JOURNAL OF COMPUTER AND SYSTEM SCIENCES 21, 236–250 (1980)

Implicit Data Structures for Fast Search and Update*

J. IAN MUNRO AND HENDRA SUWANDA

Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

Received October 15, 1979; revised May 6, 1980

We consider representations of data structures in which the relative ordering of the values stored is ***implicit*** in the pattern in which the elements are retained, rather than explicit in pointers. Several implicit schemes for storing data are introduced to permit efficient implementation of the instructions *insert*, *delete* and *search*. $\theta(N^{1/2})$ basic operations are shown to be necessary and sufficient, in the worst case, to perform these instructions provided that the data elements are kept in some fixed partial order. We demonstrate, however, that the upper bound can be reduced to $O(N^{1/3} \log N)$ if arrangements other than fixed partial orders are used.

1. INTRODUCTION

Pointers are often used to indicate partial orderings among the keys in a data structure. While their use is often crucial to the flexibility and efficiency of algorithms, their explicit representation often contributes heavily to the space requirement. In this paper our interest is in structures (and algorithms acting on them) in which structural information is implicit in the way data are stored, rather than explicit in pointers. Thus, only a simple array is needed for the data.

The classic example of such an implicit data structure is the heap [7]. A heap, containing N elements from some totally ordered set, is stored as a one dimensional array; $A[1 : : N]$. A partial order on the elements of the array is maintained so that $A[i] \leqslant A[2i]$ and $A[i] \leqslant A[2i + 1]$ (see Fig. 1). This implicit representation of a tree permits the minimum element to be found immediately and a new element to be inserted in $O(\log N)$ steps.[1] Similarly, an element in a specified position may be deleted in $O(\log N)$ steps. Furthermore, the structure has the delightful dynamic property that no wholesale (global) restructuring is required as the number of elements being stored changes. Unfortunately, a heap is a very bad representation if searches for arbitrary elements are to be performed; indeed, these operations require $\theta(N)$ comparisons.

Another example of an implicit structure is a sorted list. We can view a sorted list as being constructed by storing the median of the elements in the middle of the array,

* This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant A8237. A preliminary version of this paper was presented to the 11th Annual ACM Symposium on Theory of Computing, Atlanta, Georgia, May 1979.

[1] All logarithms are to base 2 unless otherwise stated.

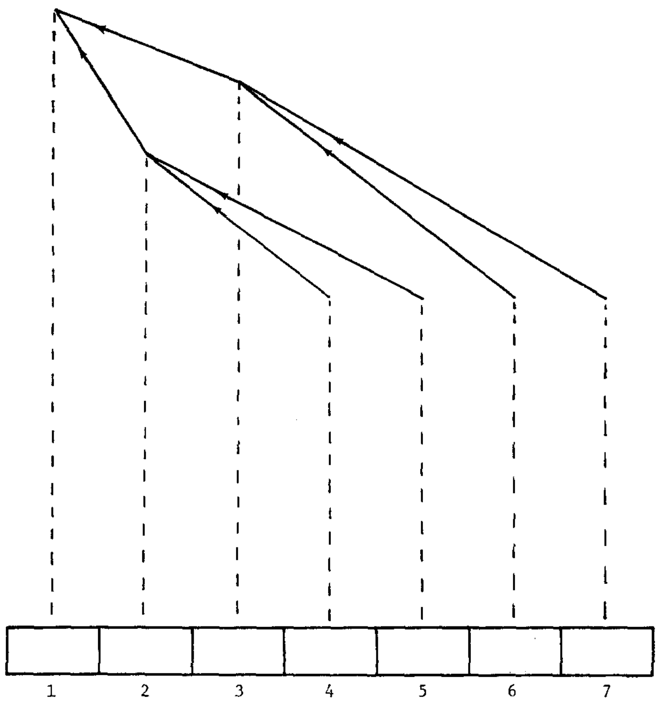

Fig. 1. A heap viewed as an implicit structure.

