IMPLICIT DATA STRUCTURES FOR FAST SEARCH AND UPDATE*

J. Ian Munro
and
Hendra Suwanda

Research Report CS-79-31

Computer Science Dept.
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Abstract

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. $O(N)$ 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.

---

* This work was supported by the National Science and Engineering Research Council of Canada under grant A8237.

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 Figure 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. 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.

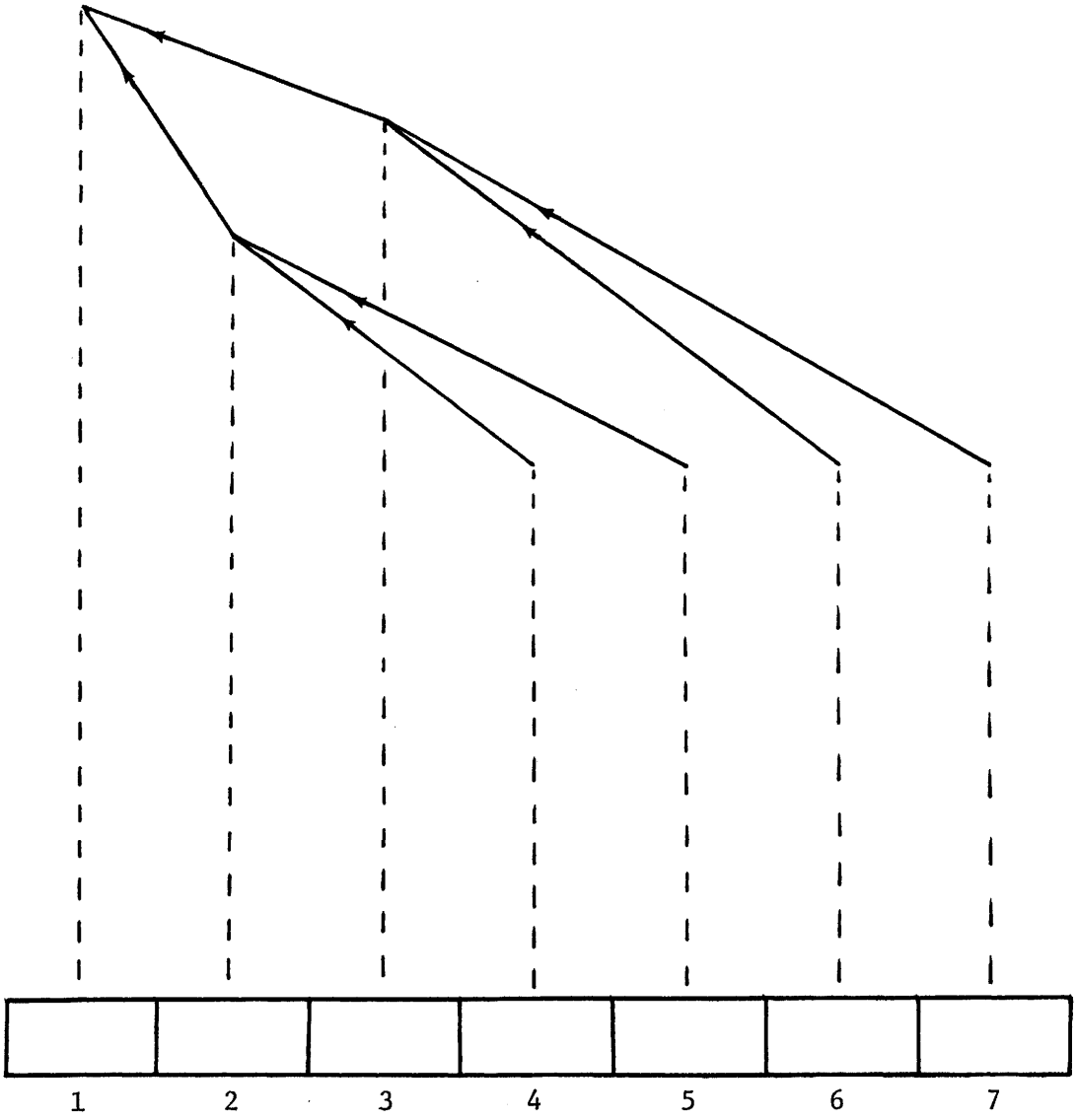

Figure 1: A Heap Viewed as an Implicit Structure

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, 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 Figure 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.

