JOURNAL OF ALGORITHMS 1, 301–358 (1980)

# Decomposable Searching Problems I. Static-to-Dynamic Transformation*

JON LOUIS BENTLEY† AND JAMES B. SAXE

*Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213*

Received October 29, 1979; revised April 15, 1980

Transformations that serve as tools in the design of new data structures are investigated. Specifically, general methods for converting static structures (in which all elements are known before any searches are performed) to dynamic structures (in which insertions of new elements can be mixed with searches) are studied. Three classes of such transformations are exhibited, each based on a different counting scheme for representing the integers, and a combinatorial model is used to show the optimality of many of the transformations. Issues such as online data structures and deletion of elements are also examined. To demonstrate the applicability of these tools, several new data structures that have been developed by applying the transformations are studied.

## 1. INTRODUCTION

The design of efficient data structures for searching problems is an important and difficult problem. In this paper we will investigate a set of *transformations* that aid in the design of such data structures, and illustrate

*This research was supported in part by the Office of Naval Research under Contract N00014-76-C-0370.

†Also with the Department of Mathematics.

0196-6774/80/040301-58$02.00/0

the use of those transformations by describing a number of new structures that have been designed by applying the transformations.

Since this paper is the first of a multipart series, we will now take a moment to describe briefly the common thread running through the work. The work deals with a class of problems called the *decomposable searching problems*, which includes most of the searching problems that have been discussed in the literature (the term "searching problem" is used here in a precise sense which we state formally in Section 2). The decomposable searching problems share the property that any data structure for solving them can also be applied as a "subroutine" in solving related problems. The objects that we will study in this work are *transformations* that can apply *any* data structure for solving *any* decomposable searching problem to solve a closely related searching problem.

The specific transformations that we will examine in this paper convert static structures (which are built once-for-all before any queries are asked) into dynamic structures (in which queries can be mixed with insertions, and perhaps deletions). In Section 2 we will examine definitions and notation necessary for discussing the transformations. The transformations are discussed in Section 3, and a proof of their optimality is given in Section 4. Online data structures and deletion are the subjects of Sections 5 and 6, and conclusions are offered in Section 7.

In later parts of this paper we will study two additional types of transformations. The first type of transformation adds a "range variable" to a query; specifically, we can associate a new variable with every object in the set and then restrict each query to objects that have that variable in a certain range, which may vary from query to query. The second type of transformation we will see facilitates tradeoffs between the query time required by the structure and the time and space required to build and store it. Readers interested in a preliminary description of these results are referred to Bentley (1979).

## 2. Definitions and Notation

In this section we will review a number of basic concepts that have to do with searching problems and give a number of definitions that will be used throughout the paper. The casual reader may therefore skim most of this section; the only part that need be read in detail is the definition of the decomposable searching problems.

We will use the term *searching problem* in a restricted sense throughout this paper. Specifically, we refer to maintaining a set $F$ of objects so that *queries* that ask the relation of a new object $x$ to the set $F$ can be answered quickly. The most common example of a query is what we call a *Member*

Query: "is $x$ a member of $F$?" If $F$ were a set of reals, we might be interested in a *Nearest Neighbor* query: "what is the distance from $x$ to the real in $F$ closest to it?" The general query is that a question containing a variable of type $T1$ is asked of a set of elements of type $T2$, giving an answer of type $T3$. In a Member query, $T1$ and $T2$ are the same, and $T3$ is boolean. In a Nearest Neighbor query, both $T1$ and $T2$ are real, and $T3$ is a nonnegative real. In the general case, the query $Q$ can be viewed as a function mapping a $T1$ and a set of $T2$'s to a $T3$, or

$$Q: T1 \times 2^{T2} \to T3.$$

Throughout this paper we will identify a *searching problem* by its query; a *solution* to a searching problem is a data structure that allows the query to be answered quickly.

In this paper we will study data structures for a class of searching problems called the *decomposable searching problems*. A searching problem with query operation $Q$ is decomposable if there exists an efficiently computable binary operator $\Box$ satisfying the condition

$$Q(x, A \cup B) = \Box[Q(x,A), Q(x,B)].$$

(Note that this definition implies that $\Box$ is both associative and commutative.) For example, the member searching problem is decomposable because

$$\text{Member}(x, A \cup B) = \bigvee[\text{Member}(x,A), \text{Member}(x,B)],$$

and (distance to) nearest neighbor searching is decomposable because

$$\text{NN}(x, A \cup B) = \min[\text{NN}(x,A), \text{NN}(x,B)].$$

We will investigate a number of decomposable searching problems throughout this paper; a list of many of them can be found in Appendix I. All of the transformations that we will see later in this paper are applicable precisely to the decomposable searching problems. They exploit decomposability by partitioning a set into subsets, and answer a query by computing answers on the subsets and then using the $\Box$ operator to combine those subanswers to yield a solution to the entire problem. Note that the $\Box$ operator is essential in this strategy.

There are two subclasses of the decomposable searching problems that will be of special interest later in the paper. The first subclass consists of those problems whose $\Box$ operator has a "zero" (or "sticky") element; that is, there exists some element $z$ such that for every element $x$,

$$\Box(z,x) = z.$$

For example, *false* is a zero for $\wedge$, and *true* is a zero for $\vee$. A second class that will be of interest consists of the problems for which the $\square$ operator has an inverse (for example, if $\square$ is addition, its inverse is subtraction). We will examine in detail both of these subclasses of the general decomposable searching problems later in the paper.

We will make a distinction between two types of data structures for solving searching problems. A *static* structure is built once and then searched many times; insertions and deletions of elements are not allowed. To describe the performance of the static structure $A$ we give three functions of $N$, the number of elements in the set represented by $A$:

$P_A(N)$ = the *preprocessing time* required to build $A$,

$Q_A(N)$ = the *query time* required to perform a search in $A$, and

$S_A(N)$ = the *storage* required to represent $A$.

(Unless explicitly noted otherwise, throughout this paper we will deal only with worst-case cost functions.) A second type of data structure is the *dynamic* structure. This structure is initially empty, and the three operations available on it are for inserting a new element, for deleting a current element, and for performing a search. We analyze the performance of the dynamic structure $B$ by giving the functions

$I_B(N)$ = the *insertion time* for $B$,

$D_B(N)$ = the *deletion time* for $B$,

$Q_B(N)$ = the *query time* required to perform a search in $B$, and

$S_B(N)$ = the *storage* required to represent $B$.

Later in this paper we will want to "mix apples and oranges" and compare the performance of the static structure $A$ with that of the dynamic structure $B$. To facilitate such comparisons we define the "insertion" time for the static structure $A$ as

$$I_A(N) = P_A(N)/N,$$

which is the cost of building an $N$-element structure amortized over the $N$ elements it represents. Likewise we define the cost of "processing" the dynamic structure $B$ to be

$$P_B(N) = \sum_{1 \le i \le N} I_B(i).$$

## 3. Transformations That Support Insertions

In this section we will investigate transformations that convert a static data structure for a decomposable searching problem into a dynamic data structure. We will restrict ourselves to the special case of dynamic structures that support only the operations of *inserting* a new element and *searching* to answer a query; we will return to the issue of deletion in Section 6.

## 3.1. *The Binary Transformation*

In this subsection we will examine a static-to-dynamic transformation that is based on the binary representation of the integers. We will study the transformation by first examining its application to the particular problem of nearest neighbor searching in the plane, and then discussing its more general properties.

In planar nearest neighbor searching we must organize a set of $N$ points in the plane so that subsequent queries can tell the distance from the query point $x$ to its nearest neighbor in the set. Therefore, objects of types $T1$ and $T2$ are points in $\mathbb{R}^2$, and those of type $T3$ are nonnegative reals. (Henceforth we will use the term "nearest neighbor searching" to refer only to the planar case; for ease of discussion we consider only the problem of finding the distance to the nearest neighbor and not that of finding the point realizing that distance.) Note that nearest neighbor searching is decomposable because it satisfies

$$\mathrm{NN}(x, A \cup B) = \min[\mathrm{NN}(x,A), \mathrm{NN}(x,B)].$$

Lipton and Tarjan (1977) have described an elegant static data structure for nearest neighbor searching (which we will call LT) with performance

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

Many applications, however, call for dynamic nearest neighbor searching, and the Lipton–Tarjan structure does not appear to be suitable for a modification that would facilitate insertions. We will now investigate a new structure (called DNN for dynamic nearest neighbor) that uses the Lipton–Tarjan static structure only as a subroutine, and does not try to modify the structure. The DNN structure that we will describe is the best of the known structures for performing dynamic nearest neighbor searching in the plane.

The DNN structure will consist of a set of LT's; that is, the elements (points) currently stored in the DNN will be partitioned into subsets that are themselves represented by LT's. When there is one element in the DNN, there is an LT containing that single element. When the second element is inserted, that LT is discarded and a new LT of size 2 is created.

At the arrival of the third element, a new LT of size 1 is created. This process continues so that when there are $N$ elements represented by the DNN, there are LT's corresponding to all of the one bits in the binary representation of $N$. For example, when there are 79 elements in the DNN, there are LT's of sizes 64, 8, 4, 2, and 1. When the 80th element is inserted, the four smallest structures are discarded and a new structure of size 16 is built. At any time in this process the distance to the nearest neighbor of a query point $x$ can be found by locating its nearest neighbors in each of the LT's (using the $O(\lg N)$ algorithm) and taking the minimum of the distances; it is here that we make essential use of decomposability.

This scheme is illustrated pictorially in Fig. 3.1 by a diagram commonly used to represent binary counting. The vertical axis in that figure denotes the number of elements currently in the dynamic structure. Each rectangle (square) represents a particular static LT structure; consider, for example, the $4 \times 4$ square that comes into existence at time 4 and is then replaced at time 8. The LT structures in existence at time $T$ can be found by drawing a horizontal line that intersects the vertical axis at $T$; for example, at time 7 there are three structures in existence—of sizes 4, 2, and 1. We will find later that this type of diagram (which we call a "history diagram") is a handy way of representing transformations.

It is easy to analyze the performance of the DNN structure given the performance of the LT structure. Since the LT requires linear storage and the DNN just partitions its elements into LT's, the DNN will also require linear storage. A DNN of $N$ elements will keep at most $\lg(N + 1)$ LT's (each of size not greater than $N$), so the query time of a DNN is bounded above by $\lg(N + 1)$ times the cost of querying an LT. The cost of inserting an element into a DNN is more difficult to analyze; note that while inserting the 1023rd element is essentially free, the 1024th element is very expensive, since a new structure of size 1024 must be built. We will therefore count the cost of inserting the first $N$ elements into an initially empty structure, which is exactly $P_{\mathrm{DNN}}(N)$. We will perform this analysis

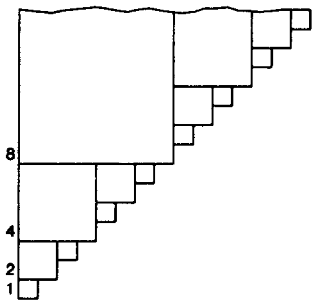

FIG. 3.1. The binary transform.

only for the case that $N = 2^j - 1$, for some nonnegative integer $i$, and discuss later the behavior of the function at other values of $N$. If we have inserted $2^j - 1$ elements, then we have built one LT structure of size $2^{j-1}$, two LT structures of size $2^{j-2}$, and $2^{k-1}$ structures of size $2^{j-k}$. (This is a trivial property of binary counting.) The total cost of inserting these elements is therefore

$$P_{\mathrm{DNN}}(2^j - 1) = 1 \cdot P_{\mathrm{LT}}(2^{j-1}) + 2 \cdot P_{\mathrm{LT}}(2^{j-2}) + \cdots + 2^{j-1} \cdot P_{\mathrm{LT}}(1).$$

For $N$ a power of 2 we can rewrite this as

$$P_{\mathrm{DNN}}(N - 1) = 1 \cdot P_{\mathrm{LT}}(N/2) + 2 \cdot P_{\mathrm{LT}}(N/4) + \cdots + (N/2) \cdot P_{\mathrm{LT}}(1).$$

We know that $P_{\mathrm{LT}}(N) = O(N \lg N)$, which means that $P_{\mathrm{LT}}(N) \leq cN \lg N$, for some positive constant $c$. Substituting this into the above equation yields

$$\begin{aligned} P_{\mathrm{DNN}}(N - 1) &\leq c \cdot [1 \cdot (N/2 \lg N/2) + 2 \cdot (N/4 \lg N/4) \\ &\qquad + \cdots + (N/2) \cdot (1 \lg 1)] \\ &= (cN/2) \cdot [\lg N/2 + \lg N/4 + \cdots + \lg 1] \\ &\leq (c/2) N \lg^2 N \\ &= O(N \lg^2 N). \end{aligned}$$

This completes our analysis of the DNN structure, establishing the following.