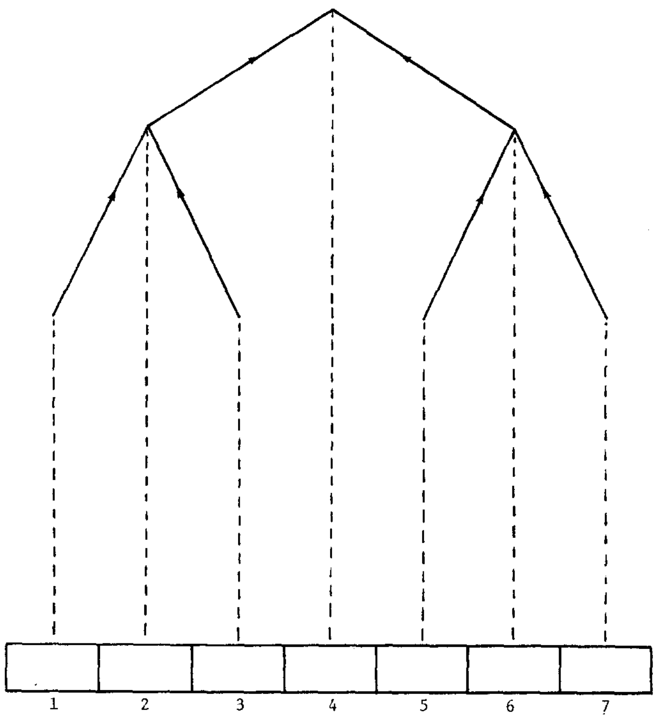

Fig. 2. A sorted list viewed as an implicit structure for performing binary search.

partitioning the elements into two groups (those smaller and those larger than the median) and repeating the same process with the smaller elements in the left part of the array and the larger ones on the right. The implicit information in this array is a binary tree corresponding to the process of the binary search (see Fig. 2). In contrast to the heap, searching an arbitrary element can be done in $O(\log N)$ steps, but an insertion or a deletion may need $\theta(N)$ steps (moves) to restructure the array.

Our major goal in this paper is to develop pointer free data structures under which the basic operations of *insert*, *delete* and *search* can be performed reasonably efficiently. We are also able to establish lower bounds on the relative costs of these operations under a reasonable model. In developing these lower bounds, it is convenient to think of an insertion and a deletion as being paired to form a *change* of a data value.

2. The Model of Computation

Our basic interest is in dynamic data structures, in particular ones in which N, the number of data items, will change with time. The model is a (potentially infinite) one dimensional array the first N locations of which contain the data items. We will draw no formal distinction between a pointer and an integer (index) in the range $[0, N]$. A data structure is then *implicit*, if the only such integer which need be retained is N itself. Most, though not all, of our attention will be devoted to such structures, although we note that one might prefer to permit a constant number of pointers to be retained and still designate the structure as implicit. We will also suggest two structures which might be described as "semi-implicit," in that a variable, but $o(N)$, number of pointers (indices) is kept. Our basic operations on data elements are making comparisons between two elements (with three possible outcomes $<$, $=$ and $>$) and moving elements. Arithmetics are allowed only for manipulating the indices.

Our measure of complexity is the maximum number of comparisons and moves required to perform any of the operations on the data structure. This worst-case analysis of single operations is in contrast with a closely related problem considered by Bentley *et al.* [1]. They store N data elements in N consecutive storage locations which are viewed as $\lceil \log(N+1) \rceil$ sorted lists. List i is of length 2^{i-1} if the ith most significant bit of the binary representation of N is 1, and of length 0 otherwise. A search is easily made on this structure in $O((\log N)^2)$ time by doing $\lceil \log(N+1) \rceil$ binary searches. An insertion is made by adding the new element to the end of the list and doing whatever merges are necessary to reconstitute the structure. They demonstrated that $O(N(\log N)^2)$ comparisons and moves are sufficient to perform any sequence of N insertions and searches when this scheme is employed. That is, in a very strong sense, an average of $O((\log N)^2)$ steps are sufficient to perform an insertion or a search on a list of N elements. They also show that this bound is within a constant factor of being optimal for their problem, provided the only reordering used is the merging of pairs of sorted sequences of elements, and that the cost of performing such a merge is equal to the sum of the lengths of the sequences. A method of deleting elements is also demonstrated, but this is at the cost of an extra bit of storage per data item and a small increase in run-time.

3. An Implicit Structure Based on a Partial Ordering

The ordering scheme presented by Bentley *et al.* is, like the heap and sorted list, a fixed partial order (for each N) imposed on the locations of the array. The elements occupying the locations must be consistent with the partial order. The more restrictive, in the sense of permitting fewer total orders, the partial order is, the easier it seems to be to perform searches, but the harder it appears to make changes to the structure (because of the many relations that must be maintained). The heap and the sorted list are two rather extreme examples demonstrating the imbalance between the cost for searching and the cost for modification. In this section we present an ordering scheme which balances these costs.

3.1. *The Biparental Heap or Beap*

As we have noted, the heap is a very poor structure upon which to perform searches for random elements. One interpretation of the reason for this difficulty is that as a result of each internal node having two sons, there are too many ($N/2$) incomparable elements in the system. The other extreme, each element having only one son, is a sorted list and so is difficult to update. Our first compromise between costs of searching and updating is to create a structure with the same father-son relationship as a heap, and with most nodes having two sons. The difference is that in general each node will have two parents. This left us with the cumbersome term, ***biparental heap***, which, fortunately, was shortened to ***beap*** by Ed Robertson.

