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

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

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

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

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).

(This immediately yields the corollary that

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

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

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

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]

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

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.

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

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

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 $M < N$,

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

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

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

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

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

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 $P_S(2i) \geq 2P_S(i)$ and we can use this fact inductively to show that

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

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

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

[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 $\lg N$.

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 Query_S, Build_S, and Unbuild_S (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 Init_D (which initializes the structure to be empty), Insert_D, and 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[High] is always empty. Init_D initializes the structure to have this invariant. Query_D answers a query by iterating through the structures and combining the answers by the $\square$ operator. 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 Init_D ←
    P[0] ← ∅; High ← 0

proc Insert_D(x) ←
    S ← {x}
    i ← 0
    while P[i] ≠ ∅ do
        S ← S ∪ Unbuild_S(P[i]); P[i] ← ∅
        i ← i + 1
    P[i] ← Build_S(S)
    if i = High then
        High ← High + 1; P[High] ← ∅

func Query_D(x) ←
    A ← Query_S(x, P[0])
    for i ← 1 to High-1 do
        A ← □(A, 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.

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

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

For N a power of 2 we can rewrite this as

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

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

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

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

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

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.

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 4×4 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 $P_{\text{DNN}}(N)$. We will perform this analysis

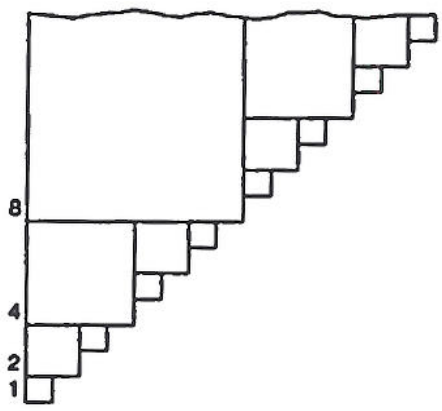

FIG. 3.1. The binary transform.

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

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

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.

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

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

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

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 $\square$ satisfying the condition

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

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

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

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 $\square$ operator to combine these subanswers to yield a solution to the entire problem. Note that the $\square$ 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 $\square$ operator has a "zero" (or "sticky") element; that is, there exists some element z such that for every element x,

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

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*

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