NEW DATA STRUCTURE 1 (dynamic nearest neighbor). The DNN structure for dynamic nearest neighbor searching in the plane has performances

$$\begin{aligned} P_{\mathrm{DNN}}(N) &\leq P_{\mathrm{LT}}(N) \cdot \lg(N + 1) = O(N \lg^2 N), \\ Q_{\mathrm{DNN}}(N) &\leq Q_{\mathrm{LT}}(N) \cdot \lg(N + 1) = O(\lg^2 N), \\ S_{\mathrm{DNN}}(N) &\leq S_{\mathrm{LT}}(N) = O(N). \end{aligned}$$

Note that the cost of doing $N$ pairs of insert, Query operations in the DNN structure is $\theta(N \lg^2 N)$; all other known dynamic nearest neighbor structures require $\Omega(N^2)$ time for the task.

The binary transformation that we have just described for nearest neighbor searching is applicable to any decomposable searching problem: given a static data structure for a particular problem, a dynamic structure is maintained by keeping a set of static structures, each of which represents a set whose cardinality is a power of 2. Insertion is accomplished by the same technique of binary counting. A query can be answered by querying all the static structures in existence at the time of the query, and combining the answers by repeated application of the $\square$ operator.

A computer program that implements the binary transform is sketched in Program 3.2. It assumes the existence of a static structure $S$ with operations $\text{Query}_S$, $\text{Build}_S$, and $\text{Unbuild}_S$ ($\text{Unbuild}_S$ returns the elements currently stored in the structure as a linked list).[1] The code implements a dynamic structure $D$ by providing routines $\text{Init}_D$ (which initializes the structure to be empty), $\text{Insert}_D$, and $\text{Query}_D$. It implements the binary strategy by maintaining a one-way infinite array $P$ with the invariant that $P[i]$ is either empty or contains a static structure of size $2^i$. The variable High is an integer that is one greater than the index of the last nonempty structure; $P[\text{High}]$ is always empty. $\text{Init}_D$ initializes the structure to have this invariant. $\text{Query}_D$ answers a query by iterating through the structures and combining the answers by the $\square$ operator. $\text{Insert}_D$ can be understood most easily by considering incrementing a binary integer by one: to do so, we scan from right to left, changing ones to zeros until we come to the first zero (which we then make a one). An Alphard program very similar to the code in Program 3.2 has been given by Bentley and Shaw (1980); they also provide both a precise specification of the transform and a proof that the program indeed meets its specifications.

```
proc $\text{Init}_D \leftarrow$
    $P[0] \leftarrow \varnothing$; High $\leftarrow 0$

proc $\text{Insert}_D(x) \leftarrow$
    $S \leftarrow \{x\}$
    $i \leftarrow 0$
    while $P[i] \neq \varnothing$ do
        $S \leftarrow S \cup \text{Unbuild}_S(P[i])$; $P[i] \leftarrow \varnothing$
        $i \leftarrow i + 1$
    $P[i] \leftarrow \text{Build}_S(S)$
    if $i =$ High then
        High $\leftarrow$ High $+ 1$; $P[\text{High}] \leftarrow \varnothing$

func $\text{Query}_D(x) \leftarrow$
    $A \leftarrow \text{Query}_S(x, P[0])$
    for $i \leftarrow 1$ to High-1 do
        $A \leftarrow \square(A, \text{Query}_S(x, P[i]))$
    return $A$
```

PROGRAM 3.2. Sketch of code for the binary transform.

[1]Throughout this paper we will retrieve a set of $T2$'s from a structure by unbuilding the structure. In some applications it might be more efficient to store the set along with the structure.

The analysis of the general transformation is quite similar to the analysis of the DNN structure.[2] Since at most $\lg(N+1)$ static structures exist for an $N$-element dynamic structure, if we assume that the static query cost is monotone nondecreasing then we have

$$Q_D(N) \le Q_S(N) \cdot \lg(N+1).$$

To analyze the storage and processing costs we need the following definition: a function $F$ is said to *grow at least linearly* if for every two positive integers, $M$ and $N$, where $M < N$,

$$F(M)/M \le F(N)/N.$$

A consequence of this definition is that if $F$ is a function that grows at least linearly and $A$ and $B$ are positive integers, then

$$\begin{aligned} F(A+B) &= A[F(A+B)/(A+B)] + B[F(A+B)/(A+B)] \\ &\ge F(A) + F(B). \end{aligned}$$

Since the dynamic structure partitions its elements among static structures without replication, if the storage costs $S_S$ of the static structure grows at least linearly then we have the relation

$$S_D(N) \le S_S(N).$$

To analyze the processing cost we will first consider the case that $N$ is a power of 2; the reasoning used in our analysis of DNN shows that

$$P_D(N-1) = P_S(N/2) + 2P_S(N/4) + \cdots + (N/2)P_S(1).$$

When $P_S$ grows at least linearly, we know that $P_S(2i) \ge 2P_S(i)$ and we can use this fact inductively to show that

$$\begin{aligned} P_D(N-1) &\le P_S(N/2) + P_S(N/2) + \cdots + P_S(N/2) \\ &= P_S(N/2) \cdot \lg N. \end{aligned}$$

We will now use a less accurate (but more general) analytic technique to establish the value of $P_D(N)$ for $N$ not one less than a power of 2. Note that after $N$ elements have been inserted, any particular element has been in at most $\lg(N+1)$ distinct static structures. We will now show that for any transform, if every element has been built into at most $k$ structures, then the static and dynamic processing costs are related by

$$P_D(N) \le P_S(N) \cdot k.$$

[2] In the analysis of the transformed structure we will count only the costs incurred by operations on the original structure. Examination of the code in Program 3.2 shows that the overhead costs for both Insert and Query are a small constant times $\lg N$.

(This immediately yields the corollary that

$$P_D(N) \le P_S(N) \cdot \lg(N+1)$$

for the binary transform, for any positive $N$.) Consider the cost that any particular element, $E$, contributes to $P_D(N)$. Each time $E$ is built into a new static structure of size $M$, we can assign it a share of that cost of $P_S(M)/M$. Because $P_S$ grows at least linearly and $M$ is less than or equal to $N$, we know that

$$P_S(M)/M \le P_S(N)/N.$$

and we can therefore assign $E$ this latter cost as an upper bound. Multiplying the number of distinct elements ($N$) by the number of times each is built into a static structure (less than $k$) times this cost yields the desired result.

To enable us to speak more precisely about transforms on data structures for decomposable searching problems, we need the following definition.

DEFINITION 3.1 (admissible transform). A transformation on decomposable searching problems is said to be an *admissible* $(F(N), G(N))$ *transform* if it converts the static structure $A$ into a dynamic structure $B$ whose semantics are correct *assuming only the property of decomposability*, and whose performance satisfies the relations[3]

$$Q_B(N) \le Q_A(N) \cdot F(N),$$
$$P_B(N) \le P_A(N) \cdot G(N),$$
$$S_B(N) \le S_A(N),$$

assuming only that $Q_A$ is monotone nondecreasing and that both $P_A$ and $S_A$ grow at least linearly.[4]

This definition will be further refined and presented as a precise model of computation in Subsection 4.1.

[3]To simplify the analysis, we will count only the costs of calls to operations on the static structure, and not the costs of bookkeeping operations nor the cost of combining the results of queries into different static structures. Careful examination of our algorithms will show that these extra costs add only a small constant factor (which does not depend on $F$ or $G$) to the computation times. In most cases, this constant quickly approaches unity as $N$ increases. Similarly, the only storage we charge to the dynamic structure is that used for storing instances of the static structure. Again, this is generally the dominant cost.

[4]For cases where $P_A$, $Q_A$, and $S_A$ do not satisfy these criteria, we may choose functions $P'_A$, $Q'_A$, and $S'_A$ that (a) satisfy the criteria and (b) dominate $P_A$, $Q_A$, and $S_A$, respectively. The relations given above will then hold between the dynamic cost functions and $P'_A, Q'_A, S'_A$.

We can now state precisely the fact that the binary transform efficiently converts a static data structure to a dynamic structure as Theorem 3.1.

THEOREM 3.1 (the binary transform). *The binary transform is an admissible* $(\lg(N+1), \lg(N+1))$ *transform.*

*Proof.* Given in the preceding text. Q.E.D.

To illustrate some "tricks" available in using the binary transform, we will study its application to the member query problem using the data structure of a sorted array. Precisely, consider the static data structure for member searching that stores the elements in increasing order in an array (built by sorting the set), and answers a query by performing a binary search. The analysis of this structure (which we call SA, for sorted array) shows

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

Consider the dynamic member searching structure achieved by applying the binary transformation to SA: we always maintain a set of sorted arrays, each of size a power of 2. A particularly efficient representation of this structure (which we will call BL, for binomial list[5]) is to store these sorted arrays sequentially in one large array, with the largest sorted segment (which we call a *run*) leftmost in the array. Two snapshots of a BL are shown in Fig. 3.3; the vertical bars in the figure separate the runs in the array. By the analysis of SA and the effect of the binary transform, we can easily see that the performance of the BL structure is

$$\begin{aligned} P_{\mathrm{BL}} &= O(N \lg^2 N), \\ S_{\mathrm{BL}} &= O(N), \\ Q_{\mathrm{BL}} &= O(\lg^2 N). \end{aligned}$$

Note that very little storage is used by a BL: it requires only $N$ array words for the elements, plus approximately $\lg N$ bits to describe the cardinality of the represented set.

There is a glaring deficiency in the straightforward implementation of this structure: the obvious insertion routine inserts the 1024th element by

[5]A scheme very similar to this was proposed by John McCarthy in the context of an "on-line merge sort" (see Knuth (1973, Question 5.2.4.17)). The binomial list structure was developed for the present application by explicit application of the binary transform, and was then studied in detail by Bentley *et al.* (1978). The name is taken from its similarity to the binomial queue data structure of Vuillemin (1978).

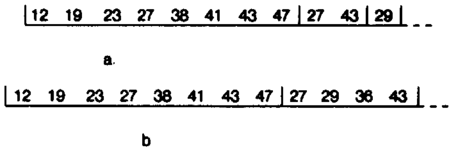

FIG. 3.3. Snapshots of a binomial list. (a) An 11-element binomial list. (b) After inserting 36.

ignoring all the structure currently in the array and resorting from scratch. A superior insertion strategy is to consider the inserted element as a rightmost one-element run, and merge that with its neighboring one-element run giving a two-element run. We then merge that with its neighbor, giving a four-element run, and so forth, until the two rightmost runs are of unequal sizes. The amount of work in building a new run in this scheme is linear in the size of the run, and the cost of inserting $N$ elements is therefore $O(N \lg N)$. We have thus avoided paying the logarithmic penalty factor inherent in the binary transform by observing that runs can be efficiently merged.[6]

We can sometimes avoid paying the transform penalty of a logarithmic slowdown in query time. Specifically, we will consider the average cost of performing a *successful* member search in a BL (that is, a search that finds the element it was looking for). If we assume that each element in the array is equally likely to be searched for, then the probability of finding the desired element in the first run is at least $1/2$. Therefore, half the time we need not search the other runs. Likewise, at least one-half of the remaining times we find the desired element in the second run, so the probability of searching the third run is less than $1/4$. Summing the cost of searching each run times the probability of performing the search, we find that a successful member search is expected to be at most twice as expensive in the BL as in the SA.

The arguments that we have just sketched have been given in detail by Bentley *et al.* (1978), who describe the following data structure.