To form this structure, the array is partitioned into roughly $(2N)^{1/2}$ blocks. The ith block consists of the i elements stored from position $(i(i-1)/2+1)$ through position $i(i+1)/2$. This enables us to increase or decrease the size of the entire structure while changing the number of elements in only one block. Indeed, this and similar blocking methods are used in all of the structures we present. The ordering imposed on this structure is that the kth element of the jth block is less than (or equal to if a multiset is to be stored) both the kth and the $k+1$st elements of the block $j+1$. This ordering is illustrated in Fig. 3. The numbers in Fig. 3a denotes the indices of the array and an arrow may be viewed either as a pointer or as $a \leqslant$ relation. The structure is then analogous to a heap; however, an element in our structure will typically have two parents, and so the height of the structure is about $(2N)^{1/2}$.

Taking a slightly different point of view, one can interpret this structure as an upper triangular of a matrix (or grid) in which locations 1, 2, 4, 7 $\cdots$ form the first column and 1, 3, 6, 10 $\cdots$ the first row. Furthermore, each row and each column are maintained in sorted order. Under this interpretation, the element in position (i, j) of the triangular grid is actually stored in location $P(i, j) = ((i+j-2)^2 + i + 3j - 2)/2$ of the array. We are, in fact, simply using the well-known diagonal pairing function. In the interest of brevity we will present several of our algorithms in terms of moving along rows and columns. Note that these manipulations can easily be performed on the structure without the awkward repeated computation of the inverses of $P(i, j)$, and in general without the explicit evaluation of $P(i, j)$ at each node visited. This structure is easily created by

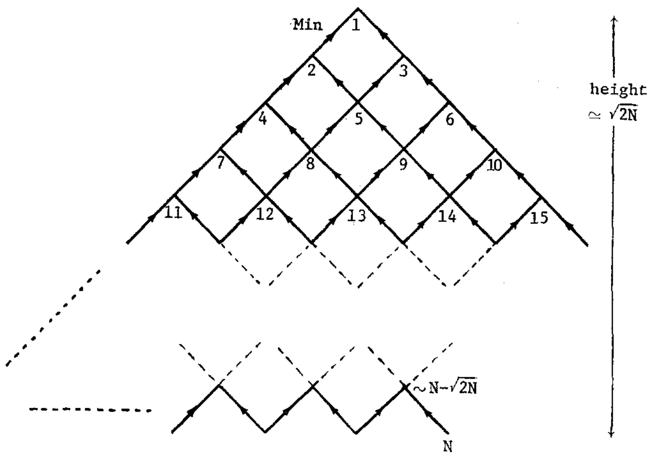

FIG. 3a. A beap.

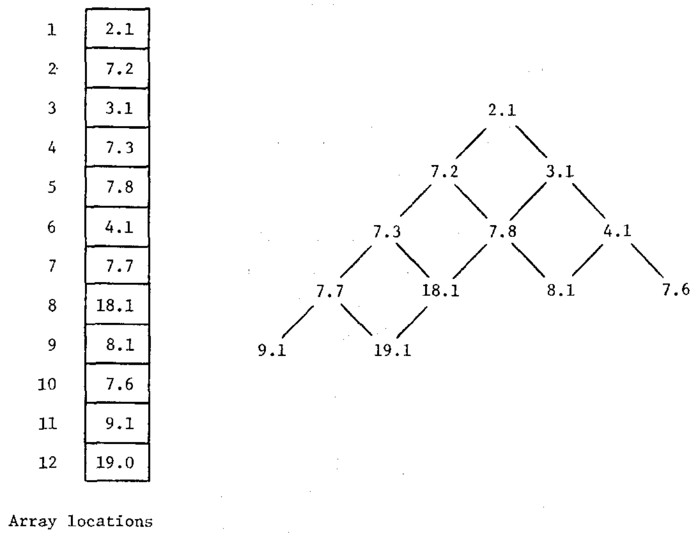

FIG. 3b. An example of a beap with 12 elements showing storage mapping and implicit structure.

sorting the entire list. In [6] it is shown that there are about $(N!)^{1/2}$ beaps on an ordered set of N elements. This implies that $\log(N!/(N!)^{1/2} \simeq (N/2) \log N$ comparisons are necessary, on the average, to create a beap, and so $\theta(N \log N)$ are necessary and sufficient. We now describe methods for performing a few basic operations on this structure.

