) [$\cup$] and Circle Intersection [$\cup$]. The queries and many others have been discussed in detail by Shamos (1978). Dobkin and Lipton (1976) investigate a number of decomposable searching problems in multidimensional space; these include such queries as "is the point on any of the lines" [$\vee$] and "is this point on any of the hyperplanes" [$\vee$]. Many of the other problems that we have already mentioned can be cast in geometric terms; these include ECDF, Maxima and Range searching.

Convex Hull searching is a very interesting problem from the viewpoint of decomposability. In its simplest form—"is point x within the convex hull of point set F?"—it is simple to prove that it is not decomposable, since whenever F contains at least two points we can partition F and specify x so that x is not in the hull of either part but either is or is not in the hull of the union. If we ask instead the query "what does the hull of the set look like from here?" (the answer being either an assertion that the query point is within the hull or a pair of angles giving the extremal points of the hull as "viewed" from the query point), the problem is now decomposable. The transforms described in this paper are therefore applicable to any data structure for Convex Hull searching, provided that structure can be cheaply modified to answer the more complicated "view" query. While this result is not of particular interest in itself (since one can develop fast *ad hoc* algorithms for dynamic Convex Hull searching), it indicates a possibly fruitful technique for extending the domain of applica-

"online" applications. Our study of dynamic structures up to this point concentrated on structures that supported only insertions and queries; in Section 6 we investigated structures that also support deletions. We saw that although it is impossible to achieve efficient deletions in the general case, they can be achieved for an important subclass of the decomposable searching problems.

The contributions of this paper can be classified on three distinct levels. On the first level are the new data structures that we have seen. Each one is currently the best-known structure for its task (with the exception of New Data Structure 6) and *each was discovered by conscious application of the transforms described in this paper*. On a second level are the transformations themselves; they are very interesting from a combinatorial viewpoint, and provide a useful addition to the algorithm designer's tool bag. On the third and final level is the new kind of result represented by the transformations: they are not just a single solution to a single problem, but rather a set of solutions to a broad class of problems. This aspect of the work will be further emphasized in later parts of this paper.

APPENDIX I: A LIST OF DECOMPOSABLE SEARCHING PROBLEMS

Throughout the body of this paper we have examined a number of operations on decomposable searching problems. In this appendix we will list some (23) searching problems that have the property of decomposability. For each problem we will note its $\square$ operator in square brackets.

The most common kind of searching problems are those defined on totally ordered sets. We already saw that Member searching (which asks "is x an element of F?") is decomposable [with $\square$ operator $\vee$]. Other examples are Successor (what is the least element in F greater than x?) [min], Predecessor [max], Rank (how many elements in F are less than x?) [+], and Count (how many elements in multiset F have value x?) [+]. Two queries on ordered sets that have no query element are the priority queue operations Min [min] and Max [max]. These problems, applications in which they arise, and data structures for their solutions are discussed in depth by Knuth (1973).

Many of the problems that arise in database applications are decomposable. In this context, the set of elements is usually a file of records, each of which contains certain keys. An Exact Match query calls for a list of all records that have all keys equal to specified values [$\cup$]. A Partial Match query asks for all records that match some subset of the keys [$\cup$]. Range queries ask for all records that have each key in a specified range of values [$\cup$]. Intersection queries specify a subset of the key space and ask for a list

S into M subsets, each of M elements, and then builds M static structures, named $P_1, \ldots, P_M$, with one static structure to represent each subset. To Delete a given element we locate its structure P_i, and rebuild P_i without the given element; note that rebuilding requires at most $P(M)$ time. (The given element can be located by storing a table, L, such that L_j gives the integer i of the structure P_i containing element j; this table requires linear time to build and constant time for a lookup.) The Query operation is accomplished by querying all M static structures and combining the answers by $M - 1$ applications of the $\square$ operator; this requires at most $M \cdot Q(M)$ time. These facts together establish the following theorem.