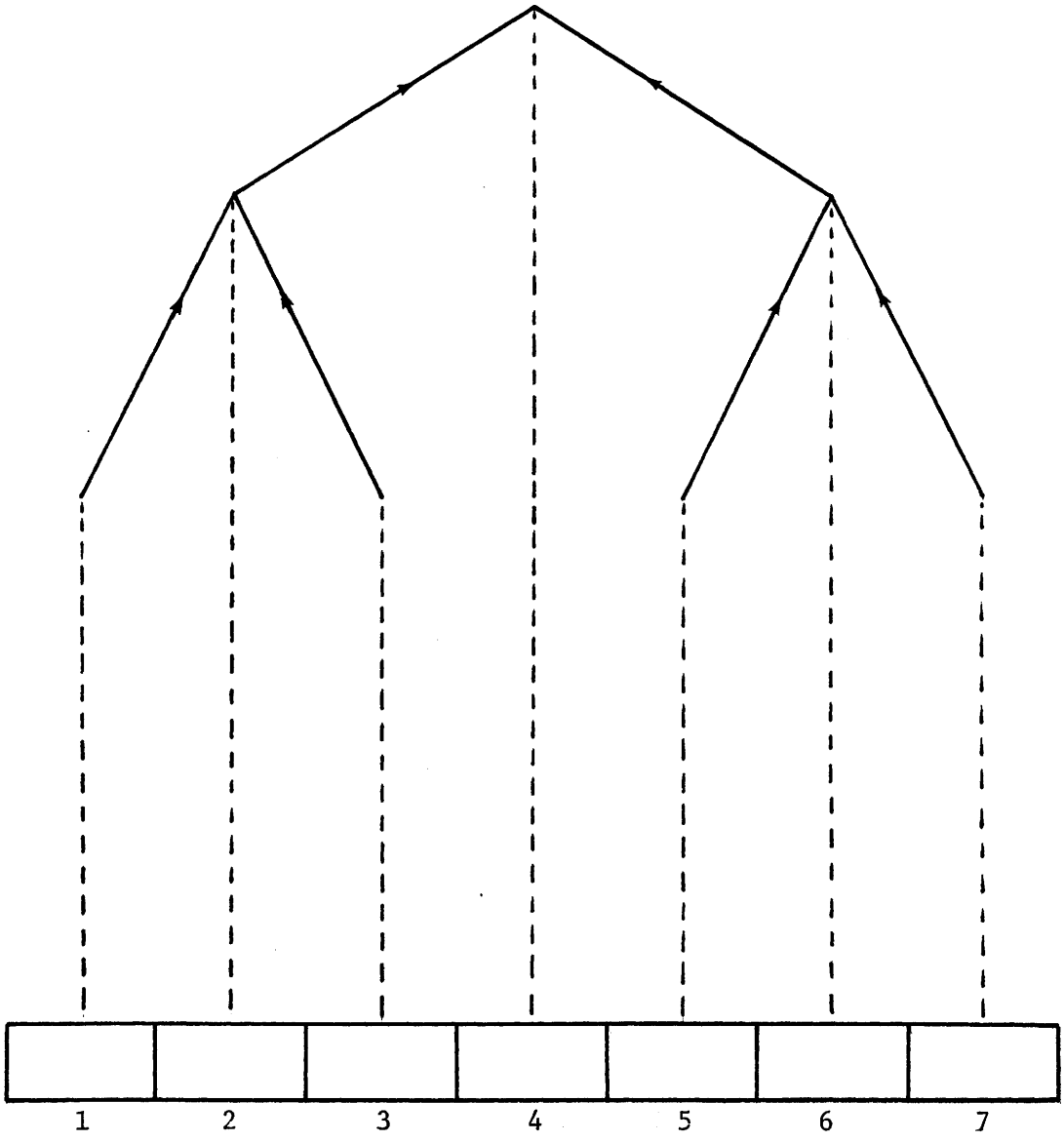

Figure 2: A Sorted List Viewed as an Implicit Structure for Performing Binary Search

2. The Model of Computation

Our model is a (potentially infinite) one dimensional array in which data are stored contiguously even after deletions have been made. 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 only a constant number of such integers need to be retained (O(log N) bits). Most, though not all, of our attention will deal with structures in which N, the current number of elements in the array (structure), is the only such value required. 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 swapping pairs of elements. Arithmetics are allowed only for manipulating the indices.

