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 ith), 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

$$G(N) = P_D(N)/P_S(N) \leq g(N)/N.$$ 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

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

$$F(N) = \max_{1 \le i \le N} Q_D(i)/Q_S(i) \quad \text{and} \quad G(N) = P_D(N)/P_S(N).$$

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.

4.6. 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

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

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

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

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.

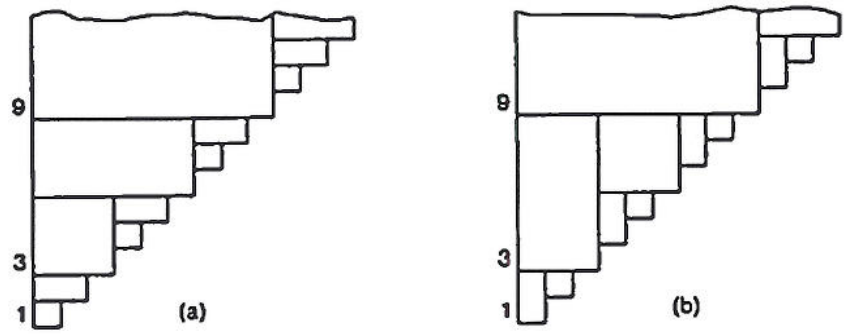

FIG. 3.11. Radix-3 transformations. (a) The ternary transform. (b) The dual ternary transform.

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

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 $((k-1)\lceil \log_k N \rceil, \lceil \log_k N \rceil)$ 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 $N > 1$.

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

$$Q_{\text{SRS}'}(N) = O(N^{\delta}),$$
$$P_{\text{SRS}'}(N) = O(N \lg N),$$
$$S_{\text{SRS}'}(N) = O(N),$$

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

$$Q_{\text{DRS}'}(N) = O(N^{\epsilon}),$$
$$P_{\text{DRS}'}(N) = O(N \lg N),$$
$$S_{\text{DRS}'}(N) = O(N)$$

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 $O(N \lg N)$.

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

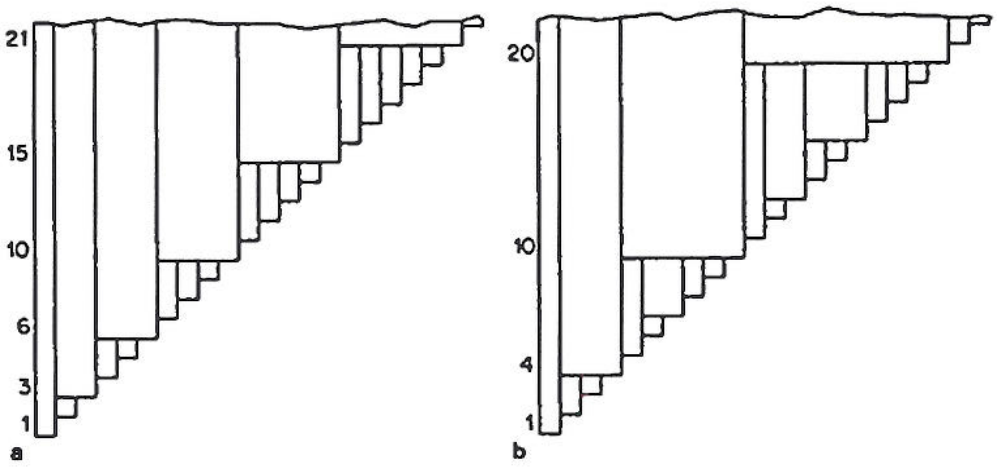

FIG. 3.8. Dual binomial transforms. (a) The dual triangular transform. (b) The dual 3-binomial transform.

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

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 $1 \leq i \leq d$. (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

$$Q_{\text{SRS}}(N) = O(\lg N),$$
$$P_{\text{SRS}}(N) = O(N^{1+\delta}),$$
$$S_{\text{SRS}}(N) = O(N^{1+\delta})$$

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

$$Q_{\text{DRS}}(N) = O(\lg N),$$
$$P_{\text{DRS}}(N) = O(N^{1+\epsilon}),$$
$$S_{\text{DRS}}(N) = O(N^{1+\epsilon})$$

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.