THEOREM 6.3 (structures with deletion only). *Given a static structure S for a decomposable searching problem there exists a dynamic structure* DO *with operations Build, Delete, and Query with performances*

$$S_{DO}(N) \leq S_S(N),$$
$$B_{DO}(N) \leq P_S(N),$$
$$D_{DO}(N) \leq P_S(N^{1/2}),$$
$$Q_{DO}(N) \leq N^{1/2} Q_S(N^{1/2}).$$

The function $B(N)$ denotes the time required to Build a structure of N elements. We assume that the functions S_S and P_S grow at least linearly and that Q_S is monotone increasing.

An adversary argument similar to that used in the proof of Theorem 6.1 can be used to show that the above construction is nearly optimal. Specifically, it can be shown that building a structure of N elements and then performing a sequence of N query-deletion operation pairs must require at least $\Omega(N^{3/2})$ time.

A new data structure achieved by the transform of Theorem 6.3, and the application of that structure in a matching algorithm, can be found in Appendix II.

7. CONCLUSIONS

We will now briefly review the contributions of this paper. The subject throughout has been general methods for converting static data structures to dynamic data structures. In Section 3 we saw three distinct classes of transformations, each based on a combinatorial representation of the integers. In Section 4 we saw that many of those transformations are optimal, in a very strong sense. In Section 5 we considered structures in which each insertion must be handled very quickly; this is important in

well use any constant A in the range $(0, 1)$. For small A, the query time decreases and the storage utilization is higher; for large A, the deletion time decreases.

As an application of this transformation, we will consider the problem of Empirical Cumulative Distribution Function (ECDF) searching in a set of N d-dimensional vectors. One vector is said to *dominate* another if it is greater than it in all components; an ECDF query asks for the number of vectors a given vector dominates. Note that ECDF searching is decomposable with $\square$ interpreted as *plus*. Bentley and Shamos (1977) describe a data structure for d-dimensional ECDF searching (for $d \geq 2$) with performances

$$P_{\text{ECDF}}(N) = O(N \lg^{d-1} N),$$
$$S_{\text{ECDF}}(N) = O(N \lg^{d-1} N),$$
$$Q_{\text{ECDF}}(N) = O(\lg^{d} N).$$

We can apply the binary transform of Section 3.1 and the transform of Theorem 6.2 to their structure to achieve the following.

NEW DATA STRUCTURE 6 (dynamic ECDF searching). It is possible to achieve a data structure for dynamic ECDF searching in which performing a sequence of N insertions and deletions requires $O(N \lg^{d} N)$ time. When representing N elements, the structure requires $O(N \lg^{d-1} N)$ space, and an ECDF query can be answered in $O(\lg^{d+1} N)$ time.

Lueker (1979) later used a different transformation on decomposable searching problems to achieve an online structure with performance identical to this, but with a logarithmic factor removed from the query time; his structure is more difficult to code, prove correct, and analyze, however.

6.3 *Structures Supporting Deletions Only*

In the two previous subsections we have examined structures that support insertions, deletions, and queries. In this subsection we will turn our attention to structures that support only deletions and queries, and do not allow insertions. Such structures are interesting both because of their symmetry with the "insertion-only" structures of previous sections, and because such a structure leads to a new algorithm with best-known running time for a particular problem. (Because that algorithm is rather difficult to describe, we defer discussion of it to Appendix II.)

We will now show how to maintain a set, S, of N elements under the operations of Build, Delete, and Query. For convenience we will assume that $N = M^2$, where M is a positive integer. The Build operation partitions

space required by the structure during the sequence. With this background we can describe the transformation supporting deletions precisely in the following theorem.