Our measure of complexity is the maximum number of comparisons and swaps required to perform the operations on the data structure. This worst-case analysis is in contrast with a closely related problem considered by Bentley et al [1]. They demonstrated that $O(N \log^2 N)$ comparisons and swaps are sufficient to perform any sequence of N insertions and searches when no explicit pointers are kept. That is, in a very strong sense, an average of $O(\log^2 N)$ 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 their 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. It is not hard to see that the more restrictive the partial order is, the easier it will be to perform searches, but the harder it will be 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

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 in that in general each node will have two parents.

To form this structure the array is partitioned into roughly $\sqrt{2N}$ blocks. The i^{th} 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 k^{th} element of the j^{th} block is less than (or equal to if a multiset is to be stored) both the k^{th} and the $k+1^{st}$ elements of the block j+1. This ordering is illustrated in Figure 3. The numbers in the figure denote the indices of the array and an arrow may be viewed either as a pointer or as a < 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 $\sqrt{2N}$.

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 ... form the first column and 1, 3, 6, 10 ... 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 $P(i,j) = 1/2((i+j-2)^2 + i + 3j - 2)$ 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

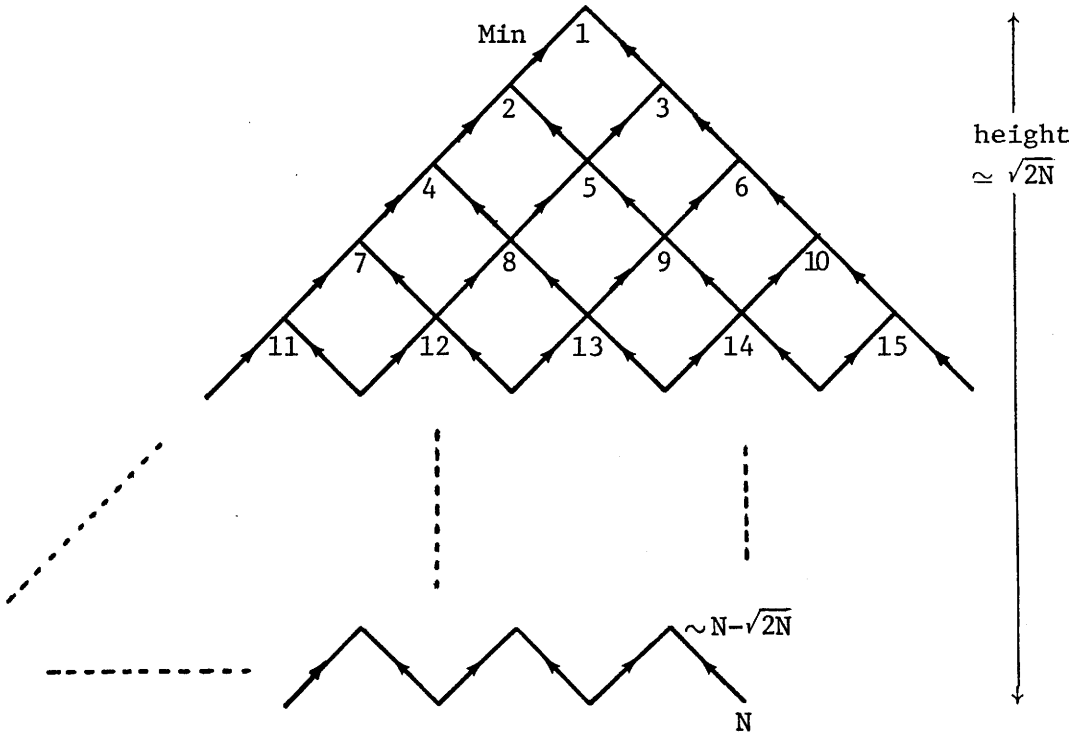

Figure 3: Biparental Heap

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 sorting the entire list. It is shown in [6] that (N/2) log N comparisons are necessary for its creation, and so Θ(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 $\sqrt{2N}$ 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 roughly $\sqrt{2N}$, one sees that at most $2\sqrt{2N}$ comparisons and $\sqrt{2N}$ swaps are performed. Furthermore, we note that if we are to insert a new element into the structure which is smaller than all elements currently

stored, then every element on some "parent-offspring" path from location 1 to one of the last $\sqrt{2N}$ 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 biparental heap.