1. Finding the minimum: This element is in the first location.

2. Finding the maximum: The maximum is in one of the last $(2N)^{1/2}$ locations.

3. Insertion: Insertion is performed in a manner analogous to insertion into a heap. A new element is inserted into location $(N + 1)$ of the array. If this element is smaller

than either of its parents, it is interchanged with the larger parent. This sifting-up process is continued until the element is in a position such that it is larger than both of its parents. Since the height of the structure is $(2N)^{1/2} + O(1)$, one sees that at most 2 $(2N)^{1/2} + O(1)$ comparisons and $(2N)^{1/2} + O(1)$ moves are performed. Furthermore, we note that if we are to insert into the structure an element which is smaller than all elements currently stored, then every element on some "parent-offspring" path from location 1 to one of the last $(2N)^{1/2}$ locations must be moved. This condition holds regardless of the insertion scheme employed. Hence the scheme outlined minimizes the maximum number of moves performed in making an insertion into a beap.

4. Deletion: Once we have determined the location of the element to be removed, simply move the element in position N of the array to that location. This element then filters up or down in essentially the manner used for insertions. Thus, the cost for a deletion is at most $2(2N)^{1/2} + O(1)$ comparisons and $(2N)^{1/2} + O(1)$ moves.

5. Search: For this operation, it is convenient to view the structure as an "upper-left" triangular matrix (see Fig. 4). We start searching for an element, say x, at the top right corner of the matrix. After comparing x with the element under consideration, we will do one of the following depending upon the outcome of the comparison.

(i) If x is less than the element under consideration, move left one position along the row.

(ii) If x exceeds the element, either move down one position along the column or if this is not possible (because we are on the diagonal) then move left and down one position each.

(iii) If the element is equal to x, stop searching.

Repeating the above process, the search path will eventually terminate successfully or meet with the left side of the triangle and be unable to move. The latter condition indi-

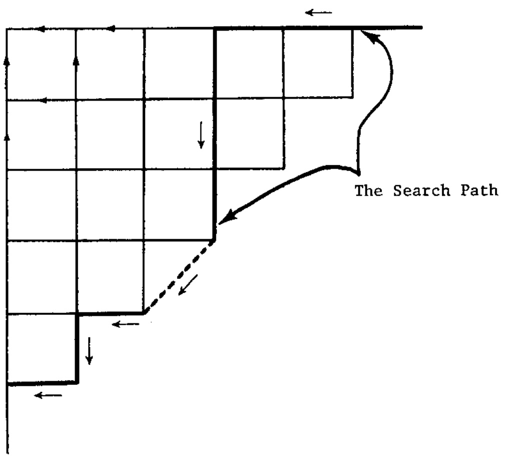

FIG. 4. A search path through a beap.

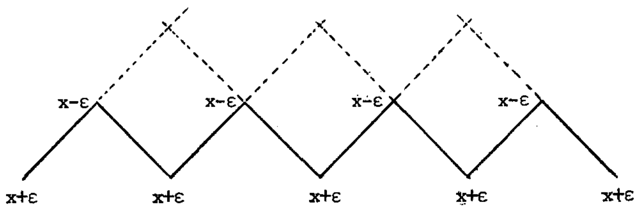

FIG. 5. Diagonal and superdiagonal containing elements near x.

cates an unsuccessful search. Thus, the cost for a search is at most $2(2N)^{1/2} + O(1)$ comparisons.

In summary, we have demonstrated the following:

THEOREM 3.1.1. *Storing N data elements in an array of length N as a beap and retaining no additional information other than the value N, it is possible to maintain a data structure under which insertions and deletions can be performed in* $2(2N)^{1/2} + O(1)$ *comparisons and* $(2N)^{1/2} + O(1)$ *moves, and searches in* $2(2N)^{1/2} + O(1)$ *comparisons.*

It is worthy of note that searching on this structure cannot be done with fewer comparisons.

THEOREM 3.1.2. $2(2N)^{1/2} + O(1)$ *comparisons are necessary and sufficient to search for an element in a beap.*

Proof. Consider the diagonal and super-diagonal (the row above the diagonal) of the structure (Fig. 5). Suppose the diagonal contains the largest elements in the structure and the super-diagonal contains elements smaller than those on diagonal but larger than any others in the rest of the system. No other information about the relative values of the elements is assumed. Suppose, then, that we are to search for an element known to be smaller than all (except perhaps one) of the elements on the diagonal, but larger than all except perhaps one) on the superdiagonal. There is no choice but to inspect all elements in both of these blocks. The theorem follows by noting the number of elements on a diagonal. ∎

4. LOWER BOUNDS

Snyder [5] has shown that if the representation of a data structure on N elements is "unique," then at least one of the operations *insert*, *delete* or *search* requires $\Omega(N^{1/2})$ comparisons or alterations. Snyder's model, however, differs from ours in that he assumes the use of explicit pointers in his representation. This implies, for example, that if the maximum element in a sorted list is to be replaced by one smaller than any of the elements in the list, only two pointer changes have to be made to return to the original form. Under an implicit ordering, every element would have to be moved to preserve the sorted order. In this sense, Snyder's lower bound can be construed as somewhat stronger than necessary for our purposes. On the other hand, he assumes the structure is effectively stored as a tree of bounded degree. This precludes some of the index calculations which we would rather permit, and in that sense his lower bounds are too weak. We can, however, demonstrate the same lower bound as Snyder's $\Omega(N^{1/2})$, for the class of implicit data structures which are based solely on storing the data in some fixed partial order. Although this result sounds very similar to Snyder's, the two are incomparable; and furthermore, the proof techniques are quite different. Observe that the structures discussed in the preceding sections are based on a fixed partial order.