NEW DATA STRUCTURE 2 (binomial lists). The binomial list (BL) structure for dynamic member searching has performances

$$P_{\mathrm{BL}}(N) = O(N \lg N),$$
$$Q_{\mathrm{BL}}(N) = O(\lg^2 N),$$
$$S_{\mathrm{BL}}(N) = O(N).$$

[6]Only constant extra space is required to merge consecutive runs in an array—see Knuth (1973, Exercise 5.2.4.18). The algorithm to accomplish this, however, is extremely difficult to code, and would probably not be used in any real application.

The linear storage used by this structure consists of exactly $N$ array words and $O(\lg N)$ additional bits, which is minimal.

Bentley *et al.* (1978) have investigated this structure in detail and have shown that it is optimal in a certain model of *minimum-storage* dynamic member searching. The BL structure provides an interesting point of comparison with the minimum-storage structure described by Munro and Suwanda (1979); the BL performs substantially better than their structure by working in a different model of computation.

There is yet another circumstance in which the logarithmic cost penalties of applying the binary transform do not have to be paid: when the original cost functions are fast growing. Consider, for example, a static data structure with $N^2$ preprocessing time. Our previous analysis shows that for $N$ a power of 2, we will have

$$\begin{aligned} P_D(N-1) &= P_S(N/2) + 2P_S(N/4) + \cdots + (N/2)P_S(1) \\ &= (N/2)^2 + 2(N/4)^2 + \cdots + (N/2)1^2 \\ &= (N^2/2)\cdot[1/2 + 1/4 + \cdots + 1/N] \\ &= O(N^2). \end{aligned}$$

Similar analyses show that the logarithmic penalty in processing cost is not incurred when the binary transform is applied to any static structure with preprocessing cost of $\Omega(N^{1+\epsilon})$, for any positive $\epsilon$. Likewise, it can be shown that the logarithmic penalty in query time will not have to be paid for any static structure with query time of $\Omega(N^{\epsilon})$.

This concludes our study of the binary transform. In the next two subsections we will see that this transform is but one of many possible ways of converting a static structure to a dynamic structure, at the cost of penalty factors in the preprocessing and query costs. As we study the other transforms and their performance, it is important to keep in mind that the penalty factors need not always be paid. In this subsection we have seen three ways of avoiding them: by *merging* structures instead of rebuilding them from scratch, by counting the *average search time* instead of the worst-case time (this is sometimes appropriate when the $\square$ operator has a zero element), and by performing separate analysis for *fast-growing functions*.

## 3.2. *Transformations with Fast Query Time*

The binary transform described in the last subsection provides an example of an admissible $(\lg(N+1), \lg(N+1))$ transform, and we might wonder if we can do better. In this subsection we will investigate a class of transforms that have faster query times than the binary transform at the

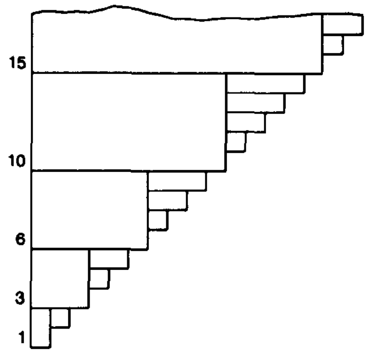

FIG. 3.4. The triangular transform.

cost of slower insertion time. Specifically, we will see that an admissible $(k,(k!N)^{1/k})$ transform exists for every positive integer $k$. To study this transform we will first investigate the case $k = 2$, and then move on to the general case.

We will call the transform for the case $k = 2$ the *triangular transform*, because it is based on the triangular numbers (that is, numbers of the form $\binom{n}{2}$). The transform is illustrated in Fig. 3.4. Note that when five elements are in the dynamic structure, there are static structures of sizes 3 and 2; when the sixth element is inserted, those structures are destroyed and a new structure of size 6 is created. At any point in the history of the dynamic structure, there will be at most two static structures in existence. The insertion algorithm creates a new "large" static structure at every triangular number; otherwise it inserts an element by unbuilding the smaller structure and building it into a new structure with one additional element. A query can be answered by searching the two static structures and combining the answers by the $\square$ operator.

The triangular structure is very easy to analyze. Because at most two static structures exist at any time, the dynamic query cost is given by

$$Q_D(N) \leq 2Q_S(N).$$

If we assume that the static storage requirements grow at least linearly, we know that the dynamic structure does not use more storage. To analyze the insertion time, consider the case in which a total of $\binom{M}{2}$ elements have been inserted. It is easy to prove by induction that no element has been built into more than $M$ structures (the proof is based on the recurrence for the triangular numbers). In general, if $N$ elements have been inserted, no single element has been built into more than $(2N)^{1/2}$ static structures. By the arguments in the previous subsection, this implies

$$P_D(N) \le P_S(N) \cdot (2N)^{1/2}.$$

These arguments together establish the following theorem.

THEOREM 3.2 (the triangular transform). *The triangular transform is an admissible* $(2, (2N)^{1/2})$ *transform.*

Just as the binary transform is isomorphic to the binary representation of the integers, so is the triangular transform isomorphic to a representation of the integers based on triangular numbers. (This system is called a "binomial number system" by Knuth (1968, Exercise 1.2.6.56).) Specifically, an integer $N$ is represented by a pair of integers $i$ and $j$ (with $i > j$) by the expression

$$N = \binom{i}{2} + \binom{j}{1}.$$

Note that both $i$ and $j$ are less than $(2N)^{1/2} + 1$; this explains the processing cost of the transform. The general transform, which we will call the $k$-binomial transform, is based on a straightforward generalization of this scheme, in which an integer is (uniquely) represented as the sum of $k$ binomial coefficients, whose lower parts are the integers 1 through $k$. This counting scheme is illustrated for the cases $k = 2$ and $k = 3$ in Fig. 3.5. Row 15 of the table is interpreted as follows: in the 2-binomial representa-

| k=2 | | | k=3 | | | |
|---|---|---|---|---|---|---|
| Integer | $\binom{}{2}$ | $\binom{}{1}$ | Integer | $\binom{}{3}$ | $\binom{}{2}$ | $\binom{}{1}$ |
| 0 = 0+0 | 1 | 0 | 0 = 0+0+0 | 2 | 1 | 0 |
| 1 = 1+0 | 2 | 0 | 1 = 1+0+0 | 3 | 1 | 0 |
| 2 = 1+1 | 2 | 1 | 2 = 1+1+0 | 3 | 2 | 0 |
| 3 = 3+0 | 3 | 0 | 3 = 1+1+1 | 3 | 2 | 1 |
| 4 = 3+1 | 3 | 1 | 4 = 4+0+0 | 4 | 1 | 0 |
| 5 = 3+2 | 3 | 2 | 5 = 4+1+0 | 4 | 2 | 0 |
| 6 = 6+0 | 4 | 0 | 6 = 4+1+1 | 4 | 2 | 1 |
| 7 = 6+1 | 4 | 1 | 7 = 4+3+0 | 4 | 3 | 0 |
| 8 = 6+2 | 4 | 2 | 8 = 4+3+1 | 4 | 3 | 1 |
| 9 = 6+3 | 4 | 3 | 9 = 4+3+2 | 4 | 3 | 2 |
| 10 = 10+0 | 5 | 0 | 10 = 10+0+0 | 5 | 1 | 0 |
| 11 = 10+1 | 5 | 1 | 11 = 10+1+0 | 5 | 2 | 0 |
| 12 = 10+2 | 5 | 2 | 12 = 10+1+1 | 5 | 2 | 1 |
| 13 = 10+3 | 5 | 3 | 13 = 10+3+0 | 5 | 3 | 0 |
| 14 = 10+4 | 5 | 4 | 14 = 10+3+1 | 5 | 3 | 1 |
| 15 = 15+0 | 6 | 0 | 15 = 10+3+2 | 5 | 3 | 2 |
| 16 = 15+1 | 6 | 1 | 16 = 10+6+0 | 5 | 4 | 0 |
| 17 = 15+2 | 6 | 2 | 17 = 10+6+1 | 5 | 4 | 1 |
| 18 = 15+3 | 6 | 3 | 18 = 10+6+2 | 5 | 4 | 2 |
| 19 = 15+4 | 6 | 4 | 19 = 10+6+3 | 5 | 4 | 3 |
| 20 = 15+5 | 6 | 5 | 20 = 20+0+0 | 6 | 1 | 0 |
| 21 = 21+0 | 7 | 0 | 21 = 20+1+0 | 6 | 2 | 0 |
| 22 = 21+1 | 7 | 1 | 22 = 20+1+1 | 6 | 2 | 1 |

FIG. 3.5. 2-Binomial and 3-binomial counting.

tion, 15 is the sum of 15 and 0, or $\binom{6}{2}$ and $\binom{0}{1}$. In the 3-binomial representation, 15 is the sum of 10, 3, and 2, or $\binom{5}{3}$, $\binom{3}{2}$, $\binom{2}{1}$.

With the example of Fig. 3.5 as background, we can now describe $k$-binomial counting more precisely. We will use an array $D[1 \cdots k]$ to store the upper parts of the binomial coefficients. The invariant of this counting scheme has two parts: first, the represented integer is given by

$$N = \binom{D[k]}{k} + \binom{D[k-1]}{k-1} + \cdots + \binom{D[1]}{1},$$

and second, each coefficient $D[i]$ satisfies the condition

$$D[i] > D[i-1],$$

for $2 \le i \le k$. We can initialize the array to represent zero by assigning each $D[i]$ to have the value $i - 1$; we will also find it handy to assume that the value of $D[k+1]$ is "infinity". The code for incrementing an integer by one is as follows.

```
$D[1] \leftarrow D[1] + 1$
$i \leftarrow 1$
while $D[i] = D[i+1]$ do
    $D[i+1] \leftarrow D[i+1] + 1$
    $D[i] \leftarrow i - 1$
    $i \leftarrow i + 1$
```

It is easy to prove by induction that this code correctly implements the above counting scheme.

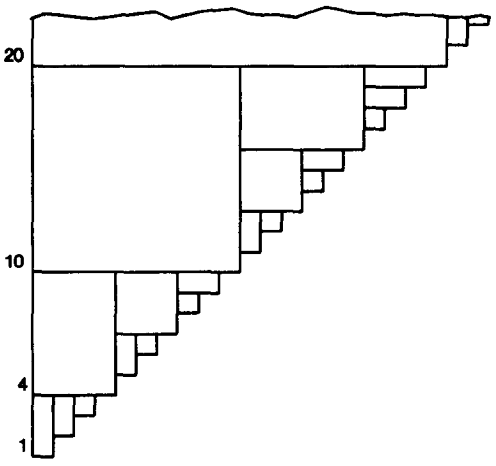

Fig. 3.7. The 3-binomial transform.

It is straightforward to modify the above counting scheme to yield an admissible transform. To do so we will retain the array $D$ (with the same invariant as above), and add an array $P[1 \cdots k]$ of static structures. The number of elements in the static structure $P[i]$ is always $\binom{D[i]}{i}$. The code for this $k$-binomial transform is given in Program 3.6, and Fig. 3.7 illustrates the 3-binomial transform.

```
proc Init$_D$ ←
    for $i \leftarrow 1$ to $k$ do
        $D[i] \leftarrow i - 1$; $P[i] \leftarrow \varnothing$
    $D[k + 1] \leftarrow \infty$

proc Insert$_D(x)$ ←
    $D[1] \leftarrow D[1] + 1$; $S \leftarrow$ Unbuild$_S(P[1]) \cup \{x\}$; $P[1] \leftarrow \varnothing$
    $i \leftarrow 1$
    while $D[i] = D[i + 1]$ do
        $D[i + 1] \leftarrow D[i + 1] + 1$; $S \leftarrow S \cup$ Unbuild$_S(P[i])$
        $D[i] \leftarrow i - 1$; $P[i] \leftarrow \varnothing$
        $i \leftarrow i + 1$
    $P[i] \leftarrow$ Build$_S(S)$

func Query$_D(x)$ ←
    $A \leftarrow$ Query$_S(x, P[1])$
    for $i \leftarrow 2$ to $k$ do
        $A \leftarrow \square(A,$ Query$_S(x, P[i]))$
    return $A$
```

PROGRAM 3.6. Code for the $k$-binomial transform.