THEOREM 6.2 (transformations supporting deletions). *Assume that there exists an admissible* $(F(N), G(N))$ *transformation. Then, given any static structure* S *for a decomposable searching problem* P *such that the inverse of the* $\square$ *operator for* P *is computable in constant time, it is possible to achieve a new structure* DD *with performances*

$$S_{DD}(M,N) \leq S_S(2(N-M)) + S_S(N-M),$$

$$P_{DD}(M,N) \leq G(N)\cdot P_S(N),$$

$$Q_{DD}(M,N) \leq F(2(N-M))\cdot Q_S(2(N-M)) + F(N-M)\cdot Q_S(N-M),$$

$$D_{DD}(M,N) \leq G(M)\cdot P_S(M) + P_S(2M).$$

We assume here that Q_S *is monotone nondecreasing and that both* P_S *and* S_S *grow at least linearly.*

Proof. The DD structure maintains two dynamic structure (each achieved by applying the admissible $(F(N), G(N))$ transform to S): the real structure and the ghost structure. Both structures are initially empty. To insert a new element into DD, insert it into the real structure. To answer a query, answer it on the real structure and subtract from that the answer on the ghost structure (using $\square^{-1}$). To delete an element, insert it into the ghost structure. If the ghost structure ever becomes half the size of the real structure, rebuild the real structure with only undeleted elements, and discard the current ghost structure.

The storage requirements of DD follow immediately from the superlinear growth of S_S. If a total of N insertions and M deletions have been performed, then at most $N-M$ elements are "really" stored in the structure. The ghost structure can therefore contain at most $N-M$ elements, and the real structure contains at most twice that number. The time spent on insertion is straightforward, and so is the query time. The time spent on deletion is at most that for inserting M elements into the ghost structure and then rebuilding the real structure; the latter action is never carried out on more than $2M$ elements. These facts together establish the theorem. Q.E.D.

There are two important facts to note about the transformation of Theorem 6.2. The first is that it is not online in the sense of Section 5; as it stands, the expense of rebuilding the real structure and discarding the ghost structure must occasionally be paid in a single block of time. The second fact is that there is nothing magic about insisting that the ghost structure be at most one-half the size of the real structure: we could just as

We saw in that subsection that this structure can be transformed to yield the binomial list data structure that efficiently supports both insertions and member queries. It is a trivial modification to have it support count queries as well; the $\square$ operator is now *plus* rather than *or*.

Binomial lists can be modified to support deletion by keeping two binomial lists at all times, which we will call the *real* and the *ghost* structures. Each time an element is inserted, it is inserted into the real structure. When an element is deleted, we insert it into the ghost structure. To count the number of times an element occurs in the set, we count the number of times it occurs in the real structure and subtract from that the number of times is occurs in the ghost structure. We maintain the further invariant that the ghost structure always holds fewer than half as many elements as the real structure; when deletion of an element violates this invariant we destroy the ghost structure, unbuild the set of elements in the real structure and subtract all deleted elements from it, and finally rebuild that set into a new real structure (giving an empty ghost structure).

We must now analyze the performance of binomial lists with deletions. The costs of inserting an element and of performing a count search remain the same; they are respectively $O(\lg N)$ and $O(\lg^2 N)$. The "immediate" cost of deleting an element is $O(\lg N)$ (for performing the insertion into the ghost structure); we must also count, however, the cost of rebuilding the structure. The cost of rebuilding an $M/2$-element real structure is incurred only after $M/2$ elements have been deleted; since the total cost is $O(M \lg M)$, we can assign each element a share proportional to $\lg M$. Thus the cost of deletion in an N-element set can be amortized to $O(\lg N)$.

The strategy of using real and ghost structures can be generalized to give a dynamic structure supporting deletions for any decomposable searching problem whose $\square$ operator has an inverse. The most common case is that in which $\square$ is *plus*, for which $\square^{-1}$ is *minus*. If $\square$ is *and* or *or*, then one can often transform the problem to involve plus instead (for instance, we could transform member queries to count queries, whose $\square$ operator is invertible). If $\square$ is *multiset union*, then this scheme works only when the size of the answer set for the ghost structure is much smaller than the size of the total answer set (and this is often not the case). Finally, if $\square$ is *min* or *max*, this scheme is usually impossible to apply.