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 $2\sqrt{2N}$ comparisons and $\sqrt{2N}$ swaps.

5. Search:

For this operation, it is convenient to view the structure as an "upper-left" triangular matrix (see Figure 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 the element is too large, move left one position along the row.

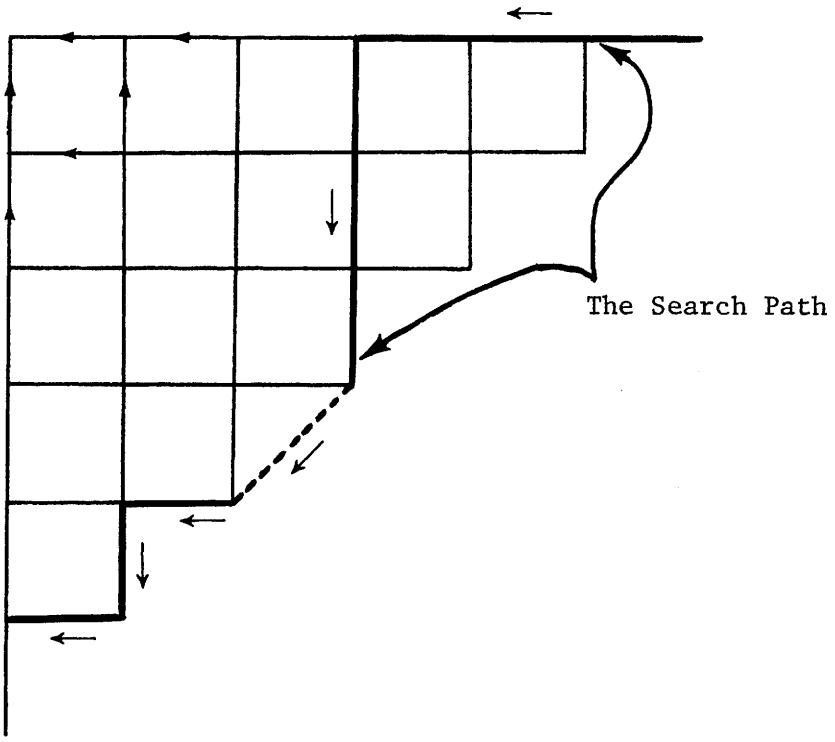

Figure 4: A Search Path through a Biparental Heap

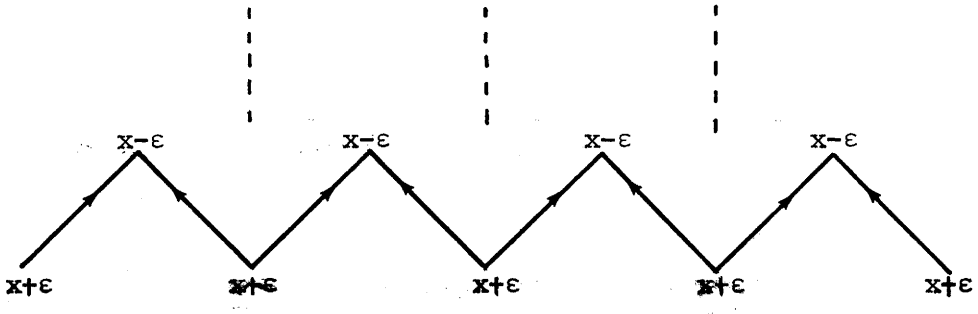

Figure 5: Diagonal and Superdiagonal Containing Elements Near x.