For simplicity let us assume that we are to perform the basic operations of search and change (i.e., deletion followed by an insertion) on our structure. We insist on pairing a deletion with an insertion only to eliminate the problem of the structure changing its size.

Let $\mathcal{S}$ denote the maximum number of comparisons necessary to search for an element and $\mathcal{C}$ denote the number of locations from which data must be moved to perform a change.

THEOREM 4.1. *If an implicit data structure containing N elements carries no structural information other than a fixed partial order on the values stored in various locations, then* $\mathcal{S} * \mathcal{C} \geqslant N$.

Proof. Consider the directed acyclic graph whose nodes correspond to the locations in a structure, and edges correspond to orderings between elements in our locations as specified by the partial order underlying the storage scheme. Let S be the largest antichain this graph. It is quite possible that the elements stored in the locations corresponding to S are of consecutive ranks among the elements stored in the structure. Hence in searching for any of these elements, no comparisons with any elements outside S can remove from consideration any in S. Therefore, the number of elements in S is a lower bound and the number of comparisons necessary to perform a search on the structure.

Now suppose the elements of the longest chain, C, are as small relative to the other elements in the structure as is consistent with the partial order. That is, if there are k elements which must be smaller than a given element in the chain, then the particular element is the $k + 1$st smallest in the structure. Finally, assume we are to replace the smallest element in C (which is the smallest element in the structure) with one greater than the largest element in C. This implies that each element in the chain is in a position requiring more specific locations be occupied by elements smaller than the given element than there exist in the entire structure. Hence, every element on the given chain must be moved (including the removing of the minimum element). The theorem now follows since the product of the length of the longest chain and the size of the largest antichain must be at least N (see, for example, [3]). ▌

COROLLARY 4.2. *A total of* $\theta(N^{1/2})$ *moves and comparisons are necessary and sufficient to perform the operations* insert, delete *and* search *on an implicit data structure in which the only information known about the structure is a fixed partial order on the array locations and the size of the structure.*

Proof. Follows from Theorems 4.1 and 3.1.1. ▮

Consider a partial order on N elements which has $\mathscr{T}$ linear extensions (i.e., there are $\mathscr{T}$ total orders consistent with the partial order). Then $\mathscr{P}$, the information theory lower bound on the number of comparisons necessary to construct this partial order, is $\log(N!/\mathscr{T}) = N \log N - \log \mathscr{T} - O(N)$. Suppose a search can be performed on a sequence of elements arranged according to this partial order using at most $\mathscr{S}$ comparisons, then Borodin *et al.* [2] show:

LEMMA 4.3. $\mathscr{P} + N \log \mathscr{S} \geqslant (1 + o(1))\, N \log N$.

Combining this with Theorem 4.1 ($\log \mathscr{S} + \log \mathscr{C} \geqslant \log N$), it follows that $\mathscr{P} + 2N \log \mathscr{S} + N \log \mathscr{C} \geqslant (2 + o(1))\, N \log N$. By substituting for $\mathscr{P}$ and observing that, in the proof of Lemma 4.3, the $o(1)$ term is large enough so that the $O(N)$ term in the expression for may be dropped, an interesting combinatorial view of the interplay between the various cost criteria is obtained.

COROLLARY 4.4. $2N \log \mathscr{S} + N \log \mathscr{C} - \log \mathscr{T} \geqslant (1 + o(1))\, N \log N$.

It is, perhaps, worth noting that the beap (a 2-dimensional grid) gives the best balance we can get between the cost for searching and the cost for insertion/deletion. By going to a one dimensional grid, which is a sorted list, insertion and deletion will become more expensive and searching will become cheaper. Going to the obvious extension in three or more dimensions reduces the cost of modification at the expense of retrieval cost.

5. On Structures Not Using a Fixed Partial Order

In this section, we present several implicit (and "semi-implicit") structures under which the product of search time and insert (or delete) time is less than N. The main trick employed is to store blocks of elements in an arbitrary cyclic shift of sorted order. We call such a block a *rotated list*. An example of a rotated list is given in Fig. 6. All techniques presented in this section are based on the assumption that all elements in the structure at the same time are distinct.