To describe the strategy more precisely we will need some notation to describe the efficiency of structures with deletions. If DD is a dynamic structure supporting deletions, we let $P_{DD}(M, N)$ denote the total insertion cost involved in a sequence of N insertions and M deletions in an initially empty structure. The function $Q_{DD}(M, N)$ denotes the cost of answering a query in a structure built by N insertions and M deletions. Finally, $D_{DD}(M, N)$ denotes the total time spent in processing deletions in a series of N insertions and M deletions, and $S_{DD}(M, N)$ denotes the maximum

(The right-hand side is from the fact that at least one-half of the queries access a structure of size $N/2C^*(N)$, and the adversary always deletes that structure.) We also know that

$$Q^*_{DD}(N) = \Omega(C^*(N)),$$

because each structure queried costs at least some constant. Multiplying these two inequalities yields

$$\begin{aligned}[Q^*_{DD}(N)]\cdot[I^*_{DD}(N) + D^*_{DD}(N) + Q^*_{DD}(N)] &= \Omega(C^*(N)\cdot P_S(N/2C^*(N))) \\ &= \Omega(P_S(N)) \\ &= \Omega(N).\end{aligned}$$

The last two inequalities both follow from the fact that P_S grows at least linearly. Q.E.D.

Several authors have recently proposed static-to-dynamic transformations with deletion that come close to achieving this lower bound by always keeping approximately $N^{1/2}$ static structures, each of size approximately $N^{1/2}$. The lower bound of this section shows that such transformations are the best that can be achieved in the general case. Fortunately, however, additional information can often be used to achieve more rapid deletion outside the model for which this lower bound holds. (Any such transform, however, is not applicable to all decomposable searching problems.)

6.2 *A Fast Special Case*

Theorem 6.1 shows that any quest for an efficient deletion transformation *for all all decomposable searching problems* must be in vain. In this section we will see a transformation that does in fact efficiently support deletions as well as insertions, but is not applicable to all decomposable searching problems. We will investigate this transform by first studying a particular example, and then turn to the general case.

The particular problem that we will study is that of *counting* the number of times a given element occurs in a multiset. A suitable static structure for this problem is the sorted array, which we discussed in Subsection 3.1; it has performances

$$\begin{aligned}P_{SA}(N) &= O(N \lg N), \\ S_{SA}(N) &= O(N), \\ Q_{SA}(N) &= O(\lg N).\end{aligned}$$

Section 4. We assume that there is a static structure S with operations Build and Query, which have performances P_S and Q_S, respectively. The function P_S grows at least linearly, and Q_S is positive and monotone nondecreasing. There is no way to answer a query other than by using the Query subroutine (on a structure built by Build) and the $\square$ operator. The only costs that we will count are those of P_S, Q_S, and a constant cost for computing $\square$.

To state the lower bound precisely, we need some definitions. For a dynamic structure with deletions (which we call DD) we will define the functions $I^*_{DD}(N)$, $D^*_{DD}(N)$, and $Q^*_{DD}(N)$ for the insertion, deletion, and query costs, respectively. To strengthen our result, we let these costs denote not the worst-case times, but rather the average cost (over a distribution that we will make precise in the proof of the theorem). We are now ready to state and prove the primary theorem of this subsection.

THEOREM 6.1 (expense of deletion). *For any dynamic structure with deletions (which we call* DD*) obtained by a transformation applicable to all decomposable searching problems, there exists a sequence of insertions, deletions, and queries for which*

$$[Q^*_{DD}(N)]\cdot[I^*_{DD}(N) + D^*_{DD}(N) + Q^*_{DD}(N)] = \Omega(N).$$

Note that this implies that at least one of the insertion, deletion, and query costs requires at least $\Omega(N^{1/2})$ time.

Proof. We will prove this theorem by considering a "steady state" in which there is a structure of size N, and a sufficiently long string of repeated query, delete, and insert operations is performed. After M repetitions of these operations, the structure will still be of size N, and a total of M queries will have been performed. Each query that is performed must examine some collection of static structures whose total size is at least N (so that each element of the set is represented in the query); assume that $C^*(N)$ such structures are examined on the average. We therefore know that at least half the queries examine no more than $2C^*(N)$ static structures each (if more were examined, then the average would be too high), and in these cases the largest structure examined must contain at least $N/(2C^*(N))$ elements.