(ii) If the element is too small, 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 indicates an unsuccessful search. Thus, the cost for a search is at most $2\sqrt{2N}-1$ comparisons.

The following lemma shows that, in fact, this is the best we can do on this structure for a search.

Lemma 3.1.1:

$2\sqrt{2N}-1$ comparisons are necessary to search for an element in a biparental heap.

Proof:

Consider the diagonal and super-diagonal of the structure (Figure 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. Note that no other information about the relative values of the elements need be known. 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. □

In summary, we have demonstrated the following:

Theorem 3.1.2:

Storing N data elements in an array of length N as a biparental heap 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 $2\sqrt{2N}$ comparisons and $\sqrt{2N}$ swaps, and searches in $2\sqrt{2N}-1$ comparisons.

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 $\Omega(\sqrt{N})$ comparisons or "changes". Snyder's model, however, differs somewhat 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 this property. 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, and in that sense his lower bounds are too weak. We can, however, demonstrate the same lower bound as Snyder's $\Omega(\sqrt{N})$, for the class of implicit data structures which are based solely on storing the data in some fixed partial order. This result is, then, related to but incomparable with Snyder's. 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 follows 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.

Theorem 4.1:

If the only information retained about an implicit data structure, other than N, the number of elements it contains, is a fixed partial order on the locations of the array. Then, the product of the maximum number of comparisons necessary to search for an element and the number of locations from which data must be moved (swaps) to perform a change is at least 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 independent set in 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. This implies that 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. Now suppose 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 which requires it to have more elements which precede it be in specific locations than there are to be elements preceding it 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 independent set must be at least N (see for example, [3]). □

Corollary 4.2

$\Theta(\sqrt{N})$ swaps or comparisons are necessary and sufficient to perform the operations insert , delete and search on an implicit data structure in which the only information 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 3.1.2. □

A related result on lower bound has been obtained by Borodin et al [2]. Let P(N) be the number of comparisons necessary to build a partial order, such that a search can be performed in at most S(N) comparisons. They show

Lemma 4.3:

$$P(N) + N \log(S(N)) \geq (1 + o(1))\, N \log N$$

Since P(N) is basically N log N minus the logarithm (base 2) of the number of total orders consistent with the partial order on N elements (#(N)), we have a relation between this number and the search cost. Combining this lemma and our theorem, and letting DI(N) denote the number of moves required to make an insert/delete pair on a structure containing N elements, we obtain:

Corollary 4.4

$$2N \log(S(N)) + N \log(DI(N)) - \log(\#(N)) \geq (1+o(1))\, N \log N.$$

Proof:

By Theorem 4.1, we get $S(N) * DI(N) \geq N$, or $\log(S(N)) + \log(DI(N)) \geq \log N$. The corollary now follows from lemma 4.3. □

It is perhaps worth noting that the triangular grid (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 ex-

pensive and searching will become cheaper. Going to 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 "nearly 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.

5.1 A Simple Structure

Again the array is partitioned into blocks such that the i^{th} block contains i elements. The order maintained is much more stringent than that of the biparental heap. We insist that

(i) all elements in block i be less than or equal to all elements in block i+1;

(ii) the elements in each block be stored in a cyclic shift of increasing order.

By condition (ii) we mean that for some r (< i), the r largest elements of block i are stored, in increasing order, in the first r locations of the block. The (i-r) smallest elements are stored (in increasing order) in the last (i-r) location of the block. An example is shown in Figure 6.

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

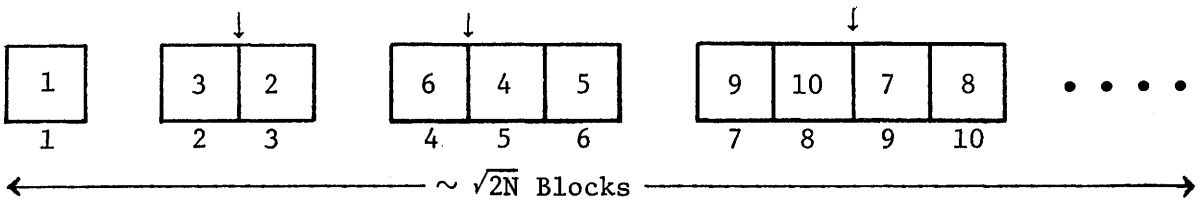

Figure 6: Each Block Stored in a Cyclic Shift of Increasing Order

Figure 7: Finding the Minimum in a Cyclicly Shifted Sorted List

1. Finding the minimum:

Again this element is in first array location.