5.1. *Rotated Lists*

Again the array is partitioned into blocks such that the ith block contains i elements. The requirements for the order maintained are much more stringent than those of the beap. We insist that

(i) all elements in block i be less than or equal to all elements in block $i + 1$;
(ii) block i is a rotated list of size i.

An example is shown in Fig. 7.

Before discussing other searches we must describe a technique for finding the minimum of a rotated list containing *distinct* elements. The method used is a bisection technique.

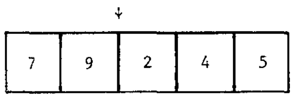

FIG. 6. A rotated list of size 5.

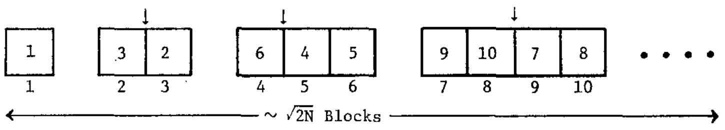

FIG. 7. Each block stored as a rotated list.

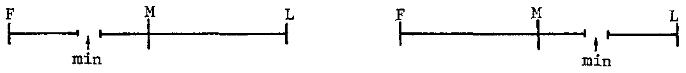

FIG. 8. Finding the minimum in a rotated list.

Figure 8 illustrates the problem of determining the minimum value in the interval $[F, L]$. The search procedure is described below:

If $F = L$ then the minimum is found. Otherwise, $M := \lfloor (F + L)/2 \rfloor$

if element $(M) <$ element (L) then min is in $[F, M]$; apply the procedure recursively.

if element $(M) >$ element (L) then min is in $[M + 1, L]$; apply the procedure recursively.

It can be shown that this process requires at most $\lceil \log i \rceil$ comparisons if i is the size of the block. We note a limitation of our technique in that if elements are repeated, this search procedure for a block minimum may not succeed. Indeed if all but one of the elements of a block are the same, then only an exhaustive search will find the minimum.

We now describe methods for performing basic operations on such structures:

1. Finding the minimum: Again this element is in the first array location.

2. Finding the maximum: The maximum is in one of the locations in the last block. By using the above search procedure, we can find the minimum element of that block; the maximum is in the immediately preceding location (or in location N, if the block minimum is in the first location in the block). Hence only $\lceil \log((2N)^{1/2}) \rceil + O(1)$ comparisons are required.

3. Search: Our basic approach is to perform a simple binary search until it is determined that the desired element lies in one of two consecutive blocks. Next the above search procedure (based on bisection technique) is performed to determine the minimum element in the larger block. Based on this outcome of a comparison between this minimum and the desired element, we either

(i) perform a (cyclicly shifted) binary search on the larger of the two candidate blocks or

(ii) determine the position of the minimum element in the smaller block, and perform a binary search on that block.

An alternate search technique with the same worst-case behavior is to first do a binary search on the blocks to find two consecutive blocks that may contain the desired element and then proceed as before.

LEMMA 5.1.1. *In the worst case, searching requires at most* $2 \log N + O(1)$ *comparisons.*

Proof. Let k and $k+1$ be the sizes of the two consecutive blocks. The number of comparisons in finding these two blocks is at most $\lceil \log N - \log k \rceil$. Locating the minimum value in the larger block requires at most $\lceil \log(k+1) \rceil$ comparisons. Completing the search requires at most $2 + \lceil \log k \rceil + \lceil \log(k+1) \rceil$ comparisons. Thus, the entire process costs at most $\log N + 2 \log k + O(1)$ comparisons. This again is bounded by $2 \log N + O(1)$ comparisons. ∎

4. Insertion: Using the basic strategy suggested for performing a search, the block into which a new element should be inserted can be found in $\log N + O(1)$ comparisons. A further $(2N)^{1/2} + O(1)$ (at most) moves suffice to insert the new element into its proper position, remove the block maximum and shift the elements which lie between the new element and the former location of the block maximum. At this point, we see that for each block larger than the one in which the insertion was made, we must perform the operation of inserting a "new" minimum element and removing the old maximum. Fortunately, the new minimum can simply take the place of the old maximum and no further shifting (within blocks) is necessary. This transformation can be performed on a block of i elements in $\log i + O(1)$ comparisons and one swap. Thus, the entire task can be accomplished in about $(N/2)^{1/2} \log N + O(N^{1/2})$ comparisons and $O(N)^{1/2}$ moves.

5. Deletion: Deletions are performed in essentially the same way as that outlined for insertions.

Summarizing the above results and observing that once the structure is formed, only $O(N^{1/2} \log N)$ comparisons are necessary to complete a sort of the list, we have shown:

THEOREM 5.1.2. *Assuming data items are distinct, there exists an implicit data structure which performs searches in 2 log N comparisons and insertions and deletions in* $(N/2)^{1/2} \log N + O(N^{1/2})$ *comparisons and moves.* $N \log N - O(N)$ *comparisons are necessary and sufficient to create this structure.*

At this point, it is natural to ask whether or not we can simultaneously achieve the $O(\log N)$ search time of the rotated sort structure and the $O(N^{1/2})$ insertion and deletion costs of the beap. In the next section we show that this is possible with $o(N)$ additional storage.