Consider now an adversary that causes each deletion in the sequence to be deleted from the largest existing static structure—because of our model of computation, this structure must now be discarded. For sufficiently long sequences of operations, static structures must be created as often as they are discarded. The costs of building the static structure must therefore be paid in insertion, deletion, and query costs, yielding

$$I^*_{DD}(N) + D^*_{DD}(N) + Q^*_{DD}(N) \geq \tfrac{1}{2}P_S(N/2C^*(N)).$$

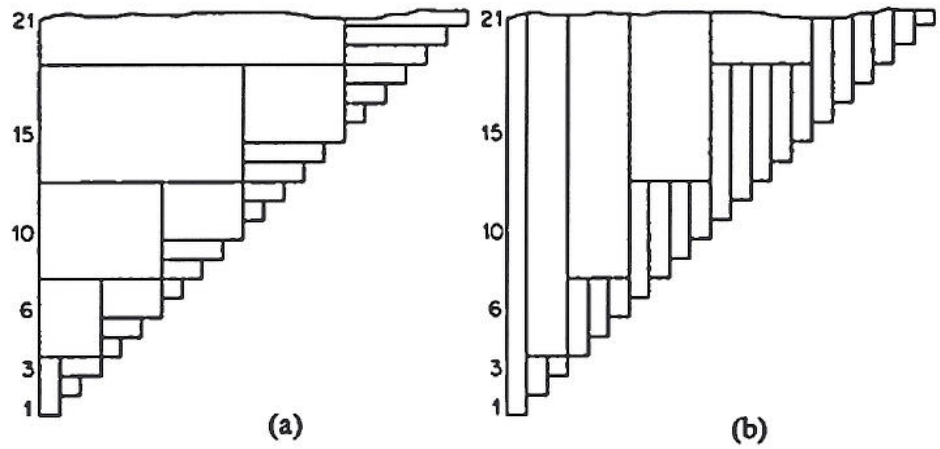

FIG. 5.2. (a) Online triangular transform. (b) Online dual triangular transform.

Similarly, the *online dual triangular transform*, shown in Fig. 5.2b, achieves

$$I_D(N) \leq 2I_S(N),$$
$$Q_D(N) \leq 3(2N)^{1/2}Q_S(N),$$
$$S_D(N) \sim S_S(N).$$

Determination of good lower bounds for the penalty factors associated with online transformations remains an open problem.

6. Transformations That Support Deletion

So far in this paper we have considered dynamic data structures that support only insertions and queries. In this section we will present two results dealing with data structures that support deletions along with insertions and queries, and the realization of such structures by decomposable transforms. In Subsection 6.1 we present a negative result that says that, in general, it is impossible to achieve by a transform a data structure that efficiently supports deletions. In Subsection 6.2 we will examine a transformation that efficiently achieves deletion, but is applicable only to a subset of the decomposable searching problems. We then examine in Subsection 6.3 a transformation that yields structures that support deletions and queries, but no insertions.

6.1 *A Lower Bound*

In this subsection we will study a lower bound on the efficiency of performing deletion in a structure achieved by a decomposable transformation. As with all lower-bound proofs, it is important that we accurately define our model of computation, which is very similar to that used in

during the Nth insertion have cardinalities which are exact powers of 2 and which are $\leq N$. Moreover, there are never more than two active structures of any given cardinality. This and assumption (1) justify the claim about Q_D. Similarly, assumption (3) and the fact that there is never more than one pending structure of any cardinality together justify the claim about I_D. Finally, we note that the sum of the cardinalities of all structures active and pending after the Nth insertion is no more than $3N$ (N for the active structures and no more than $2N$ for the pending structures). Together with assumptions (4) and (5), this fact justifies the claim about S_D. Q.E.D.

