BIT 28 (1988), 37–53

ON THE DYNAMIZATION OF DATA STRUCTURES

NAGESWARA S. V. RAO†, VIJAY K. VAISHNAVI‡ and S. SITHARAMA IYENGAR†

† *Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA*

‡ *Department of Computer Information Systems, Georgia State University, University Plaza, Atlanta, GA 30303, USA*

Abstract.

In this paper we present a simple dynamization method that preserves the query and storage costs of a static data structure and ensures reasonable update costs. In this method, the majority of data elements are maintained in a single data structure, and the updates are handled using "smaller" auxiliary data structures. We analyze the query, storage, and amortized update costs for the dynamic version of a static data structure in terms of a function f, such that $f(n) < n$, that bounds the sizes of the auxiliary data structures (where n is the number of elements in the data structure). The conditions on f for minimal (with respect to asymptotic upper bounds) amortized update costs are then obtained. The proposed method is shown to be particularly suited for the cases where the merging of two data structures is more efficient than building the resultant data structure from scratch. Its effectiveness is illustrated by applying it to a class of data structures that have linear merging cost; this class consists of data structures such as Voronoi diagrams, K-d trees, quadtrees, multiple attribute trees, etc.

CR Categories: E.1.

Additional Keywords and Phrases: dynamization, amortized insertion and deletion costs.

1. Introduction.

In recent years, the dynamization of data structures has evolved into a well-established discipline in the area of Theoretical Computer Science. This fact is reflected in the large number of research results reported in the past five or six years. The design of efficient dynamic data structures has been continuing to be a challenging task. The main objective of the dynamization is to design data structures that achieve the query and storage costs of static data structures with fast update times. It is desirable to obtain a dynamic version for a given static data structure wherein the cost of n insertions (starting with the empty set) is approximately equal to the cost of preprocessing n elements to construct the static data structure. The single-dimensional data structures such as AVL-trees, etc. have such optimal characteristics. However, similar results seem extremely difficult to obtain for a general data structure. Many techniques exhibit some

Received September 1986. Revised September 1987.

interesting trade-offs by achieving optimality for some performance measures at the expense of the other measures.

Many techniques have been proposed and investigated for dynamizing data structures. See Overmars [6] and Mehlhorn [4] for a comprehensive treatment on the techniques. They can be very broadly classified into two classes. In the first class, the entire set of data elements is stored in a single data structure; the insertions and deletions are accommodated by carrying out some local or global changes on it. Overmars [6] presents a comprehensive treatment on the various techniques of this class such as local, partial, and global rebuilding. In the second class, a set of n data elements is organized into a collection of data structures. Of special significance is the case when the set of data elements is organized into a set of $\log n$ data structures. Bentley and Saxe [1] present a detailed discussion on these techniques for decomposable search problems (see Mehlhorn [4], and Overmars [6] also). Again, the appropriateness of a particular technique to an application depends on the nature of the problem. A close look at these methods seems to reveal that it may not be possible to obtain a general methodology that achieves optimal performance for all costs.

In this paper we present a simple dynamization technique that preserves the query and storage complexities of a static data structure. We consider the case where no element is repeatedly inserted or deleted. First, we consider the decomposable problems for semi-dynamic solutions. Then, we present a dynamic solution for a class of problems called the deletion-decomposable problems. These problems, to be defined formally in section 3, are similar to decomposable searching problems. Problems such as the member searching problem, the range search problems, the counting problem, etc., belong to this class of deletion-decomposable problems. In our method the majority of data elements are stored in a single *main* data structure. The insertions and deletions are carried out by using two *auxiliary* data structures; one to handle insertions and the other to handle deletions. To carry out these operations, the data elements are inserted into the auxiliary data structures. When these have grown to a certain size, a new main data structure is constructed by merging the existing main and auxiliary data structures. The sizes of the auxiliary data structures are restricted to $f(n)$, a value given by a positive, non-decreasing and integer-valued function $f: N \to N$, where n is the size of the main data structure at this point, such that $f(1) = 1$ and $f(n) < n$, $n > 1$. In this case the query and the storage costs of the dynamic version are the same as those of the static data structure. The amortized insertion and deletion costs are estimated in terms of $f(n)$ and in terms of the parameters of the static data structure. The choice of f is *static*, i.e. f is decided based on the parameters of the static data structure once at the beginning. Only the value of $f(n)$ is computed after every insertion or deletion. Different trade-offs in the insertion and deletion costs exist based on the function f chosen for the application. We also obtain the condition on f that yields minimal (with respect to derived asymptotic upper bounds) update costs.

A closed form expression for f (expressed in terms of n) satisfying this condition is not guaranteed in a general case. This technique is particularly suited for the case where merging two existing data structures – to form a larger data structure – is more efficient than building the resultant data structure from scratch. We illustrate the effectiveness of this technique by applying it to a class of data structures with linear merging costs. This class includes data structures such as Voronoi diagrams, quadtrees, k-d trees, multiple attribute trees, balanced binary search trees, etc. The preprocessing cost is $O(n \log n)$ for these examples, where n is the size of the set of data elements. Each data structure of size n can be constructed by merging two smaller data structures of sizes n_1 and n_2, $n = n_1 + n_2$. Thus the merging cost is $O(n)$. The amortized insertion cost is $O(n/\sqrt{\log n})$, and the amortized deletion cost is $O(n\sqrt{\log n})$ for these data structures in one case. In a second case, both the amortized insertion and deletion costs are given by $O(n\sqrt{(\log n/\log\log n)})$ for this class of data structures.