The correctness of the code can be proved by induction, and its analysis establishes the following theorem.

THEOREM 3.3 (the $k$-binomial transform). *The k-binomial transform is an admissible* $(k, (k!N)^{1/k})$ *transform.*

*Proof.* Since at most $k$ structures exist at any one time, we have

$$Q_D(N) \leq Q_S(N) \cdot k.$$

Since the space requirement for the static structure grows at least linearly with the number of elements, the dynamic structure can be no more expensive. To bound the processing time of the dynamic structure, we will investigate the maximum number of structures into which any element may be built during the first $N$ insertions. Note that after $N$ insertions, we have

$$N \geq \binom{D[k]}{k}$$
$$\geq (D[k] - k + 1)^k / k!,$$

implying

$$D[k] \leq (k!N)^{1/k} + k - 1.$$

This, together with the invariant that

$$D[k] > D[k-1] > \cdots > D[1] \geq 1$$

implies that each $D[i]$ satisfies

$$0 \leq D[i] - i \leq (k!N)^{1/k} - 1$$

for $1 \leq i \leq k$. Finally, we note that whenever a structure is discarded and its elements are rebuilt into a new structure, the difference between the upper and lower parts of the binomial coefficient giving the size of the structure increases by one; that is, a structure of size

$$\binom{m}{i}$$

is always replaced by a structure of size

$$\binom{m+1}{i}$$

or of size

$$\binom{m+2}{i+1}.$$

This implies that no element is ever built into more than $(k!N)^{1/k}$ static structures, from which it follows that

$$P_D(N) \leq P_S(N) \cdot (k!N)^{1/k}.$$

Q.E.D.

Note that for all positive $k, k!^{1/k} < k$. For large $k$, Stirling's approximation gives[7] $k!^{1/k} \sim k/e$.

[7]We use the notation "$A \sim B$" as a shorthand for "$|A - B| = o(B)$."

To illustrate the application of the binomial transforms, we will consider the problem of range searching. In this problem, the stored set contains points in a $d$-dimensional space; that is, each element in the set has the $d$ attributes $A_1, A_2, \ldots, A_d$. A query asks for all points with each dimension $A_i$ in a specified range $[L_i, U_i]$, for $1 \le i \le d$. (Note that this problem is decomposable with the $\square$ operator interpreted as $\cup$[8].) Bentley and Maurer (1980) describe a structure for static range searching (SRS) with performances

$$\begin{aligned} Q_{\mathrm{SRS}}(N) &= O(\lg N), \\ P_{\mathrm{SRS}}(N) &= O(N^{1+\delta}), \\ S_{\mathrm{SRS}}(N) &= O(N^{1+\delta}) \end{aligned}$$

for any fixed $\delta > 0$. By choosing, for example, $k = \lceil 2/\epsilon \rceil$ and $\delta = \epsilon/2$, we can apply the $k$-binomial transform to achieve the following structure.

New Data Structure 3 (dynamic range searching). A dynamic range searching (DRS) structure supporting insertions and queries for point sets in $d$-space with performance

$$\begin{aligned} Q_{\mathrm{DRS}}(N) &= O(\lg N), \\ P_{\mathrm{DRS}}(N) &= O(N^{1+\epsilon}), \\ S_{\mathrm{DRS}}(N) &= O(N^{1+\epsilon}) \end{aligned}$$

can be achieved for any fixed $\epsilon > 0$ and positive integer $d$.

Such a structure is useful for range searching in a situation in which the number of queries is known greatly to exceed the number of insertions. Specifically, if the number of insertions in a set of $N$ insertions and queries were known to be $\theta(N^p)$ for some $p < 1$, then this structure would allow the operations to be processed in $\theta(N \lg N)$ time. The best performance for this task prior to this structure was (independently) achieved by Lueker (1978) and by Willard (1978); their structures require $\theta(N \lg^d N)$ time.

It is important to observe that the penalties incurred by the $k$-binomial transform need not always be paid. Just as in the binomial transform, they can occasionally be avoided by merging static structures, by counting the expected query cost, or by performing separate analyses for fast-growing functions.

[8]In order to implement (multiset) union as a constant-time operation, we ask that a query return a tree whose leaves are the points within the specified range. Two such trees can be combined in constant time by allocating a new root node containing pointers to the two trees.

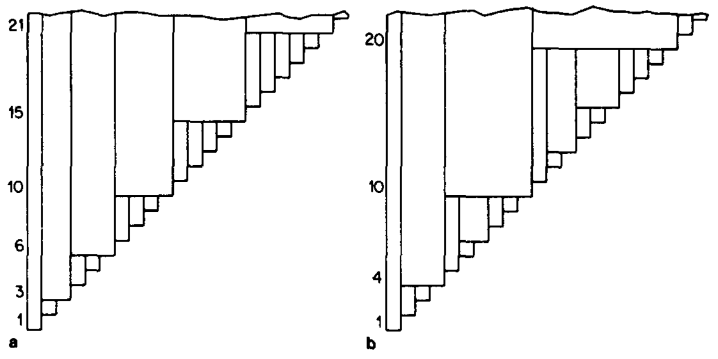

FIG. 3.8. Dual binomial transforms. (a) The dual triangular transform. (b) The dual 3-binomial transform.

## 3.3 *Transformations with Fast Insertion Time*

In the last subsection we investigated a set of transforms that only slightly increase the query time at the cost of greatly increasing the processing time. In this subsection we will study a class of structures *dual* to those, which only slightly increase the processing time but greatly increase the query time. Specifically, we will see that there exists an admissible $\left(k(k!N)^{1/k}, k\right)$ transform for each positive integer $k$. As before, we will first investigate the case $k = 2$, and then turn to the general case.

The *dual triangular transform* is illustrated pictorially in Fig. 3.8a. At time 9, there are six structures (of sizes 1, 2, 3, 1, 1, and 1); when the 10th element is inserted it is combined with the last three structures to create a new static structure of size 4. In general, when the $\binom{M}{2}$th element is inserted, $M$ elements are combined to form a static structure of size $M$; other elements are kept in singleton structures as they are inserted. Since each element is built into only two static structures (the large and the singleton), we know that

$$P_D(N) \leq 2P_S(N).$$

It is easy to show that at most $2(2N)^{1/2}$ static structures exist at any time, so we have

$$Q_D(N) \leq Q_S(N) \cdot 2(2N)^{1/2}.$$

These facts together establish the following theorem.

THEOREM 3.4 (the dual triangular transform). *The dual triangular transform is an admissible* $(2(2N)^{1/2}, 2)$ *transform.*

That this transform is dual to the triangular transform studied in Subsection 3.2 is intuitively clear from Fig. 3.8a. To make the duality more precise we will study the dual triangular transform from the viewpoint of the triangular-number counting scheme of the last subsection. The history

| Structures | | Number | |
|---|---|---|---|
| Large | Small | Large | Small |
| ( ) | ( ) | 0 | 0 |
| (1) | ( ) | 1 | 0 |
| (1) | (1) | 1 | 1 |
| (1,2) | ( ) | 3 | 0 |
| (1,2) | (1) | 3 | 1 |
| (1,2) | (1,1) | 3 | 2 |
| (1,2,3) | ( ) | 6 | 0 |
| (1,2,3) | (1) | 6 | 1 |
| (1,2,3) | (1,1) | 6 | 2 |
| (1,2,3) | (1,1,1) | 6 | 3 |
| (1,2,3,4) | ( ) | 10 | 0 |
| (1,2,3,4) | (1) | 10 | 1 |
| (1,2,3,4) | (1,1) | 10 | 2 |

FIG. 3.9. History of the dual triangular transform.

of the dynamic structure is shown in tabular form in Fig. 3.9. The eighth row shows that when eight elements are in the dynamic structure, there are five static structures: three "large" structures (of sizes 1, 2, and 3) and two "small" structures (each of only one element). In general, if the number in the "large" column is $\binom{M}{2}$, then there are large structures of sizes $1, 2, 3, \ldots, M - 1$. The number in the "small" column gives the number of unit-sized static structures. Note that the entries in the number columns are identical to the 2-binomial counting depicted in Fig. 3.5. This duality carries through to the $k$-binomial transform. For the case of the dual 3-binomial transform, each element will be built into at most three static structures (which we call small, medium, and large). All small structures have exactly one element, medium structures have an integer number of elements, and large structures contain a triangular number of elements. At any point in the history of the transform, each set of existing small, medium, and large structures contains structures of adjacent sizes. The following table shows the history of the dual 3-binomial transform from the insertion of the fourth through the tenth elements; a history diagram of the dual 3-binomial transform appears in Fig. 3.8b.

| | Structures | | | Populations | | |
|---|---|---|---|---|---|---|
| $N$ | Large | Med | Small | Large | Med | Small |
| 4 | (1,3) | ( ) | ( ) | 4 | 0 | 0 |
| 5 | (1,3) | (1) | ( ) | 4 | 1 | 0 |
| 6 | (1,3) | (1) | (1) | 4 | 1 | 1 |
| 7 | (1,3) | (1,2) | ( ) | 4 | 3 | 0 |
| 8 | (1,3) | (1,2) | (1) | 4 | 3 | 1 |
| 9 | (1,3) | (1,2) | (1,1) | 4 | 3 | 2 |
| 10 | (1,3,6) | ( ) | ( ) | 10 | 0 | 0 |

The extension of this strategy from the dual 3-binomial transform to the dual $k$-binomial transform is straightforward. The code of Program 3.6 is modified so that instead of containing a static structure of $\binom{D[i]}{i}$ elements, $P[i]$ now contains a list of structures of sizes

$$\binom{D[i]-1}{i-1}, \binom{D[i]-2}{i-1}, \ldots, \binom{i-1}{i-1}.$$

Note that the sum of the sizes of the structures is $\binom{D[i]}{i}$. This allows us to establish the following theorem.

THEOREM 3.5 (the dual $k$-binomial transform). *The dual k-binomial transform is an admissible* $\left(k(k!N)^{1/k}, k\right)$ *transform.*

*Proof.* Because each element is built into at most $k$ static structures, it is clear that the processing cost increases by at most a factor of $k$. The analysis used in the proof of Theorem 3.3 shows that each of the $k$ classes of structures contains at most $(k!N)^{1/k}$ distinct structures at any point. Therefore at most $k(k!N)^{1/k}$ static structures exist at any time, providing the upper bound on the query time penalty. Q.E.D.

To illustrate the application of this transformation we will again consider the problem of range searching in a $d$-dimensional point set. Bentley and Maurer (1980) describe a second structure for range searching (which we will call SRS′) with properties

$$\begin{aligned} Q_{\mathrm{SRS'}}(N) &= O(N^{\delta}),\\ P_{\mathrm{SRS'}}(N) &= O(N \lg N),\\ S_{\mathrm{SRS'}}(N) &= O(N), \end{aligned}$$

for any fixed $\delta > 0$. By choosing, for example, $k = \lceil 2/\epsilon \rceil$ and $\delta = \epsilon/2$, we can apply the dual $k$-binomial transform to achieve the following structure.

NEW DATA STRUCTURE 4 (dual dynamic range searching). A dynamic range searching (DRS′) structure supporting insertions and queries for point sets in $d$-space with performance

$$\begin{aligned} Q_{\mathrm{DRS'}}(N) &= O(N^{\epsilon}),\\ P_{\mathrm{DRS'}}(N) &= O(N \lg N),\\ S_{\mathrm{DRS'}}(N) &= O(N) \end{aligned}$$

can be achieved for any fixed $\epsilon > 0$ and positive integer $d$.

Note that this structure is appropriate when there are many more insertions than queries; it reduces the cost of the computation of certain sequences of $N$ insert and query operations (analogous to those discussed at the end of Subsection 3.2) from the $O(N \lg^d N)$ time required by Lueker's (1978) or Willard's (1978) methods to $O(N \lg N)$.