To illustrate the application of the online binary transformation, we will consider the problem of d-dimensional maxima searching. A vector is said to be maximal with respect to a set of vectors if no vector in the set is greater than the given vector in all coordinates. Preparata (1978) has given a data structure SMS for d-dimensional maxima searching with performances

$$P_{SMS}(N) = O(N \lg^{d-2} N),$$
$$S_{SMS}(N) = O(N \lg^{d-2} N),$$
$$Q_{SMS}(N) = O(\lg^{d-2} N),$$

for any $d \geq 3$. Applying the online binary transform to this structure yields the following.

NEW DATA STRUCTURE 5 (dynamic maxima searching). For any fixed $d \geq 3$ there exists a dynamic data structure DMS for d-dimensional maxima searching with performance

$$I_{DMS}(N) = O(\lg^{d-1} N),$$
$$Q_{DMS}(N) = O(\lg^{d-1} N),$$
$$S_{DMS}(N) = O(N \lg^{d-2} N).$$

This structure has the same performance as Lueker's (1979), but is substantially easier to code and prove correct; his structure, however, also supports deletions. (The two structures were discovered independently.)

The other transforms we have studied may also be modified to give online versions, as shown by the examples in Fig. 5.2. The *online triangular transform*, shown in Fig. 5.2a, gives the performance

$$I_D(N) \leq (2N)^{1/2} I_S(N),$$
$$Q_D(N) \leq 3Q_S(N),$$
$$S_D(N) \leq 2S_S(N).$$

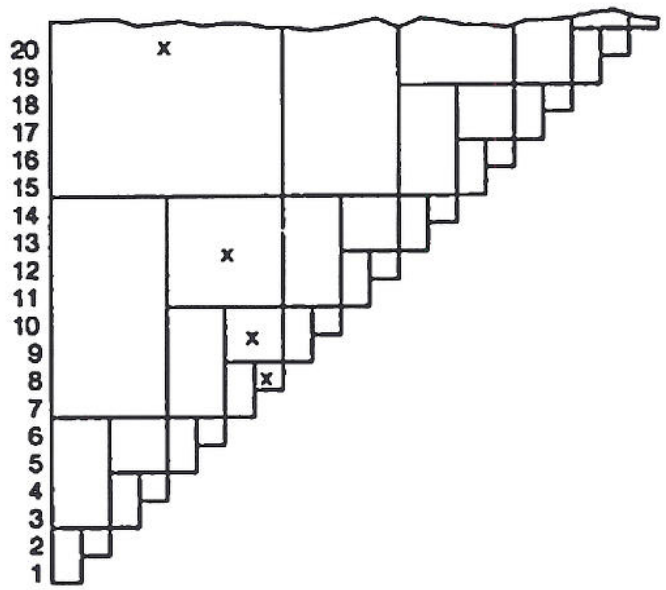

FIG. 5.1. The online binary transform.

transformation the *online* binary transformation. Analysis of this transform's performance yields the following theorem.

THEOREM 5.1 (the on-line binary transformation). *Suppose we are given a static structure, S, for a decomposable problem such that*

(1) $Q_S(N)$ *is monotone nondecreasing,*

(2) *a structure of cardinality N may be built by N calls, each of cost* $I_S(N)$ *(recall that* $I_S(N)$ *is defined as* $P_S(N)/N$*),*

(3) $I_S(N)$ *is montone nondecreasing,*

(4) *the space used at any point during the formation of a static structure is at most* $S_S(N)$*, and*

(5) $S_S(N)$ *grows at least linearly.*

Then, there exists a dynamic structure, D, such that

$$Q_D(N) \leq 2\lfloor \lg(N+1) \rfloor Q_S(N),$$
$$I_D(N) \leq \lceil \lg N \rceil I_S(N),$$
$$S_D(N) \leq 3S_S(N).$$

(Recall that $I_D(N)$ *is the worst-case time to insert the Nth element in a dynamic structure.)*

Proof. By assumption (2), application of the online binary transform is well defined. We will now show that the resulting dynamic algorithm has the stated performance. We first note that all structures which are either active (completed but not yet discarded) after the Nth insertion or pending