We characterize a static data structure S, containing n elements, with the following parameters:

$Q_S(n)$: cost of answering a query on S;
$S_S(n)$: amount of storage needed for storing S;
$P_S(n)$: cost of building S, i.e. preprocessing cost of S;
$M_S(n)$: cost of merging two data structures (each of type S)
of sizes n_1 and n_2 respectively to form
single data structure (of type S) of size $n = n_1 + n_2$.

Let $P_S(n) = M_S(n)\xi(n)$. The function $\xi(n)$ is at least of the order $O(1)$ and $M_S(n)$ is at least of the order $O(n)$. In general $\xi(n)$ is indicative of how efficient building a data structure by merging two existing sub-data structures is, compared to building the data structure from scratch. For Voronoi diagrams we have $\xi(n) = O(\log n)$.

We characterize a dynamic data structure D with the following parameters:

$Q_D(n)$: cost of answering a given query on D;
$S_D(n)$: amount of storage needed to store D;
$\bar{I}_D(n)$: amortized insertion cost,
given by [maximal total time spent on executing a sequence of n insertions starting with an empty set]/n;
$\bar{D}_D(n)$: amortized deletion cost,
given by [maximal total time spent on executing a sequence of n *operations*[1] (insertions or deletions) starting with an empty set]/n;

The organization of this paper is as follows: In Section 2, we present a semi-

[1]) In [4], a query is also considered as an operation. In our discussion the amortized deletion cost could be more appropriately called amortized update cost. However, we retain the former for pedantic reasons.

dynamic solution for the decomposable searching problems. In particular, we evaluate the query, storage, and amortized insertion costs in terms of $f(n)$. We also obtain the condition for the minimal form of the derived asymptotic upper bound on the amortized insertion cost. We then present a dynamic solution for the deletion-decomposable searching problems in Section 3. We estimate the various costs in terms of $f(n)$. We obtain the condition for the minimal form for the derived asymptotic upper bounds on the amortized insertion and deletion costs. Application of this technique to a class of data structures with linear merging costs is presented in Section 4.

2. The semi-dynamic solution.

In this section we present a semi-dynamic version D of a static data structure S, i.e., D supports queries and insertions. Here, we consider the decomposable searching problems. A problem π on a set A is said to be *decomposable* if for all partitions B, C of A, i.e. $A = B \cup C$, $B \cap C = \phi$, we have

$$\pi(A) = \square(\pi(B), \pi(C))$$

where $\pi(R)$ denotes the answer for the problem π on the set R [1]. Moreover $\square$ is computable in $O(1)$ time.

At any point of time the set of data elements is stored in two data structures; the *main* data structure M and the *auxiliary* data structure I. Let D contain n data elements of which n_1 elements are stored in M and the remaining are stored in I (see Fig. 1). In Fig. 1 each data structure is represented as a triangle. The

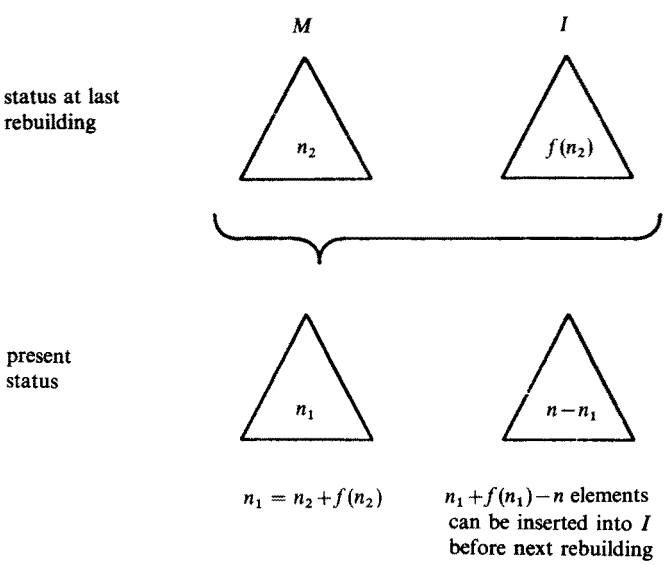

Fig. 1. Semi-dynamic solution.

number inside the triangle indicates the number of elements contained in the data structure. A query on D is handled by posing it individually on M and I, and then combining the results to give an answer to the query. Insertion of a new element x is carried out by inserting x into I by merging it into I. The auxiliary data structure I is allowed to grow to the size given by $f(n_1) < n_1$. After the size of I becomes $f(n_1)$, I and M are combined to form a new M of size $n_1 + f(n_1)$. This process is called the *rebuilding*. The rebuilding operation is carried out on-line as soon as I attains the size of $f(n_1)$. Now we have $n_1 \leq n \leq n_1 + f(n_1)$ (see Figure 1). We analyze the performance of this technique subsequently in this section. We estimate the query, storage and amortized insertion costs of D in terms of the performance measures of S and the function f.