TABLE 3.10
Summary of Transformations

| Transformation | Query factor | Processing factor |
|---|---|---|
| $k$-Binomial | $k$ | $(k!N)^{1/k}$ |
| Binary | $\lg(N+1)$ | $\lg(N+1)$ |
| Dual $k$-binomial | $k(k!N)^{1/k}$ | $k$ |

## 3.4. *Summary of the Transformations*

In this section we have seen a number of different static-to-dynamic transformations on data structures for decomposable searching problems. We will now spend a moment reviewing these transformations. The transformations themselves are summarized in Table 3.10.

There are many other transformations besides those that we have already investigated. A simple way of achieving a new transformation is by isomorphism to a particular number system (counting scheme). This is illustrated in Fig. 3.11 for the radix-3 number system (ternary counting). Part (a) of that figure shows the ternary transform: each static structure has cardinality of either a power of 3 or twice a power of 3 and corresponds to either a 1 or a 2 in the ternary representation of the number of elements in the dynamic structure. This transform is an admissible $(\lceil \log_3 N \rceil, 2\lceil \log_3 N \rceil)$ transform.[9] Its dual is shown in part (b) of the figure; every structure in the dual is of size a power of 3, and there are zero, one, or two structures for any power of 3, corresponding to the appropriate digit in the ternary expression of the integer size of the structure. This is an admissible $(2\lceil \log_3 N \rceil, \lceil \log_3 N \rceil)$ transform. This scheme can be extended to radix-$k$ counting to yield a primary $(\lceil \log_k N \rceil, (k-1)\lceil \log_k N \rceil)$ transform and a dual $((k-1)\lceil \log_k N \rceil,$ $[\log_k N])$ transform. An interesting open problem is to examine other counting schemes (such as Fibonacci or factorial counting) for their properties as transforms; in Section 4 we will see techniques that enable us to establish lower bounds on the cost of transforms and thereby give us a touchstone for evaluating various derived transforms.

It is now easy to state formally the relationship of the primary and dual transforms derived from a particular counting scheme. In the primary transform, there is a single structure corresponding to each digit, whereas in the dual transform each digit corresponds to a set of structures that are the "carries" from its right neighbor (the unit digit is a set of structures of size 1).

The transformations of this section together provide a powerful set of tools for designing new data structures both for particular applications and

[9]This and the following claims about radix-$k$ transforms assume that $N > 1$.

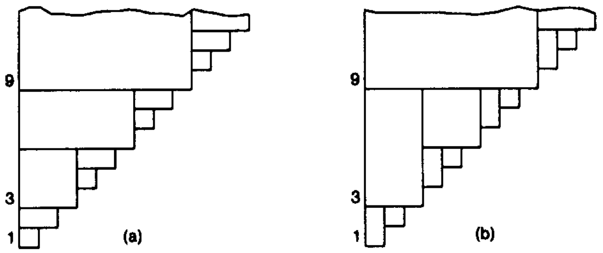

FIG. 3.11. Radix-3 transformations. (a) The ternary transform. (b) The dual ternary transform.

as a component in larger algorithms. To design a dynamic structure in a given context, the algorithm designer first designs a static structure (which is usually much easier than designing a dynamic structure), and then applies one of the transformations to achieve an efficient dynamic structure. Which transformation is used depends on the relative efficiency of the static preprocessing and query costs and on the expected frequency of insertions and queries.

As we mentioned before, the cost penalties of the transformations need not always be paid. One can often avoid them by merging static structures, by analyzing the average query time, or by performing separate analyses for fast-growing cost functions.

## 4. Lower Bounds on Transformations

Our main goal in this section is to prove the optimality, in a certain sense, of some of the transformations discussed in Section 3. Our path to this goal will have many steps, and the reasons for each step might not be clear in advance. To aid the reader, we now briefly sketch the contents of this section.

In Subsection 4.1 we define the model of computation which we will use throughout the rest of the section. We also advise the reader that the use of this model implies certain limitations on the applicability of the results we will obtain. In Subsections 4.2 through 4.4 we show a method for representing an initial sequence of insertions under some transform as a binary tree, and show how the efficiencies of transformations are related to properties of the corresponding trees. To achieve the correspondence between transforms and trees, we restrict our attention to a class of transforms that we call the *arboreal* transforms. In Subsection 4.5 we state and solve a recurrence relating the various tree properties defined in Subsection 4.4, and interpret this result as it applies to the $k$-binomial transformations. We then extend the basic result to answer questions about other transformations (including the binary transformation) in Subsection

4.6. In Subsection 4.7 we discuss the justification of the restriction to arboreal strategies, and in Subsection 4.8 we return to explore the limitations (implied by our model) of the preceding results, showing a number of cases in which our "lower bounds" can be beaten by going outside the model.

## 4.1. *The Model of Computation*

The most important assumption of our model is that the transformations under consideration are not allowed to use any specific knowledge about the original problem or static structure except for the fact that the problem is decomposable. It therefore remains plausible for any particular decomposable searching problem, $P$, that there exists a dynamic data structure for $P$ having performance better than that produced by applying any optimal static-to-dynamic transform to any static structure for $P$. For example, AVL trees (see Knuth (1973)) provide a dynamic data structure for member searching with

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

The results of this section imply that no dynamic structure with this efficiency can be obtained (in the worst case) by applying a general transform to a static structure for member searching; the efficiency of AVL trees depends on particular properties of the member searching problem other than decomposability (in particular, the ability to maintain the structural invariant under rotation).

Our model of computation is that we have three operations, Build, Query, and $\square$, whose inner workings we may not examine. Build works with performance $P_S$ to create static structures. Query works with performance $Q_S$ to search the structures created by Build. The $\square$ operator is guaranteed to have the property

$$\square(\mathrm{Query}(x, \mathrm{Build}(A)), \mathrm{Query}(x, \mathrm{Build}(B)) = \mathrm{Query}(\mathrm{Build}(A \cup B)).$$

The only way to answer a query is by applying Query one or more times to structures created by Build and then combining the results using $\square$. We assume that $P_S$ grows at least linearly and that $Q_S$ is monotone nondecreasing.

To measure the computation costs ($P_D$ and $Q_D$) associated with a dynamic structure, we will charge only for the computation time of calls to Build and Query. It should be noted that these costs will generally be the dominant parts of the total costs of the dynamic algorithms. In any case, this approximation is certainly acceptable for the purpose of establishing *lower* bounds on the costs of dynamic algorithms.

Our goal in the search for efficient transformations is to minimize simultaneously the penalty functions

$$F(N) = \operatorname*{Max}_{1 \le i \le N} Q_D(i)/Q_S(i) \qquad \text{and} \qquad G(N) = P_D(N)/P_S(N).$$

The bulk of this section will be devoted to showing limits on just how far this process may be carried in the worst case. Our interpretation of the term "worst case" in this context is a bit tricky. We have already mentioned that we may assume no specific knowledge about the problem or the original static structure except for decomposability. It is also important to note that we do not allow ourselves to assume any specific knowledge about the *efficiency* of the underlying static structure, except that $P$ is at least linear and $Q$ is monotone nondecreasing. (Note, for example, that the improvements in $F$ and $G$ which occur for fast-growing $P$ and $Q$ are not examples of worst-case behavior, so there is no contradiction in the fact that our lower bounds deny the possibility of such improvements in the general case.)

The reader may find it helpful to think of the worst case as that in which $P$ is linear and $Q$ is constant, the intuition being that it is hardest for the dynamic structure's costs to approach the static structure's costs when the latter are as small as possible. Since we may not use any specific knowledge about the original static problem or data structure, any solution to the dynamic problem must work by maintaining a collection of static structures. Whenever an element is inserted, a new structure must be created containing that element[10] and possibly some other elements. Also, some existing static structures may be thrown away. When a query is made to the dynamic structure, it is necessary to search some set of static structures which together contain all the elements inserted so far.

For the following analysis, we will place a few restrictions on the nature of the dynamic structures we will consider. We will return later to the problem of justifying these restrictions. Our first restriction is as follows:

RESTRICTION 4.1 (dynamic structures partition elements into static structures). We assume that at any time there exists exactly one static structure containing each element which has been inserted so far. That is, the static structures partition the set of elements represented by the dynamic structure.

With the preceding assumptions in mind, we are now ready to move on to the first steps of our analysis.

[10] While we may conceive of strategies in which new static structures are created by queries into the dynamic structure, we need not consider this possibility for this worst-case analysis, since $P_S$ could grow much more rapidly than $Q_S$.

## 4.2. *Computing F and G*

We now give some rules for determining the worst-case values of the penalty functions $F$ and $G$ associated with a particular strategy.

DEFINITIONS ($f$ and $g$). Consider the history of a dynamic structure over the course of any number of insertions starting when the structure is empty. We define $f(N)$ as the maximum number of static structures existing after one of the first $N$ insertions. We define $g(N)$ as the sum of the cardinalities of all sets of elements built into static structures created over the course of the first $N$ insertions.

Note that, while the definitions of $f$ and $g$ actually depend on the specific transform used, the identity of the transforms under consideration will always be clear from context. We may now bound $F$ and $G$ as follows:

THEOREM 4.1 ($f$ bounds $F$). *For any positive integer $N$, $F(N) \leq f(N)$.*

*Proof.* After any of the first $N$ insertions (say the $i$th), at most $f(N)$ static structures exist. To compute the cost of answering a query, we charge precisely for querying these structures. Since each of these structures has cardinality no larger than $i$, and since $Q_S$ is monotone nondecreasing, the total cost is at most $f(N)Q_S(i)$. Q.E.D.

THEOREM 4.2 ($g/N$ bounds $G$). *For every positive integer, $N$, $G(N) \leq g(N)/N$.*

*Proof.* We note that any static structure built during the first $N$ insertions will have cardinality no larger than $N$. Consider such a structure, $S$, having cardinality $i$. By the fact that $P_S$ grows at least linearly, we may bound the cost of building $S$ by the inequality

$$P_S(i) \leq iP_S(N)/N.$$

Summing over all static structures, we get

$$P_D(N) \leq g(N)P_S(N)/N,$$

implying

$$G(N) = P_D(N)/P_S(N) \leq g(N)/N. \qquad \text{Q.E.D.}$$

By the assumptions in Subsection 4.1, the preceding bounds are the tightest possible for the general case. We will therefore concern ourselves henceforth with the problem of minimizing $f$ and $g$ rather than $F$ and $G$.

## 4.3. *Transforming History Diagrams to Trees*

The transforms we discussed in Section 3 are all representable by history diagrams, such as those in Figs. 3.1, 3.4, 3.7, 3.8, and 3.11. It is not the case, however, that all transforms are so representable; in order for a static structure to be represented as a (contiguous) rectangle in a history diagram, it is necessary that it be built from a set of elements which were inserted consecutively during the history of the structure. We now impose our second restriction on the class of dynamic structures to be considered:

RESTRICTION 4.2 (contiguity of static structures). We will restrict our attention to transforms whose histories are representable by history diagrams.

Indeed, we will further restrict our attention to history diagrams (such as those in Section 3) in which every rectangle reaches to the "diagonal" of the diagram. We may state this otherwise as

RESTRICTION 4.3 (eagerness of static structures). We will restrict our attention to transforms in which each static structure is built as soon as all its elements have been inserted, and in which the elements of any discarded static structure are always built into a single new static structure (along with some additional elements).

Strategies which satisfy Restrictions 4.1, 4.2, and 4.3 will be called *arboreal* strategies for a reason that will soon become obvious.

Consider the history diagram for the first $N$ insertions into a dynamic structure which is maintained by an arboreal strategy. Any such diagram induces a binary tree, as shown in Fig. 4.1. We may draw this tree by tracing the left and upper edges of each rectangle in the diagram. The internal nodes of the tree will thus be at the upper left corners of the various rectangles; each internal node of the tree corresponds to a (unique) static structure. We will now go on to study some relationships between the efficiencies of arboreal strategies and properties of their induced trees.