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 "$|A - B| = o(B)$."

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 ← 1 to k do
        D[i] ← i − 1; P[i] ← ∅
    D[k + 1] ← ∞

proc Insert_D(x) ←
    D[1] ← D[1] + 1; S ← Unbuild_S(P[1]) ∪ {x}; P[1] ← ∅
    i ← 1
    while D[i] = D[i + 1] do
        D[i + 1] ← D[i + 1] + 1; S ← S ∪ Unbuild_S(P[i])
        D[i] ← i − 1; P[i] ← ∅
        i ← i + 1
    P[i] ← Build_S(S)

func Query_D(x) ←
    A ← Query_S(x, P[1])
    for i ← 2 to k do
        A ← □(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 $\left(k, (k!N)^{1/k}\right)$ 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

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 $2 \leq i \leq k$. 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] ← D[1] + 1
i ← 1
while D[i] = D[i + 1] do
    D[i + 1] ← D[i + 1] + 1
    D[i] ← i − 1
    i ← i + 1
```

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

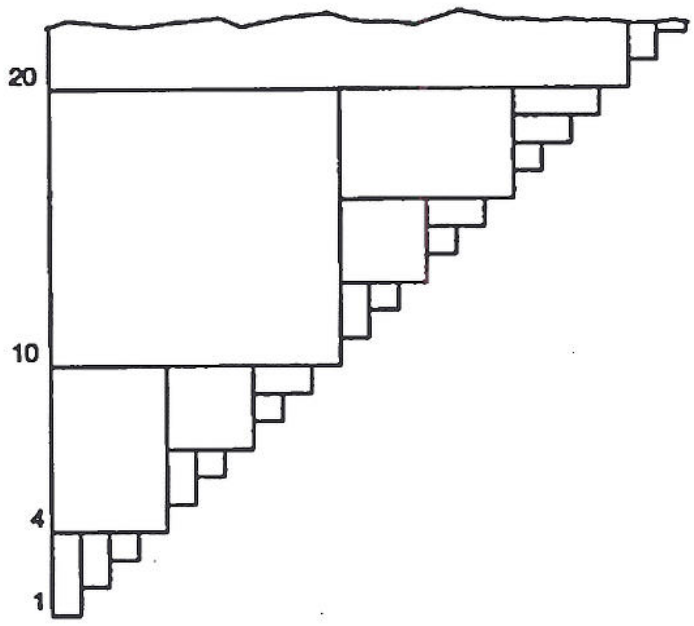

FIG. 3.7. The 3-binomial transform.

the arguments in the previous subsection, this implies

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

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	$(_2)$	$(_1)$	Integer	$(_3)$	$(_2)$	$(_1)$
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	6	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.

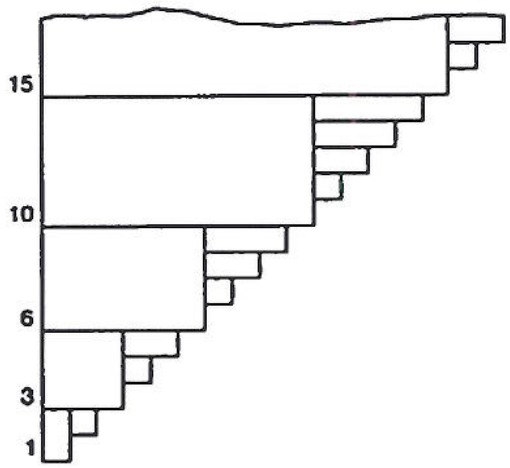

FIG. 3.4. The triangular transform.

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 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 ϵ. 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

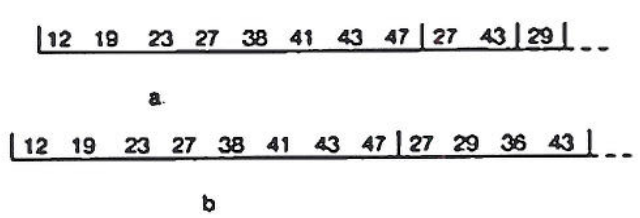

FIG. 3.3. Snapshots of a binomial list. (a) An 11-element binomial list. (b) After inserting 36.

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.