(a) Query and storage costs:

The cost of a query on D is given by

$$Q_D(n) \leq Q_D(n_1 + f(n_1)) \leq Q_S(n_1) + Q_S(f(n_1)).$$

The first term $Q_S(n_1)$ corresponds to the cost of answering the query on the main data structure M, and the second term $Q_S(f(n_1))$ corresponds to the cost of answering the query on the auxiliary data structure I. The cost of combining the partial results is $O(1)$. Now, $f(n_1) < n_1$, and

$$Q_D(n) \leq 2Q_S(n_1) = O(Q_S(n)).$$

Thus, the dynamic version D has the same query complexity as the static version S. The storage cost of D is given by

$$S_D(n) \leq S_S(n_1) + S_S(n - n_1).$$

The first term corresponds to the storage cost of the main data structure M, and the second term corresponds to the storage cost of the auxiliary data structure I. We have $n - n_1 \leq f(n_1) \leq n_1$, and hence $S_S(n - n_1) \leq S_S(f(n_1)) \leq S_S(n_1)$. Therefore,

$$S_D(n) \leq 2S_S(n_1) = O(S_S(n)).$$

Thus, the dynamic version D has the same storage complexity as the static version S. This is significant, as many of the existing dynamization methods result in an increase in one or both of these costs.

(b) Amortized insertion cost:

We compute the amortized insertion cost by computing the total cost involved in building D and dividing this cost by n. The number of rebuilding operations carried out in building a D of size n is given in the following lemma.

LEMMA 1: *The number of rebuildings, k, carried out in building the data structure D of size n, n > 1, has the following bounds*

(*i*) $k = \Omega(\log(n - f(n)))$

(*ii*) $k = O(n/w(n))$, *where* $w(n) = f(\frac{1}{2}\log(n - f(n)) + 1)$

(*iii*) $k = O(n/f(\frac{1}{2}\log n + 1))$ *if the nth insertion results in a rebuilding operation.*

PROOF: Let $T(i)$ denote the size of M after the ith merging. At the point of the ith merging we have

$$\begin{aligned} T(i) &= T(i-1) + f(T(i-1)) \\ T(i) - T(i-1) &= f(T(i-1)). \end{aligned} \tag{2.1}$$

Now $f(n) < n$ for $n > 1$ and we have

$$\begin{aligned} T(k) &\leq 2T(k-1) \\ T(k) &\leq 2^k T(0). \end{aligned}$$

Initially we assume that the main data structure contains a single element, i.e., $T(0) = 1$, and hence we have $T(k) = n_1 \leq 2^k$. Thus the lower bound on k is given by

$$k \geq \log n_1. \tag{2.2}$$

By expanding equation (2.1) we have

$$\begin{aligned} T(k) - T(k-1) &= f(T(k-1)) \\ T(k-1) - T(k-2) &= f(T(k-2)) \\ \cdots &= \cdots \\ T(k/2+1) - T(k/2) &= f(T(k/2)) \end{aligned} \tag{2.3}$$

Summation of the above equations results in the following equation:

$$\begin{aligned} & T(k) - T(k/2) = f(T(k-1)) + f(T(k-2)) + \cdots + f(T(k/2)) \\ & T(k) \geq (k/2) f(T(k/2)) \\ & k \leq 2T(k)/f(T(k/2)). \end{aligned} \tag{2.4}$$

From equation (2.2) we have $T(k/2) \geq T(\frac{1}{2}\log n_1) \geq \frac{1}{2}\log n_1 + 1$, since $T(0) = 1$ and f is a positive integer-valued function. Thus we have, $f(T(k/2)) \geq f((\log n_1/2) + 1)$. Using this value in equation (2.4), we get

$$2n_1/f(\tfrac{1}{2}\log n_1 + 1). \tag{2.5}$$

Now $n \geq n_1$ and $n \leq n_1 + f(n_1) \leq n_1 + f(n)$. Thus, $n_1 \geq n - f(n)$, and $f(n_1) \geq f(n - f(n))$. Also equation (2.2) reduces to

$$k = \Omega(\log(n - f(n))).$$

Thus Part (i) is proven. Putting $w(n) = f(\frac{1}{2}\log(n - f(n)) + 1)$ equation (2.5)

reduces to

$$k = O(n/w(n)).$$

Thus Part (ii) is proven. After the nth insertion we have $n_1 = n$. Using (2.5) we obtain

$$k \leq 2n/f(\tfrac{1}{2}\log n + 1) = O(n/f(\tfrac{1}{2}\log n + 1)).$$

Hence, Part (iii). ■

We now compute the cost $P_D(n)$ incurred in constructing a D of size n. This cost has two components:

(a) The total cost involved in carrying out various rebuilding operations. Each rebuilding operation involves merging the elements of the main data structure M and the elements of the auxiliary data structure I to form a new main data structure.

(b) The total cost of building the auxiliary data structures. Note that the auxiliary data structures are built as the insertions are carried out.