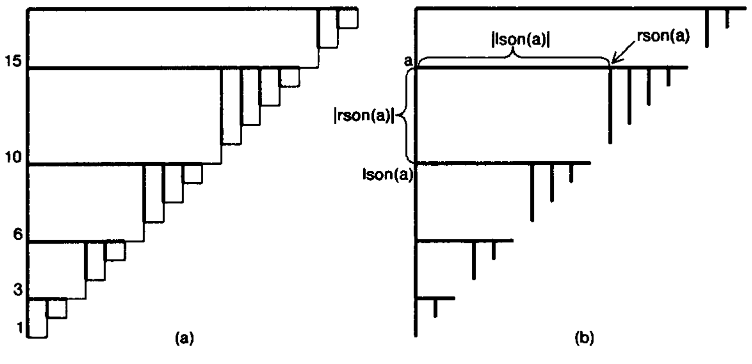

FIG. 4.1. A history diagram and its induced tree. (a) A partial history diagram. (b) The induced tree.

## 4.4 *Tree Properties and Their Relation to Performance*

We now introduce some basic vocabulary for discussing properties of binary trees.

DEFINITIONS (tree properties). Let $T$ be a binary tree. Then leaves($T$) denotes the set of all leaves of $T$ and nodes($T$) denotes the set of all internal nodes of $T$. The *weight* of $T$, denoted $|T|$, is defined as the cardinality of leaves($T$). For any internal node, $a$, of $T$ the left and right sons of $a$ are denoted lson($a$) and rson($a$), respectively. If $a$ is a leaf of $T$, then the *right depth* of $a$, written rd($a$), is defined as the number of right branches along the path from the root of $T$ to $a$. The *right height* of $T$, rh($T$), is the maximum right depth of any leaf of $T$. The *right path length* of $T$, R($T$), is defined as the sum of the right depths of all leaves of $T$. Left depth, left height, and left path length are defined analogously.

We will sometimes identify a (not necessarily internal) node, $x$, of a tree with the subtree rooted at $x$. For example, we may write $|x|$ to indicate the number of leaves which are descendants of $x$.

We now make the following observation:

THEOREM 4.3 (alternate characterization of left path length). *Let $T$ be a tree. Then,*

$$\mathrm{L}(T) = \sum_{n \in \mathrm{nodes}(T)} |\mathrm{lson}(n)|.$$

*Proof.* Consider any leaf, $x$, of $T$. We need only note that the left branches along the path from the root of $T$ to $x$ emanate precisely from those nodes of $T$ whose left sons contain $x$. Q.E.D.

With this characterization of left path length in mind, we may now relate the trees induced by arboreal strategies to the penalty functions associated with those strategies.

Consider the tree in Figure 4.1b. To each static structure created during the partial history represented by that tree, there corresponds a right (horizontal in the diagram) branch whose length (in the diagram) is proportional to the cardinality of that static structure. Moreover, for any internal node, $n$, of the tree, the length (in the diagram) of the right branch from $n$ corresponds precisely to the number of leaves in the *left* son of $n$. By summing over all internal nodes of the tree, we establish the following result:

THEOREM 4.4 (relation of $g$ to left path length). *Let $N$ be a positive integer and let $T$ be the tree induced from the history diagram representing the* *first $N$ insertions into a dynamic structure maintained by some arboreal strategy. Then,* $\mathrm{L}(T) = g(N)$.

*Proof.* Given in the preceding text. Q.E.D.

We may also characterize $N$ and $f$ in terms of tree properties:

THEOREM 4.5 (relation of $N$ and $f$ to tree properties). *Let $N$ be a positive integer and let $T$ be the tree induced from the history diagram representing the first $N$ insertions into a dynamic structure maintained by some arboreal strategy. Then,*

$$|T| = N + 1 \qquad \textit{and} \qquad \mathrm{rh}(T) = f(N).$$

*Proof.* Inspection of Fig. 4.1 will reveal that these results are obvious. Q.E.D.

In the remainder of this section we will use $N$ to denote the number of elements inserted into a dynamic structure under some arboreal strategy, and $n = N + 1$ to indicate the number of leaves in the corresponding tree.

The theorems proved so far in this section allow use to address the problem of "simultaneously minimizing" $F$ and $G$ by investigating a closely related problem about trees, namely, that of "simultaneously minimizing" the right height and left path length of a tree with a fixed number of nodes. To discuss this more precisely, we make the following definition:

DEFINITION (minimal left path length). Let $n$ and $k$ be positive integers. We define

$$L_k(n) = \mathrm{Min}\{\mathrm{L}(T) | T \text{ is a tree such that } |T| = n \text{ and } \mathrm{rh}(T) \leq k\}.$$

Since the only tree with zero right height is the tree of one node (which also has zero left path length), we also define

$$L_0(1) = 0.$$

By convention, we will regard $L_0(n)$ as "positive infinity" whenever $n > 1$. A tree with $n$ leaves, right path length $k$, and left path length $L_k(n)$ will be called an *economical* tree.

In the next subsection, we will investigate the behavior of $L_k(n)$ as $k$ and $n$ vary, and then restate our findings in terms of lower bounds on worst-case penalty functions.

## 4.5 *The Behavior of $L_k(n)$*

Consider a binary tree, $T$, with root node $t$. Let $A$ and $B$ be the subtrees rooted at $a = \mathrm{lson}(t)$ and $b = \mathrm{rson}(t)$, respectively. The weight, right height, and left path length of $T$ may be recursively computed from properties of $A$ and $B$ by the relations

$$\begin{aligned} |T| &= |A| + |B|, \\ \mathrm{rh}(T) &= \max(\mathrm{rh}(A), \mathrm{rh}(B) + 1), \\ \mathrm{L}(T) &= \mathrm{L}(A) + |A| + \mathrm{L}(B). \end{aligned}$$

From this, we obtain the following recurrence for $L_k(n)$:

THEOREM 4.6 (recurrence for $L_k(n)$). *Let $n$ and $k$ be any positive integers. Then,*

$$L_k(n) = \begin{cases} 0, & n = 1, \\ n - 1 + L_k(n-1) = \binom{n}{2}, & k = 1, n > 1, \\ \min\limits_{1 \le i \le n-1} \left[ L_k(i) + i + L_{k-1}(n-i) \right], & k > 1, n > 1. \end{cases}$$

*Proof.* The results for $k = 1$ follow by considering the unique binary tree of any weight which has right height $\le 1$. For the case $k > 1$, we consider a tree $T$ having weight $n > 1$ and height $k$. Let $t$ be the root of $T$, and let $A$ and $B$ be the subtrees rooted at $a = \mathrm{lson}(t)$ and $b = \mathrm{rson}(t)$, respectively. Then we must have

$$\begin{aligned} 1 \le |A| &< n, \\ |A| + |B| &= n, \\ \mathrm{rh}(A) &\le k, \\ \mathrm{rh}(B) &\le k - 1. \end{aligned}$$

Moreover, if the left path length of $T$ is to be minimal, then the left path lengths of $A$ and $B$ must be minimal. That is, we must have

$$\mathrm{L}(A) = L_k(|A|) \qquad \text{and} \qquad \mathrm{L}(B) = L_{k-1}(|B|).$$

These requirements are precisely captured by our recurrence. Q.E.D.

We now come to the principal theorem of this section, wherein the behavior of $L_k(n)$ is precisely characterized in terms of binomial coefficients.

THEOREM 4.7 (characterization of $L_k(n)$). *Let $k$ and $m$ be nonnegative integers such that $k \le m$, and let $n$ be a positive integer satisfying*

$$\binom{m}{k} \le n \le \binom{m+1}{k}.$$

*Then,*

$$L_k(n) = k\binom{m}{k+1} + (m - k + 1)N, \tag{I}$$

*where*

$$N = n - \binom{m}{k}.$$

*Proof.* Our proof will proceed by induction on $k$ and, for each fixed positive value of $k$, by induction on $n$.

*Base step* ($k = 0$). In this case, we have

$$\binom{m}{k} = 1 = \binom{m+1}{k}.$$

This implies that $n = 1$, so the right-hand side of (I) reduces to

$$\begin{aligned}(0)\binom{m}{0+1} + (m - 0 + 1)\left(n - \binom{m}{k}\right) &= 0 + (m + 1)(1 - 1).\\ &= 0\\ &= L_0(1).\end{aligned}$$

*Inductive step* ($k > 0$). We now must show that the theorem holds for any $k > 0$ assuming it holds for all smaller $k$. We proceed by induction on $n$. In doing this, we must take note of the interaction between $m$ and $n$. Since $k$ is positive, $\binom{m}{k}$ increases monotonically with $m$. Thus, the minimum possible value of $n$ is $\binom{k}{k} = 1$, and for any positive value of $n$, there is at least one possible value for $m$ (and occasionally there will be two).

*Base step* ($n = 1$). In this case we may choose either

(a) $m = k - 1$; $N = 1 - \binom{k-1}{k} = 1 - 0 = 1$, or

(b) $m = k$; $N = 1 - \binom{k}{k} = 1 - 1 = 0$.

We must show that (I) hold for either choice of $m$. For the case $m = k - 1$, we have

$$\begin{aligned}k\binom{m}{k+1} + (m - k + 1)N &= k\binom{k-1}{k+1} + ((k - 1) - k + 1)N\\ &= k(0) + (0)1\\ &= 0\\ &= L_k(1).\end{aligned}$$

For the case $m = k$, we have

$$\begin{aligned} k\binom{m}{k+1} + (m-k+1)N &= k\binom{k}{k+1} + (k-k+1)N \\ &= k(0) + (1)0 \\ &= 0 \\ &= L_k(1). \end{aligned}$$

*Inductive step* ($n > 1$). We first show that the right-hand side of (I) gives an upper bound on $L_k(n)$. Note that

$$\binom{m-1}{k} + \binom{m-1}{k-1} = \binom{m}{k} \le n \le \binom{m+1}{k} = \binom{m}{k} + \binom{m}{k-1}.$$

We now pick $a$ and $b$ such that

$$\begin{aligned} &\binom{m-1}{k} \le a \le \binom{m}{k}, \\ &\binom{m-1}{k-1} \le b \le \binom{m}{k-1}, \qquad\qquad \text{(II)} \\ &a + b = n. \end{aligned}$$

By Theorem 4.6, we have

$$\begin{aligned} L_k(n) &\le L_k(a) + a + L_{k-1}(b) \\ &= k\binom{m-1}{k+1} + ((m-1)-k+1)(A) \\ &\quad + \binom{m-1}{k} + A \\ &\quad + (k-1)\binom{m-1}{k} + ((m-1)-(k-1)+1)(B) \\ &= k\left[\binom{m-1}{k+1} + \binom{m-1}{k}\right] + (m-k+1)(A+B) \\ &= k\binom{m}{k+1} + (m-k+1)N. \end{aligned}$$

where

$$\begin{aligned} A &= a - \binom{m-1}{k}, \\ B &= b - \binom{m-1}{k-1}, \\ N &= n - \binom{m}{k}. \end{aligned}$$

This establishes that our expression is an upper bound on $L_k(n)$. To establish that this is also a lower bound, we must show that no other way of expressing $n$ as the sum of two positive numbers, $a$ and $b$, will give a smaller value for

$$L_k(a) + a + L_{k-1}(b). \tag{III}$$

To show this, we consider the effect on the value of expression (III) of increasing or decreasing $a$ by steps of 1.[11] Suppose we start with $a$ and $b$ chosen to satisfy (II), and then start incrementing $a$ and decrementing $b$ by steps of 1. So long as $a$ remains less than $\binom{m}{k}$ and $b$ remains greater than $\binom{m-1}{k-1}$, the effect of each increment will be to increase $L_k(a) + a$ by $((m-1) - k + 1) + 1 = m - k + 1$ and to decrease $L_{k-1}(b)$ by $(m-1) - (k-1) + 1 = m - k + 1$, leaving the total value of (III) unchanged.[12] However, as soon as either $a$ or $b$ exceeds the stated bound, one or more of the following things will happen:

1. the incremental growth of $L_k(a)$ will increase while the incremental shrinkage of $L_{k-1}(b)$ decreases or remains the same,

2. the incremental shrinkage of $L_{k-1}(b)$ will decrease while the incremental growth of $L_k(a)$ increases or remains the same, or

3. $b$ will diminish to 0.