5.2. *Improving Insertion Time with Extra Storage*

5.2.1. *With* $(2N)^{1/2} + O(1)$ *Pointers*

Observe that the $\theta(N^{1/2} \log N)$ behavior of the above technique is due to the search, in each block, for the (local) maximum. By retaining a pointer to the maximum of each block, the insertion and deletion times are reduced to $O(N^{1/2})$.

5.2.2. *Batching the Updates with a Small Auxiliary Map*

Another approach is to "batch" insertions and deletions. This can be accomplished with k (for an arbitrary value of k) extra locations to store the pointers (indices) to the array and, with each of these k indices, a bit to indicate whether insertion or deletion is to be performed. $k \log N$ or so bits are required for this modification map; furthermore, it is possible that k extra locations in the array are used for elements which have already been deleted. The basic approach is simply to note the modifications to be made in the auxiliarly table; actually making the changes in the table is delayed. A search is made by scanning the main table and the modification map. We now sketch this general approach.

1. Deletion: Put a pointer (in the modification map) to the key to be deleted, assuming there is no pointer to it already. If the element is marked to be deleted, do no more. If it is a newly inserted element, and so has a pointer in the modification map, remove the element and adjust the map in the straightforward manner.

2. Insertion: Search the map to see if the "new" element is actually in the array but to be deleted. If this is the case, we just remove the corresponding entry in the map. Otherwise, we put the key in the next location, and keep a new pointer to this location in the modification map. Note we must retain both the size of the basic table and the location of the most recently added element.

3. Search: We search the map first, if there is an entry pointing to the desired element in the array, the answer will depend on the extra bit denoting a deletion or an insertion. Otherwise, we search on the structure by using methods described in the previous section. The cost will be $O(k + \log N)$ comparisons.

When the map is full, we sort the newly inserted and deleted keys and then, the changes are made in a single pass. This restructuring will require at most $O((k + N) \log N)$ operations, and so can be "time-shared" with the next k insert/delete commands, although the details of doing so are admittedly complicated. Clearly choosing k to be rather large reduces the cost of making changes at the expense of the search time. The judicious choice of $\log N$ as the value of k achieves our goal of $O(\log N)$ search time and $O(N^{1/2})$ to make an insertion or deletion.

At this point, one is also inclined to ask whether or not both the search and modification costs may reduce to below $\theta(N^{1/2})$. The answer is in fact positive. This can be achieved by combining the ideas of a beap and the rotated list as described below.

5.3. *A Beap of Rotated Lists*

In this section we will present a structure on N elements which allows us to perform a search, a deletion or an insertion in $O(N^{1/3} \log N)$ comparisons and swaps. Again the array is partitioned into blocks. The ith block is stored from location $(i - 1)i(2i - 1)/6 + 1$ through location $i(i + 1)(2i + 1)/6$, and is divided into i consecutive subblocks containing i elements each. The subblocks correspond to the elements (nodes) of the beap and each subblock is stored as a rotated list. Note that the height of the structure is about $(3N)^{1/3}$, which is equal to the number of blocks. More precisely,

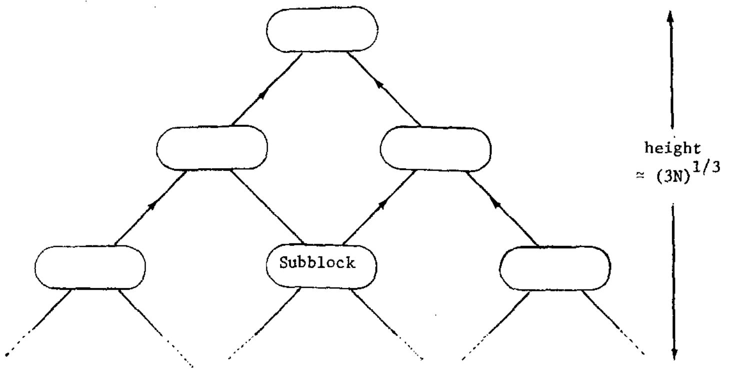

FIG. 9. A beap of rotated lists.

(i) each subblock of block i is a rotated list of size i;

(ii) all the elements of the kth subblock of the jth block are less then all elements of the kth and $k + 1$st subblocks of block $j + 1$.