Consider that rebuilding operation which involves the largest number of elements. It is the latest rebuilding operation that resulted in M of size n_1 (see Figure 1). This operation involves the merging of M of size n_2 with I of size $f(n_2)$, where $n_1 = n_2 + f(n_2)$. The cost of any rebuilding operation carried out (so far in the construction of D) is less than or equal to $M_S(n_2 + f(n_2)) = M_S(n_1) \leq M_S(n)$. There can be $O(n/w(n))$ rebuilding operations. Hence, the total cost involved in the rebuilding operations is $O(nM_S(n)/w(n))$. After the rebuilding operation, I is initialized to an empty data structure, i.e., contains no elements. The new elements are progressively merged into the existing data structure I till it reaches the size given by $f(n_1)$. The cost of any insertion operation is at most $M_S(f(n_1) - 1 + 1) \leq M_S(f(n))$. In the construction of D, there are $O(nf(n)/w(n))$ insertions, in view of Lemma 1. Thus the amortized insertion cost is given by

$$\bar{I}_D(n) = P_D(n)/n$$

$$= O\left(\frac{M_S(n) + f(n)M_S(f(n))}{w(n)}\right) = O\left(\frac{P_S(n)}{w(n)\xi(n)} + \frac{f(n)P_S(f(n))}{w(n)\xi(f(n))}\right)$$

since $M_S(n) = P_S(n)/\xi(n)$. Note that the first term corresponds to the total cost incurred in carrying out the rebuilding operations, and the second term to the total cost incurred in building the required auxiliary data structures. The discussion of this section proves the following Theorem.

THEOREM 1: *Given a static data structure S and f (with parameters defined as in section 1), there exists a semi-dynamic data structure D such that, for n > 1,*

$$Q_D(n) = O(Q_S(n))$$

$$S_D(n) = O(S_D(n))$$

$$\bar{I}_D(n) = O\left(\frac{M_S(n)+f(n)M_S(f(n))}{w(n)}\right)$$

Consider $U(\bar{I}_D(n)) = (M_S(n)+f(n)M_S(f(n)))/w(n)$, the asymptotic upper bound on $\bar{I}_D(n)$ given in the above theorem. If $f(n) = \Theta(n)$ then $U(\bar{I}_D(n)) = O(nM_S(n))$, since in such case the order of $w(n)$ is at least $O(1)$. In particular, if $n/c_1 \leq f(n) \leq n/c_2$ where c_1 and c_2 are positive integers greater than 1, then we can directly derive that $w(n) = \Theta(\log n)$ and $\bar{I}_D(n) = O(nM_S(n)/\log(n))$.

If, on the other hand, $f(n) < \Theta(n)$ (i.e. $f(n)$ is an order less than $O(n)$), then we have $f(n) < n/2$, $n-f(n) = \Theta(n)$, since $f(n) \geq 1$, and $w(n) = \Theta(f(\log n))$. Then,

$$\begin{aligned} U(\bar{I}_D(n)) &= (M_S(n)+f(n)M_S(f(n)))/f(\log n) \\ &= \Theta(f(n)M_S(f(n))/f(\log n)) \end{aligned}$$

when $M_S(n) < \Theta(f(n)M_S(f(n)))$, a function that decreases (in its rate of growth) as f decreases (note that $M_S(n) = \Omega(n)$), or

$$= \Theta(M_S(n)/f(\log n))$$

when $M_S(n) \geq \Theta(f(n)M_S(f(n)))$, a function that increases as f decreases. Clearly, the minimum (rate of growth) of $U(\bar{I}_D(n))$ is achieved when the following condition is satisfied:

$$M_S(n) = \Theta(f(n)M_S(f(n))). \tag{2.6}$$

Equivalently, we have

$$P_S(n)/\xi(n) = f(n)P_S(f(n))/\xi(f(n)). \tag{2.7}$$

Intuitively, $U(\bar{I}_D(n))$ is composed of two conflicting terms (namely the total cost of rebuildings and the total construction cost of auxiliary data structures), and an attempt to reduce one by suitably choosing f will result in the increase of the other. Thus if a critical function f is chosen such that the cost of rebuildings is equal to the accumulated cost of constructing the auxiliary data structure, to within a constant factor, then we have obtained the minimal form for $U(\bar{I}_D(n))$. Any deviation of f from this critical form will result in an increase in one of the two terms of $U(\bar{I}_D(n))$. A value for f lower (in terms of order) than the critical value results in an increased number of rebuildings and thus the first term dominates. On the other hand a value for f larger (in terms

of order) than the critical value results in high cost for building the auxiliary data structures, and thus the second term dominates $\bar{I}_D(n)$.

In general, the f that satisfies the condition in (2.7) may or may not have a closed form expression. For the case where such an f has a closed form expression, the amortized insertion cost is given by

$$\bar{I}_D(n) = O(P_S(n)/w(n)\xi(n)) = O(M_S(n)/w(n)).$$

For the corresponding case where the nth insertion initiates a rebuilding operation, we can write

$$\bar{I}_D(n) = O(M_S(n)/f(\tfrac{1}{2}\log n)).$$

In section 4 we discuss a class of data structures that have linear merging times. This class includes Voronoi diagrams, k-d trees, quad trees, multiple attribute trees, balanced binary trees, etc. For this class $f(n) = \sqrt{n}$, satisfies the condition for the minimality given in (2.7). In the next section, we present a dynamic solution that supports insertions and deletions.