In any case, a smaller value for (III) will not be obtained. Similarly, if we start with $a$ and $b$ as in (II) and decrease the value of $a$ while increasing $b$, then we will have zero or more steps at which (III) remains unchanged, zero or more steps where the increase in $L_{k-1}(b)$ exceeds the decrease in $L_k(a) + a$, and finally the step at which $a$ diminishes to zero. Thus, the rules given in (II) give an optimal partitioning of $n$ into $a$ and $b$. This completes the induction step and the proof. Q.E.D.

The use of the auxiliary variable, $m$, in expression (I) makes it a bit difficult to grasp intuitively what is being said about the effects of $n$ and $k$ on $L_k(n)$. To make the picture clearer, we will briefly study the asymptotic behavior of $L_k(n)$ as $k$ remains fixed and $n$ grows without bound. Consider first what happens as $n$ ranges only over binomial coefficients of the form $\binom{m}{k}$. We note that $n = \binom{m}{k}$ implies

$$m - k + 1 \le (n/k!)^{1/k} \le m.$$

[11]In the following, we assume that $k > 1$. If $k = 1$ we must always take $b = 1$ (and $a = n - 1$), since only then is $L_0(b)$ defined.

[12]The incremental changes given here are found by substitution into the second term of the right-hand side of (I), under the induction hypothesis.

So,

$$\begin{aligned} L_k(n) &= k\binom{m}{k+1} \\ &= kn(m-k)/(k+1) \\ &\sim [k/(k+1)]k!^{1/k}n^{1+1/k}. \end{aligned}$$

Since the growth of $L_k(n)$ is very well behaved,[13] the preceding may be extended to cover all values of $n$.

THEOREM 4.8 (asymptotic behavior of $L_k(n)$). *Let $k$ be any positive integer. Then,*

$$L_k(n) \sim [k/(k+1)]k!^{1/k}n^{1+1/k}.$$

*Proof.* The result follows directly from the preceding text. Q.E.D.

By precisely characterizing $L_k(n)$, Theorem 4.7 gives us a bound on the efficiencies of arboreal static-to-dynamic transforms. Any such strategy which has $f(N) \le k$ for all $N$ must always have $g(N) \ge L_k(N+1)$. The asymptotic behavior of $L_k(n)$ given by Theorem 4.8 and our knowledge that Theorems 4.1 and 4.2 are the best possible within our model tell us that whenever we have

$$F(N) < k$$

for any positive integer $k$, we must also have

$$G(N) \ge L_k(N+1)/N \sim (k!N)^{1/k}.$$

This is precisely the behavior achieved by the $k$-binomial transforms, up to lower-order terms. Note, however, that the exact lower bound is not always achievable. The reason for this is the consideration of *immutability of history*. If we know in advance that there will be exactly $N$ insertions, then an optimal strategy can be devised by working backward from an economical tree of weight $N+1$ and right height $k$. If the total number of insertions to be made turns out to be larger, though, then a different strategy for the first $N$ insertions may have been appropriate. Fortunately, the results of this restriction turn out not to be too severe, since the $k$-binomial strategies have efficiency very close to this theoretical limit. The following theorem shows that, for any $k$, the $G(N)$ achieved by the $k$-binomial transform is optimal (for $(F(N) \le k)$ not only to within lower-order terms but actually to within an additive constant of unity.

[13]Given the values where $n$ is of the form "$m$ choose $k$," we can find the *exact* values at all other $n$ by linear interpolation.

THEOREM 4.9 (optimality of $k$-binomial transforms). *For any positive integer, $k$, the $k$-binomial transform achieves*

$$f(N) \leq k \quad \textit{and} \quad g(N) \leq L_k(N+1) + N$$

*for all positive $N$.*

*Proof.* Examination of the optimal construction given in the proof of Theorem 4.7 shows that the $k$-binomial strategy achieves the optimal value of

$$f(N) = L_k(N+1)$$

when $N$ is of the form

$$N = \binom{m}{k} - 1$$

for some $m \geq k$. For intermediate values of $N$, we need only note that, after the first $N$ insertions under the $k$-binomial strategy, the sum of the cardinalities of all structures formed so far except *those still in existence after the Nth insertion* (note that the latter must have a total cardinality of $N$) will never be greater than $L_k(n)$. This fact may be established by induction on $k$, using the fact that values of $L_k(n)$ are given exactly by linear interpolation between points at which the $k$-binomial transform gives absolutely minimal values of $f(N)$. Q.E.D.

## 4.6 *Allowing the Number of Static Structures to Grow*

So far in this section we have only considered minimizing $g(N)$ where $f(N)$ is bounded by a constant. In other words, we have considered only strategies which allow some fixed maximum number of static structures to exist at one time. In Section 3, however, we also investigated strategies (the binary and the dual $k$-binomial transforms) which allow the number of static structures to grow without limit as the total number of elements in the dynamic structure increases. We will now, therefore, briefly investigate transforms which allow $f(n)$ to grow without bound.

To study the efficiency of transforms in which $f(N)$ is unbounded, we may consider the behavior of $L_k(n)$, where $k$ is allowed to vary with $n$.[14] We must be aware of two possible consequences of allowing $k$ to grow:

(1) For any particular $k$, $n$ may never grow large enough for $L_k(n)$ to approach the asymptotic behavior given by Theorem 4.9.

[14] In accordance with the notational conventions of this section, we have $k = f(n) = f(N+1)$, since the first $N$ insertions always give a history diagram which induces a tree of weight $N + 1$.

(2) Our previous caveat about the immutability of history may become more significant.

Since the asymptotic approach of $L_k(n)/[k/(k+1)]k!^{1/k}n^{1+1/k}$ to unity (as $n$ grows and $k$ remains constant) is from below, (1) may be ignored for the purpose of investigating upper bounds. Since the immutability of history can never make it easier to devise efficient transforms, this consideration may be ignored for the investigation of lower bounds. Because of these complicating factors, our results for transforms with unbounded $f$ are less precise than those for bounded $f$. A few results are nonetheless worth noting, the first of which is the following.

THEOREM 4.10a (optimality of the binary transform). *For any arboreal transform such that $f(N) = O(\lg N)$, $g(N) = \Omega(N \lg N)$.*

*Proof.* Since constraining the growth of $f$ can only increase and never decrease the necessary growth of $g$, we need only consider the case where $f(N) = \theta(\lg N)$. We must show that $L_{f(N)}(N+1) = \Omega(N \lg N)$. We define the function $M$ by

$$M(n,k) = \operatorname{Max}\left\{m \,\middle|\, \binom{m}{k} \le n\right\}.$$

From the fact that $f(N) = \theta(\lg N)$, it follows that $M(N,f(N)) - f(N) = \theta(\lg N)$. This gives us

$$\begin{aligned} g(N) &\ge L_{f(N)}(N+1) \\ &\ge L_{f(N)}(N) \\ &\ge f(N)\binom{M(N,f(N))}{f(N)+1} \\ &\sim [f(N)/(f(N)+1)][M(N,f(N)) - f(N)]N \\ &= \theta(N \lg N) = \Omega(N \lg N). \end{aligned}$$

Q.E.D.

This result tells us that the binary transform is optimal in the sense that any transform that pays as small a penalty in search cost (within a constant factor) must pay at least as large a penalty in insertion (again within a constant factor); any arboreal transform which achieves $F(N) = O(\lg N)$ in the worst case must also pay $G(N) = \Omega(\lg N)$.[15] The binary transform is also optimal in the sense that any transform which is actually cheaper (by more than a constant factor) for searches must be strictly

[15]This follows from the fact that Theorems 4.1 and 4.2 are the tightest results possible within our model.

more expensive (again by more than a constant factor) for insertions. We state this result more formally in the following theorem.

THEOREM 4.10b (optimality of the binary transform). *For any arboreal transform such that $f(N) = o(\lg N)$, $g(N) = \omega(N \lg N)$.*

*Proof.* Let the function $h$ be defined by

$$h(N) = (\lg N)/f(N).$$

From the hypothesis that $f(N) = o(\lg N)$, it follows that $h(N) = \omega(1)$. Moreover, since $M(N,f(N)) \geq \lg N$, we have $f(N) = o(M(N,f(N)))$, which means that the approximation in Theorem 4.8 remains valid.[16] This gives us

$$\begin{aligned} g(N) &\geq L_{f(N)}(N+1) \\ &\geq L_{f(N)}(N) \\ &\sim [f(N)/(f(N)+1)]\, f(N)!^{1/f(N)} N^{1+1/f(N)} \\ &\sim [1](f(N)/e) N^{1/f(N)} N \\ &= [(\lg N)/(e \cdot h(N))] 2^{h(N)} N \\ &= \omega(N \lg N). \end{aligned}$$

Q.E.D.

This implies that any arboreal transform which achieves $F(N) = o(\lg N)$ in the worst case must also pay $G(N) = \omega(\lg N)$.

In the preceding proof, we saw that the approximation given in Theorem 4.8 still serves to provide a lower bound on the growth of $g$ even when $f$ is allowed to grow without bound, provided that $f(N) = o(\lg N)$. The next natural question is whether this bound can always be achieved. It turns out that this is not always possible. If $f$ grows in a very irregular manner, having sudden spurts of growth separated by intervals of almost no change, then the immutability of history will cause $g(N)$ to be much larger than $L_{f(N)}(N+1)$ for values of $N$ immediately following the sudden increases. If $f$ grows "smoothly" (the precise meaning of this term is implicit in the following theorem), however, this lower bound for $g(N)$ is very nearly obtainable. We state this result formally as follows.

THEOREM 4.11 (optimizing $g$ for slowly growing $f$). *Let $h$ be a monotone non-decreasing differentiable function such that*

$$h(x) = \omega(1) \quad \textit{and} \quad h'(x) = o(1/x).$$

[16] That is, consideration (1) may be disregarded.

*Then, there exists a transform having*

$$f(N) \leq \lceil h(N) \rceil, \tag{I}$$

$$g(N) \sim (h(N)/e) N^{1+1/h(N)}. \tag{II}$$

*Moreover, given* (I), (II) *is optimal up to lower-order terms.*

*Proof.* A structure having the performance described may be formed by a process of "cutting and pasting" from the history diagrams of the various $k$-binomial strategies. We omit the details for brevity and for the sake of keeping the reader awake. The optimality of (II), given (I), is implicit in the proof of Theorem 4.10b. Q.E.D.

Our results for transforms in which $f(N) = \omega(\lg N)$ are much less complete. In particular, we know that the performance of the dual $k$-binomial transforms falls substantially short of the bound given by the inequality

$$g(N) \geq L_{f(N)}(N+1).$$

We conjecture that this is an inevitable penalty of the immutability of history, and that the dual binomial transforms are in fact optimal in some strong sense, similar to that of Theorem 4.9 for the ordinary binomial transforms. The problem of finding optimal transforms in which $f(N)$ grows faster than $\lg N$ but slower than $n^{\epsilon}$ for any positive $\epsilon$ remains open.[17]

## 4.7 *Justification of the Restriction to Arboreal Transforms*

In Subsections 4.1 and 4.3, we introduced three restrictions which together constrained our investigation to arboreal transforms. While we conjecture that arboreal strategies are optimal, in the sense that for any nonarboreal transform there exists an arboreal transform which is at least as good (given the "black box" model described in Subsection 4.1), we have not yet found a rigorous proof. In this subsection, we will summarize our reasons for considering each of the restrictions reasonable.

Restriction 4.1 forbids the existence of multiple structures containing the same element. Our intuition is that any strategy that permits such overlapping structures can be improved by omitting the shared elements from all but one of the overlapping structures. To justify this intuition would require careful examination of the consequences of this omission when that one structure is finally destroyed. We may also forbid overlapping structures on the grounds that transformations which allow them cannot be

[17]We may equivalently view this as the problem of optimizing $f$ when $g(N)$ grows asymptotically faster than $N$ but slower than $N \lg N$.

optimal for space in the worst case. An even more serious objection is that there are a number of problems that satisfy the definition of decomposability only when the unions involved are of disjoint sets.

Our intuitive justification for Restriction 4.2 (contiguity of static structures) is the belief that a partial history which does not satisfy this restriction can be turned into one that does, at no cost in $f(N)$ or $g(N)$, by a kind of "permutation of the names of the elements." To show this would justify the restriction at least for the cases where $f$ is bounded or grows slowly and smoothly, so that the immutability of history is not a significant problem.