A structural example of a beap of rotated lists is given in Fig. 9. As in the case of the beap, we note that moving along rows and columns from subblock to subblock can be done easily without computing the pairing function and its inverses, provided three or four parameters are kept. Now we describe how to perform the operations:

1. Finding the minimum: Again this element is in position 1.

2. Finding the maximum: The maximum is in one of the last $(3N)^{1/3}$ subblocks. The maximum element in a subblock can be found in $(\log N)/3 + O(1)$ comparisons, and so the maximum elements in the entire structure can be found by searching each of these subblocks in a total of $(N/9)^{1/3} \log N + O(N^{1/3})$ comparisons.

3. Insertion: By combining methods for the beap and the rotated lists, we insert the new element into the $N + 1$st position of the array, which is part of a subblock. Since the subblock is cyclicly sorted, about $2/3 \log N$ comparisons and $(3N)^{1/3}$ swaps are required in the worst case to insert the new element into the subblock. As in the beap, if the new element is smaller than either of the maximum elements of its parents, it is interchanged with the larger one. This process continues (as in the beap) until the imposed ordering is restored. Since the height of the structure is about $(3N)^{1/3}$, $O(N^{1/3} \log N)$ comparisons and swaps are in fact used.

4. Deletion: Similar to insertion.

5. Search: Basically searches are performed in the same manner as described for the beap. The key differences are that a comparison with a single element in the beap is replaced by a (modified) binary search to find the minimum (and hence the maximum) of a subblock, and so the decision to move left along the row or down along the column will need more comparisons. Again, we start searching for an element, say x, at the top

right corner subblock of the matrix. After finding the minimum and maximum of the subblock under consideration we will do the following:

(i) If x is less than the minimum element, move left one subblock along the row.

(ii) If x is larger than the maximum element, move down one subblock along the column, if this cannot be done (on diagonal) then move left and down one subblock each.

(iii) If x is in the range of this subblock, then do a binary search to find x. If successful, then stop searching; otherwise, move left and down one subblock each.

This process is repeated until either x is found or the required move cannot be made. In this manner a search can be performed in $O(N^{1/3} \log N)$ comparisons. Hence we have

THEOREM 5.3.1. *There is an implicit data structure through which it is possible to perform each of the operations insert, delete and search in* $O(N^{1/3} \log N)$ *comparisons and moves.*

We note that the easiest way to initialize the structure is to sort the N elements of the array. The cost of doing so is within a (small) constant factor of that of the optimal method.

6. Conclusion

We have drawn attention to implicit data structures as a class of representations worthy of study. A new application of this class has been demonstrated by using them to maintain structures in which insertions, deletion and searches can be performed reasonably efficiently. Table I summarizes the behaviour of algorithms acting on the implicit structures we have proposed.

TABLE 1
Costs of the Implicit Structures Discussed

Structure	Search	Insert/delete	Extra storage
Beap	$O(N)^{1/2}$	$O(N^{1/2})$	$\log N$ bits
Rotated lists	$O(\log N)$	$O(N^{1/2} \log N)$	$\log N$ bits
Rotated lists with block pointers	$O(\log N)$	$O(N^{1/2})$	$O(N^{1/2} \log N)$ bits
Rotated lists with auxiliary table	$O(\log N)$	$O(N^{1/2})$	$O((\log N)^2)$ bits
Beap of rotated lists	$O(N^{1/3} \log N)$	$O(N^{1/3} \log N)$	$\log N$ bits

Acknowlegments

The authors are pleased to thank Kelly Booth, Frank Tompa and the referees for many helpful suggestions on clarifying the presentation of this paper.

Note added in proof. Greg Frederickson has extended the notion of rotated lists to construct an implicit structure permitting searches in $O(\log N)$ time and updates in $o(N^\epsilon)$ time. ("Proc. 21st IEEE Symp. on Foundations of Computer Science," pp. 255–259).

References

1. J. L. Bentley, D. Detig, L. Guibas, and J. Saxe, An optimal data structure for minimal storage dynamic member searching, unpublished manuscript.
2. A. B. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao, Efficient searching via partial ordering, unpublished manuscript. April 1979.
3. J. A. Bondy and U. S. R. Murty, "Graph Theory with Applications," American Elsevier, New York, 1976.
4. D. E. Knuth, "The Art of Computer Programming," Vol. III, "Sorting and Searching," Addison–Wesley, Don Mills, Ontario, 1973.
5. L. Snyder, On uniquely representable data structures, *in* "Proceedings, 18th IEEE Symposium on Foundations of Computer Science, 1977," pp. 142–146.
6. H. Suwanda, "Implicit Data Structures for the Dictionary Problem," Ph.D. Thesis, Dept. of Computer Science, University of Waterloo. (Also as Research Report CS-80-04.)
7. J. W. J. Williams, "Algorithm 232: Heapsort," *Comm. Assoc. Comput. Mach.* **7** (1964), 347–348.

Printed in Belgium