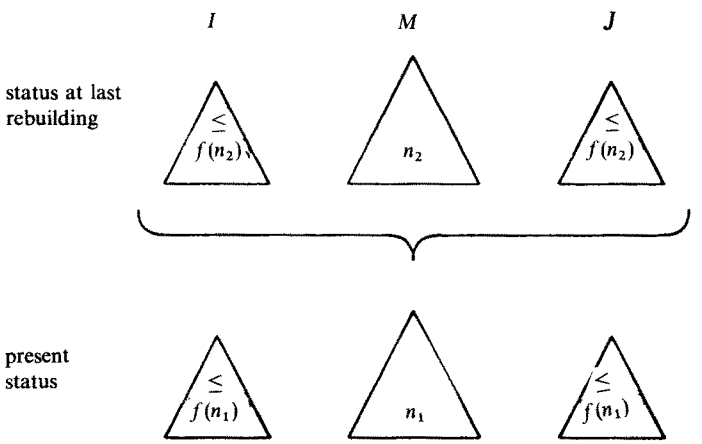

Fig. 2. Dynamic solution.

3. The dynamic solution.

The dynamic solution that supports insertions and deletions is obtained by extending the concepts used in the semi-dynamic solution described in Section 2. The composite data structure D now contains a second auxiliary data structure J of the same type as S. When an element x is to be deleted, it is inserted into J by merging x into J. A query on D is processed by posing it individually on

M, I, and J and then combining the partial answers. We restrict our treatment to the class of deletion-decomposable problems. A problem π on a set A is said to be *deletion-decomposable* if there exists a function $\square$ such that

$$\pi(A) = \square(\pi(A \cup B), \pi(B))$$

for any set B, $A \cap B = \phi$, where $\pi(R)$ denotes the answer to π on the set R. Furthermore $\square$ is computable in a time less than or equal to the time taken for the computation of $\pi(A \cup B)$ or $\pi(B)$, whichever is larger. Many problems belonging to the classes of membership, counting, and enumeration problems satisfy this property. Table 1 shows the results for computing $\square$ for a membership problem. In this case the answer $\pi(A)$ is either "yes" or "no". As an example for an enumeration problem consider the range search problem. In this case $\pi(A)$ is a subset of A containing the elements (of A) that satisfy the range condition specified by π. We have $\pi(A) = \pi(A \cup B) - \pi(B)$. The computation time of this set intersection is at most equal to the time taken for computing $\pi(A \cup B)$.

Table 1. *The computations of* $\square$ *for membership problem.*

$\pi(A \cup B)$	$\pi(B)$	$\pi(A)$
yes	yes	no
yes	no	yes
no	no	no
no	yes	does not arise

The *rebuilding* is carried out whenever either I or J reaches the size $f(n_1)$, where n_1 is the size of M (see Figure 2). A rebuilding operation is carried out (on-line) as soon as this condition is reached. The rebuilding operation is carried out in two phases: (a) the set of elements contained in either M or I and not contained in J are computed, (b) the main data structure is constructed by preprocessing the resultant set of elements obtained in phase (a). However, in the cases when J is empty, M and I are merged as in the semi-dynamic case. This condition can be checked easily by maintaining a flag variable. Subsequently in this section we analyze the query, storage and amortized update costs for the dynamic version D.

(a) Query and storage costs:

The cost of processing a query on D is given by

$$Q_D(n) \leq Q_S(n_1) + Q_S(f(n_1)) + Q_S(f(n_1)) + Q_S(n_1).$$

The first three terms correspond to answering the query on M, I, and J respectively. The fourth term corresponds to computing the final answer from the partial answers. Now we have

$$Q_D(n) \leq 4Q_S(n) = O(Q_S(n)).$$

The storage cost of D is given by

$$S_D(n) \leq S_S(n_1)+S_S(f(n_1))+S_S(f(n_1)).$$

The first, second and third terms correspond to the storage costs of M, I and J respectively. Now,

$$S_D(n) \leq 3S_S(n) = O(S_S(n)).$$

Thus the dynamic version D has the same query and storage costs as those of the static structure S. In the remainder of this section, we estimate the amortized insertion and deletion costs for the dynamic version D.

(b) Amortized insertion and deletion costs:

Consider the latest point of merging in the construction of D (Figure 2). At this point the present M, of size $n_1 \leq n_2+f(n_2)$, is built from the old M (of size n_2) and the corresponding I and J; each of I and J contains at most $f(n_2)$ elements and at least one of them contains $f(n_2)$ elements. We compute the set of elements contained either in M or in I but not in J as follows: we construct a balanced binary tree of the elements of J. The time complexity of this construction is $O(f(n)\log(f(n)))$. We compute the union of elements of M and I in a linked list L. Time complexity of constructing L is $O(n_1+f(n_1)) = O(n)$. We traverse L element by element checking for the membership of current element in the tree containing elements of J. If the element is present in the tree then it is removed from L. The cost of checking for each element is $O(\log(f(n)))$ and the cost of deletion is $O(1)$ (by keeping a pointer to the previously examined element in the list). Hence the cost of computing the set of elements contained either in M or in I and not in J is $O(n\log(f(n)))$. The cost of building a new M for this data structure is given by $P_S(n)$. Note that $n_1 = n_2+f(n_2)$ is an upper bound on the size of new M.