For Restriction 4.3 (eagerness of static structures), we can actually give a rigorous justification, at least over the class of transforms which already satisfy Restrictions 4.1 and 4.2. We express this in the following theorem:

THEOREM 4.12 (optimality of eager strategies). *Let $N$ be a positive integer. For any partial history consisting of the first $N$ insertions and satisfying Restrictions* 4.1 *and* 4.2, *there exists a partial history which also satisfies Restriction* 4.3 *and which has $f(N)$ and $g(N)$ no greater than those for the original partial history.*

*Proof.* Any partial history which satisfies the first two restrictions may be represented by a history diagram. We may ensure that the rectangle in the upper left corner of the diagram represents a structure which is formed as soon as all its elements become available, for any diagram that does not have this property can be transformed at no cost into one that does. The construction is as follows:

> Let $R$ be the upper left rectangle in the diagram. Consider the leftmost rectangle immediately below $R$. If it is wider than $R$, then we extend it upward to the top of the diagram, obliterating $R$; if it is narrower than $R$, then we extend $R$ downward by one step. This process is repeated until the property holds.

But now the rest of the diagram (excluding the upper-left rectangle) must consist of zero, one, or two staircase-shaped pieces to which the same process may be applied recursively, finally yielding a diagram satisfying Restriction 4.3. No step in this process increases either the total preprocessing cost or the maximum number of simultaneously existing structures, so Restriction 4.3 has been formally justified. Q.E.D.

## 4.8 *Limitations on the Significance of the Lower Bounds*

The lower bounds we have derived in this section are based on the model of computation given in Subsection 4.1. Before concluding the section, we will mention some of the limitations which this implies for the applicability of our results.

We have already mentioned that is is often possible to obtain superior dynamic data structures for individual decomposable problems (e.g., Member) by using specific properties of those problems. Another assumption on which our lower bounds depend is that Theorems 4.1 and 4.2 are the strongest possible results of their kind, because we assume no knowledge about the performance of the original static algorithm. As we saw at the end of Subsection 3.1 the penalty factors, $F(N)$ and $G(N)$, may be greatly reduced (from $\theta(\lg N)$ to $\theta(1)$ in the example of Subsection 3.1) if the cost functions of the static structure are already fast-growing. We now present some results concerning a slightly different way of lowering the penalty functions given fast-growing cost functions for the original static structure.

Suppose we are given a static structure for a decomposable searching problem having preprocessing cost $P_S(N)$ and query cost $Q_S(N)$. We will make only the usual assumption about $Q_S$—that it is monotone nondecreasing. We will, however, make the assumption that $P_S(N)$ not only grows at least linearly with $N$, but is actually $\theta(N^2)$. If we apply the 2-binomial (triangular) transform, we will obtain a dynamic structure having cost functions, $P_D$ and $Q_D$, which satisfy

$$Q_D(N) \le 2Q_S(N) \qquad \text{and} \qquad P_D(N) = \theta(N^{5/2}).$$

The reader is advised to go through the exercise of verifying the latter assertion. The penalty factor in preprocessing is given by

$$G(N) = P_D(N)/P_S(N) = \theta(N^{1/2}),$$

which is at most a constant factor improvement over the worst-case result given in Theorem 3.2. We appear to get negligible compensation for the fact that the preprocessing cost is already much more than linear. If we look a little more carefully, however, we may notice an interesting phenomenon.

In the triangular strategy, we maintain two structures, a large one, having cardinality $O(N)$, and a small one, having cardinality $O(N^{1/2})$. If we break down $P_D(N)$ into the cost of forming all the large structures built during the first $N$ insertions and the cost of forming all the small structures built during the first $N$ insertions, we find that the large structures have a total cost of $\theta(N^{5/2})$, while the total cost of the small structures is only $\theta(N^2)$. If $P_S$ had been linear, then the costs of the two families of structures would have been equal within a constant factor, each being $\theta(N^{3/2})$. The present disparity suggests that it might be better to merge the small structures into the large ones less frequently. And, indeed, if we adopt the strategy of rebuilding all the elements into a single structure only when the size of the small structure would exceed $N^{2/3}$, we achieve a dynamic structure having

$$Q_D(N) \le 2Q_D(N) \qquad \text{and} \qquad P_D(N) = \theta(N^{7/3}) = O(N^{1/3}P(N))$$

(as the reader may again wish to verify), the total preprocessing cost being split evenly (within a constant factor) between the two families of structures. The preceding results may be generalized to arbitrary polynomial preprocessing costs and arbitrary binomial transforms, as shown in the following theorem.

THEOREM 4.13 (shift-of-strategy speed-ups). *Let $k$ be an arbitrary positive integer and let $r$ be a real number*[18] *gretaer than* 1. *Suppose that we are given a static structure for a decomposable searching problem with cost functions satisfying the following criteria*:

*$Q_S(N)$ is monotone nondecreasing.*

*$P_S(N)$ grows at least linearly, and*

*$P_S(N) = \omega(N^r)$.*

Then, a dynamic data structure can be constructed such that

$$Q_D(N) \leq kQ_S(N) \qquad \text{and} \qquad P_D(N) = O(N^R P_S(N)),$$

where

$$R = (r-1)/(r^k-1).$$

*Proof.* We maintain a set of structures satisfying the following invariants:

(1) After any insertion there are at most $k$ static structures.

(2) Let $j$ be a positive integer. After the $N$th insertion, the cardinality, $C_j$, of the $j$th largest structure (if there are at least $j$ structures in existence) satisfies

$$C_j \leq N^{(r^k - r^{j-1})/(r^k-1)}.$$

When an element is inserted, we see how many structures already exist. If there are fewer than $k$, we simply build the new element into a static structure of cardinality one. If $k$ structures already exist, we rebuild the smallest structure to include the new element. We then repeatedly (zero or more times) merge the smallest two structures until (2) is satisfied. We leave it to the reader to verify that this strategy achieves the advertised performance. Q.E.D.

In any strategy based on the construction in the previous proof, the total preprocessing will be divided evenly (up to constant factors) among $k$

[18] The nit-picking reader will delight in noting that it is not quite correct to allow $r$ to be an arbitrary real number. In order for the desired transform to be implementable, $r$ must be Turing computable. Even then, if $r$ is very expensive to compute, the bookkeeping costs may kill us. Similar considerations apply to the function $h$ in Theorem 4.11.

families of structures. We conjecture that this gives optimal $P_D$ within a constant factor (which may depend on $r$ and $k$). Needless to say, similar improvements are available, both in preprocessing time and in query time, for a number of other transformations, given sufficiently fast-growing cost functions. Only a small fraction of the possibilities have been explored.

## 5. Online Transformations

All the transforms in Section 3 have the property that some insertions are very cheap while others are very expensive. For example, in the binary transform the 1023rd insertion is much less costly than the 1024th. While this situation is quite acceptable in certain applications (such as when the total cost of accessing a structure throughout an entire algorithm is counted), it is prohibitive in others (such as online data bases). In this section we will show how the transforms in Sections 3 can be modified to amortize the cost of building static structures over the time of many insertions.

In Section 4, we worked on the principle that any static structure might as well be formed as soon as all its elements became available, since the cost of building it would eventually have to be paid anyway. While this is reasonable if we are concerned only with the total cost of all insertions, it is inappropriate if we wish to make sure that no individual insertion is inordinately expensive. Figure 5.1 shows a strategy which is similar to the binary strategy of Subsection 3.1, except that each structure of cardinality $C$ is completed at the end of the $C$th insertion that al! its elements are available, rather than at the end of the first such insertion. A structure, $s$, is said to be *pending* during the $N$th insertion if the all elements of $s$ become available at or before the beginning of the $N$th insertion and $s$ is completed during the $N$th insertion or later. (The $\times$'s in Fig. 5.1 denote the structures that are pending during the eighth insertion). A structure of cardinality $C$ will therefore be pending during exactly $C$ insertions.

To limit the work done in any insertion step, we require that $1/C$ of the work required to build any structure of size $C$ be performed during each of the $C$ steps in which that structure is pending.[19] We call the resulting

[19]The exact means by which this is ensured are left unspecified. We may modify the static algorithm to include appropriate breakpoints (generally an easier task than totally reworking the algorithm into a dynamic algorithm by *ad hoc* methods), or we could assume that we can determine the required computation time in advance (at negligible cost) and set a hardware interrupt. For our present purposes, we will assume that the ability to partition the compute time of a call to insert is available by magic. It should also be noted that the partitioning of the work into equal parts will not be exact in practice; this will lead to slightly greater insertion times than those we are about to advertise.

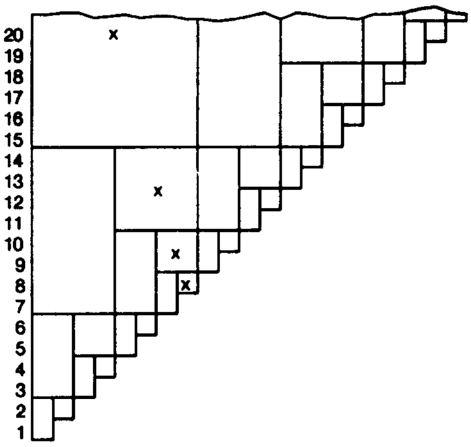

FIG. 5.1. The online binary transform.

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*

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

*(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 $N$th insertion or pending during the $N$th 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 $N$th 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

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

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

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

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

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

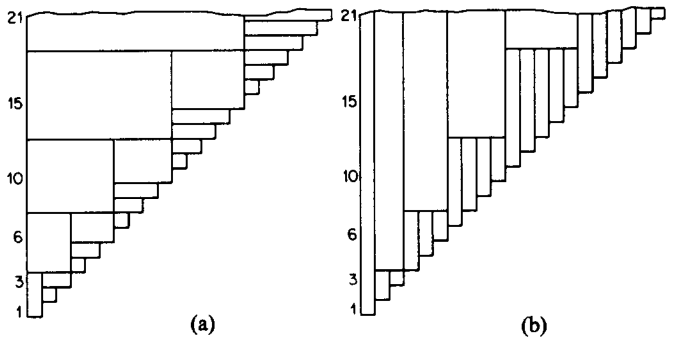

FIG. 5.2. (a) Online triangular transform. (b) Online dual triangular transform.

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

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

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

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

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

(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

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

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

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

The last two inequalities both follow from the fact that $P_{\mathrm{S}}$ 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

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

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 $P_{\mathrm{DD}}(M, N)$ denote the total insertion cost involved in a sequence of $N$ insertions and $M$ deletions in an initially empty structure. The function $Q_{\mathrm{DD}}(M, N)$ denotes the cost of answering a query in a structure built by $N$ insertions and $M$ deletions. Finally, $D_{\mathrm{DD}}(M, N)$ denotes the total time spent in processing deletions in a series of $N$ insertions and $M$ deletions, and $S_{\mathrm{DD}}(M, N)$ denotes the maximum 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*

$$S_{DD}(M,N) \le S_S(2(N-M)) + S_S(N-M),$$
$$P_{DD}(M,N) \le G(N)\cdot P_S(N),$$
$$Q_{DD}(M,N) \le F(2(N-M))\cdot Q_S(2(N-M)) + F(N-M)\cdot Q_S(N-M),$$
$$D_{DD}(M,N) \le G(M)\cdot P_S(M) + P_S(2M).$$

*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 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

$S$ 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*

$$\begin{aligned} S_{\mathrm{DO}}(N) &\le S_{\mathrm{S}}(N), \\ B_{\mathrm{DO}}(N) &\le P_{\mathrm{S}}(N), \\ D_{\mathrm{DO}}(N) &\le P_{\mathrm{S}}(N^{1/2}), \\ Q_{\mathrm{DO}}(N) &\le N^{1/2} Q_{\mathrm{S}}(N^{1/2}). \end{aligned}$$

*The function $B(N)$ denotes the time required to Build a structure of $N$ elements. We assume that the functions $S_{\mathrm{S}}$ and $P_{\mathrm{S}}$ grow at least linearly and that $Q_{\mathrm{S}}$ 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 "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 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 applicability 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.

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 $S$.

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.

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.