The amortized insertion cost is obtained by computing $P_D(n)$, the cost of constructing D by n consecutive insertions. In this case, at each rebuilding operation the J is empty, and hence the rebuilding operation consists of direct merging. Hence, this amortized cost is the same as that computed in section 2. Thus the amortized insertion cost is given by

$$\bar{I}_D(n) = O\left(\frac{M_S(n)+f(n)M_S(f(n))}{w(n)}\right).$$

We now compute the amortized deletion cost. First, we compute the total worst-case construction cost $P_D(n)$ incurred in carrying out a sequence of n update – either an insertion or a deletion – operations. This cost, again, is constituted by two factors: (a) the total cost involved in performing the rebuilding operations, and (b) the total cost involved in constructing the various I and J data structures. The worst-case sequence of n operations satisfies the properties stated in the following Lemma.

LEMMA 2: *The number of rebuildings involved in the worst-case sequence is* $O(n/w(n))$.

PROOF. A sequence of n operations must contain at least $n/2$ insertions, since the data structure is initially empty. The amortized insertion (deletion) cost has two components: (a) the cost corresponding to insertion into I (J). (b) The cost corresponding to the merging (preprocessing) operation at the time of rebuilding. The amortized insertion cost involves the "averaged" cost incurred in merging the element into the auxiliary data structure and the "averaged" cost involved in rebuilding operations carried out using merging. The amortized deletion cost involves the "averaged" cost incurred in merging the element into the auxiliary data structure and the "averaged" cost involved in rebuilding operations carried out using preprocessing. Since the preprocessing cost is at least as much as merging cost, the amortized insertion cost $\bar{I}_D(n)$ is at most as much as the amortized deletion cost $\bar{D}_D(n)$. Here, we have two cases: Case (i): When $\bar{I}_D(n) = \bar{D}_D(n)$, the worst-case sequence is clearly a sequence of n insertions. Thus the claim is true by (ii) of Lemma 1. Case (ii): Consider the case $\bar{I}_D(n) < \bar{D}_D(n)$. We now show that the cost of worst-case sequence is no more than the cost of a sequence of n insertions followed by a sequence of n deletions.

First, for the worst-case sequence, we prove that the sequence of operations, in between any two consecutive rebuildings, entirely consists of either insertions or deletions. Let i be the number of elements in D immediately after the kth rebuilding. The $(k+1)$th rebuilding takes place after the size of I or J reaches $f(i)$. The cost of building either I or J is $\sum_{j=1}^{f(i)} M_S(j) = O(\sum_{j=i}^{f(i)} M_S(j))$. The complexity of constructing both of them is also the same. By the same argument, the cost of constructing one (either I or J) fully and the other either fully or partially has the same cost of $O(\sum_{j=1}^{f(i)} M_S(j))$. Thus for maximizing the total cost only one should be constructed in between any two rebuildings.

We now show that the first $n/2$ operations must be insertions. Let $I_D(i)$ and $D_D(i)$ denote the insertion and deletion costs, respectively when D contains i elements. We have $I_D(i) \leq I_D(i+1)$ and $D_D(i) \leq D_D(i+1)$. The cost of the sequence of $n/2$ insertions followed by $n/2$ deletions is given by (assume n to be even for discussion sake)

$$C_1 = I_D(0)+I_D(1)+\cdots+I_D(n/2-1)+D_D(n/2)+D_D(n/2-1)+\cdots+D_D(1).$$

Now consider the case in which some deletions take place during the first $n/2$ operations. For the worst-case sequence the deletions must occur in a complete sequence in between two consecutive rebuildings. Let the sequence of $f(j)$ deletions take place after M is freshly built as a result of j insertions. Then the total cost corresponding to this sequence of n operations is given by

$$\begin{aligned} C_2 &= I_D(0)+\cdots+I_D(j-1)+D_D(j)+D_D(j-1)+\cdots+D_D(j-f(j)+1) \\ &\quad +I_D(j-f(j))+\cdots+I_D(n/2-f(j)-1)+D_D(n/2-f(j))+\cdots+D_D(1) \\ &= I_D(0)+\cdots+2[I_D(j-f(j))+\cdots+I_D(j-1)]+I_D(j)+\cdots+I_D(n/2-f(j)-1) \\ &\quad +D_D(n/2-f(j))+\cdots+2[D_D(j)+\cdots+D_D(j-f(j)+1)]+\cdots+D_D(1). \end{aligned}$$

Thus, $C_1 \geq C_2$, because both $I_D(n)$ and $D_D(n)$ are non-decreasing functions. Thus, in the worst-case sequence the first $n/2$ operations must be insertions. However, there may be more than $n/2$ such insertions in a worst-case sequence.

Let first j, $n/2 \leq j \leq n$, operations be insertions in a worst-case sequence. By applying the same method as above we can show that the next $n-j$ operations must be deletions. Thus a worst-case sequence contains j, $n/2 \leq j \leq n$, insertions followed by $n-j$ deletions. Now the cost of such a sequence is at most equal to the cost of a sequence consisting of n insertions followed by n deletions. The number of rebuildings in such a sequence is $O(n/w(n)+n/w(n)) = O(n/w(n))$. ■

The cost of each rebuilding operation is at most $P_S(n)+O(n\log(f(n)))$, and the number of such rebuilding operations is as given in Lemma 2. The cost of constructing an auxiliary data structure is the same as in the semi-dynamic case because the size of an auxiliary data structure is at most $f(n)$, and only one (either I or J) is constructed in between two rebuildings. Thus the amortized deletion cost is given by

$$\bar{D}_D(n) = O\left(\frac{P_S(n)+n\log(f(n))}{w(n)} + \frac{f(n)P_S(f(n))}{w(n)\xi(f(n))}\right).$$

The above discussion leads to the following Theorem.

THEOREM 2: *Given a static data structure S and f (with parameters as defined in section 1), there exists a dynamic data structure D such that, for $n > 1$*

$$\begin{aligned} Q_D(n) &= O(Q_S(n)) \\ S_D(n) &= O(S_S(n)) \\ \bar{I}_D(n) &= O\left(\frac{M_S(n)+f(n)M_S(f(n))}{w(n)}\right) \end{aligned}$$

$$\bar{D}_D(n) = O\left(\frac{P_S(n)+n\log(f(n))}{w(n)} + \frac{f(n)M_S(f(n))}{w(n)}\right).$$

As in the semi-dynamic case, if f is chosen such that $f(n)$ takes values near n then the second terms of $U(\bar{I}_D(n)) = (M_S(n)+f(n)M_S(f(n)))/w(n)$ and $U(\bar{D}_D(n) = (P_S(n)+n\log(f(n)))/w(n)+f(n)M_S(f(n))/w(n)$, (asymptotic upper bounds for $\bar{I}_D(n)$ and $\bar{D}_D(n)$, respectively) dominate. Similarly, a slower growing f makes the first term dominate. It is clear from Theorem 2 that different functional forms of f minimize $U(\bar{I}_D(n))$ and $U(\bar{D}_D(n))$. The condition for the minimal form for $U(\bar{I}_D(n))$ is the same as in condition (2.6) or (2.7). For such an f we have

$$\bar{I}_D(n) = O(P_S(n)/w(n)\xi(n)) = O(M_S(n)/w(n)).$$

This leads to the following complexities

$$\bar{D}_D(n) = O\left(\frac{P_S(n)+n\log f(n)}{w(n)}\right) \tag{3.1}$$

$$\bar{I}_D(n) = O(M_S(n)/w(n)). \tag{3.2}$$

Consider the cost of $\bar{D}_D(n)$. We have two cases.

Case (a): $P_S(n) \geq O(n\log n)$.

We have

$$\bar{D}_D(n) = O\left(\frac{P_S(n)}{w(n)} + \frac{f(n)M_S(f(n))}{w(n)}\right).$$

Let

$$U(\bar{D}_D(n)) = \left(\frac{P_S(n)}{w(n)} + \frac{f(n)M_S(f(n))}{w(n)}\right)$$

so that $\bar{D}_D(n) = O(U(\bar{D}_D(n)))$. The minimal form for $U(\bar{D}_D(n))$ is obtained when

$$P_S(n) = \Theta(f(n)M_S(f(n))) \tag{3.3}$$

using arguments similar to those used in showing (2.6). Equivalently we have

$$P_S(n) = \Theta(f(n)P_S(f(n))/\xi(f(n))). \tag{3.4}$$

If the above condition is satisfied then

$$\bar{D}_D(n) = O(P_S(n)/w(n) = \bar{I}_D(n). \tag{3.5}$$

Case (*b*): $P_S(n) < O(n \log n)$.

We have

$$\bar{D}_D(n) = O\left(\frac{n \log n}{w(n)} + \frac{f(n)M_S(f(n))}{w(n)}\right).$$

Let

$$U(\bar{D}_D(n)) = \left(\frac{n \log n}{w(n)} + \frac{f(n)M_S(f(n))}{w(n)}\right)$$

so that $\bar{D}_D(n) = O(U(\bar{D}_D(n)))$. It can be easily seen that the minimal form of $U(\bar{D}_D(n))$ is given by the condition

$$n \log n = \Theta(f(n)M_S(f(n))) \tag{3.6}$$

or equivalently, we have

$$n \log n = \Theta(f(n)P_S(f(n))/\xi(f(n))). \tag{3.7}$$

If the above condition is satisfied then

$$\bar{D}_D(n) = O(n \log n/w(n)) = \bar{I}_D(n).$$

We denote the choice of f satisfying (2.6) by f_1. We also denote the f that satisfies (3.3) or (3.6) by f_2. In general f_1 and f_2 are different and it is easy to see that $f_1(n) = O(f_2(n))$ since $f_1(n) = \Theta(M_S(n)/M_S(f(n)))$ by (2.6) and $f_2(n) = \Theta(P_S(n)/M_S(f(n)))$ by (3.3). Also, if f is chosen as f_1 then the derived asymptotic upper bound for $\bar{I}_D(n)$ is the same or better than if it is chosen as f_2. Conversely, if f is chosen as f_2 then the derived asymptotic upper bound for $\bar{D}_D(n)$ is the same or better than if it is chosen as f_1. Thus, we suggest that f_1 be chosen if insertions are more frequent than deletions, and f_2 be chosen otherwise.

4. Applications.

For a given static data structure S, and a given f, we can obtain the amortized deletion costs by applying Theorem 2. In this section, we discuss a special class of data structures that are naturally suited for the technique presented in this paper. This class is characterized by the property that two data structures can be merged in linear time. The examples are Voronoi diagrams, k-d trees, quadtrees, multiple attribute tree, balanced binary trees, etc. which are applied in various disciplines such as computational geometry, physical database organization, image processing, multidimensional searching, etc. The Voronoi

diagrams are extensively used in computational geometry for solving various proximity problems [3, 4, 7]. The quadtrees are used for two-dimensional image processing applications. The k-d trees and multiple attribute trees are used for various applications such as complete match, partial match, and range queries for multidimensional data [5]. Specifically, the multiple attribute trees are used for physical database organization [5]. All these data structures share the property that $M_S(n) = \Theta(n)$. Now the condition (2.6) is used as follows to obtain the minimal form for $f(n)$

$$M(n) = \Theta(f(n)M(f(n))) \tag{4.1}$$

$$f(n) = \sqrt{n}.$$

Using (3.1) and (3.2), we have

$$\bar{I}_D(n) = O(n/\sqrt{\log n}) \tag{4.2}$$

$$\bar{D}_D(n) = O((n\xi(n) + n\log n)/\sqrt{\log n}). \tag{4.3}$$

Now consider the Voronoi diagrams. We have $\xi(n) = O(\log n)$, $S_M(n) = O(n)$, $P_S(n) = O(n\log n)$.

Using (4.2) and (4.3) we obtain

$$\bar{I}_D(n) = O(n/\sqrt{\log n})$$

$$\bar{D}_D(n) = O(n\sqrt{\log n}).$$

Considering the condition (3.4) we have $n\log n = f(n)f(n)$ or equivalently $f(n) = \sqrt{(n\log n)}$.

Using (3.5) we obtain

$$\bar{D}_D(n) = \bar{I}_D(n) = O(n\log n/f(\tfrac{1}{2}\log n + 1)) = O(n\sqrt{\log n}/\sqrt{\log\log n}).$$

Notice that the complexity of amortized deletion cost is less in the latter case. This reduction results in an increase in the amortized insertion cost.

Similar results are presented in [2] for the Voronoi diagrams and related data structures used for geometric applications. In that paper, for Voronoi diagrams, the query cost is the same as the cost obtained here. Better update costs are realized by using additional amount of storage. Our technique presented in sections 2 and 3 is more generalized than that of [2]. The latter is restricted to data structures with linear merging costs whereas our method is applicable to a general data structure.

We note that the closed form expression for $f(n)$ that satisfies the condition in (2.7) or (3.3) or (3.6) is not guaranteed in a general case. In such a case, an expression for $f(n)$ that makes the two terms of $\bar{I}_D(n)$ and/or $\bar{D}_D(n)$ as close as possible, can be used as a good approximation. Furthermore, the bounds

in Theorems 1 and 2 are general. It may be possible to obtain tighter bounds in a specific given case.

5. Conclusions.

The technique presented in this paper can be modified and extended in many ways. First, the complexity estimates presented here are gross upper bounds. Evaluating tighter upper bounds and general lower bounds will provide more insight into the dynamization technique. Second, it would be interesting to see if f that gives rise to minimal or close to minimal form update costs has a closed form expression in a general case. We feel that some more classes of data structure (such as that presented in section 4) can be found wherein the f will have closed form expressions. Third, in our approach the auxiliary data structures are of the same type as the main data structures. The technique presented in this paper can be generalized by allowing the auxiliary data structures to be of any general type. It is interesting to see if some specific choices for the auxiliary data structures provide some useful performance trade-offs or improvements.

Acknowledgements.

We deeply appreciate the thorough and careful reviews by the anonymous referees, which greatly improved the clarity and quality of the presentation in the paper.

REFERENCES

1. J. L. Bentley and J. B. Saxe, *Decomposable searching problems* I: *Static-to-dynamic transformations*, J. of Algorithms 1, 1980, pp. 571–577.
2. I. G. Gowda and D. G. Kirkpatrick, *Exploiting linear merging and extra storage in the maintenance of fully dynamic geometric data structures*, Proc. 19th Annual Allerton Conf. on Communication, Control and Computing, 1980, pp. 1–10.
3. D. T. Lee and F. P. Preparata, *Computational geometry – a survey*, IEEE Trans. Computers C-33 (12), Dec. 1985, pp. 1072–1101.
4. K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, 1984.
5. S. V. N. Rao, S. S. Iyengar and C. E. V. Madhavan, *A comparative study of multiple attribute tree and inverted file structures for large bibliographic files*, Information Process. and Mgmt. 21, (5), 1985, pp. 433–442.
6. M. H. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science No. 156, Springer-Verlag, 1984.
7. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1985.