Before discussing other searches we must describe a technique for finding the minimum of a block which is in cyclicly sorted order. The method is a modified binary search. Figure 7 illustrates the problem of determining the minimum value in the interval [F,L].

The search procedure is described below:

If [F,L] contains only two elements, the minimum is found by a simple comparison. 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,L]; apply the procedure recursively.

It can be shown that this process requires at most $\lceil \log(i-1) \rceil + 1$ comparisons if i is the size of the block.

2. Finding the maximum:

The maximum is in one of the locations in the last block. By using the modified binary search, we can find the minimum element of that block; the maximum is in the immediately preceding location (or in location N, if the

block maximum is in the first location in the block). Hence only $\lceil \log(\sqrt{2N-1}) \rceil + 1$ 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 a modified binary search 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.

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 $(\log N - \log k)$. Locating the minimum value in the larger block requires at most $\log k - 1$ comparisons. Completing the search requires at most

log k + 1 + log k comparisons. Thus, the entire process costs at most log N + 2 log k + 2 comparisons. This again is bounded by 2 log N + 3 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 $\sqrt{2N}$ (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 $\sqrt{N/2}$ log N + $O(\sqrt{N})$ comparisons and $O(\sqrt{N})$ 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 $O(\sqrt{N} \log N)$ comparisons are necessary to complete a sort of the list, we have shown:

Theorem 5.1.2

Storing N data elements in an array of length N, and retaining no information other than the data and the value of N, it is possible to perform searches in 2 log N comparisons and insertions and deletions in $\sqrt{N/2} \log N + O(\sqrt{2N})$ comparisons and swaps. 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 $O(\sqrt{N})$ modification cost of the triangular grid. 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 $\sqrt{2N}$ Pointers

Observe that the $\Theta(\sqrt{N} \log N)$ behaviour 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 $O(\sqrt{N})$.

5.2.2 Batching the Updates with a Small Auxiliary Map

Another approach is to "batch" insertions and deletions. This can be accomplished with log N extra locations to store the pointers (indices) to the array and a bit to indicate whether insertion or deletion is to be performed. A total of $O(\log^2 N)$ bits are required for this modification map, furthermore, it is possible that log N extra locations in the array are used for elements which have already been deleted. We now describe the basic operations.

1. Deletion:

   Put a pointer (in the modification map) to the key to be deleted.

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 location N+1 (and increment N at the same time) and keep a new pointer to this location in the modification map.

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.

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 $O(\sqrt{N} \log N)$ operations, and so can be "time-shared" with the next log N insert/delete commands.

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

5.3 A Biparental Heap with Rotated Sorted 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 i^{th} 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 biparental heap. Note that the height of the structure is about $(3N)^{1/3}$, which is equal to the number of blocks. The ordering imposed on this structure is that of a biparental heap whose elements are rotated lists. More precisely,

(i) the elements in each subblock are stored in a cyclic shift of increasing order.

(ii) all the elements of the k^{th} subblock of the j^{th} block are less then all elements of the k^{th} and $k+1^{st}$ subblocks of block $j+1$.

As in the case of the biparental heap, 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 parameter 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 in $(N/9)^{1/3} \log N + O(N^{1/3})$ comparisons.

3. Insertion:

   By combining methods for the grid 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 biparental heap, 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 biparental heap) 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 biparental heap. The key differences are that a comparison with a single element in the grid 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

Storing N data elements in the first N locations of an array and retaining, in addition to data, only the current value of N, it is possible to perform each of the operations insert, delete and search in $O(N^{1/3} \log N)$ comparisons and swaps.

We note that the easiest way to initialise 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 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, deletions and searches can be performed reasonably efficiently. Table I summarizes the behaviour of algorithms acting on the implicit structures we have proposed.

Structure	Search	Insert/Delete	Extra Storage
Biparental heap	$\theta(\sqrt{N})$	$\theta(\sqrt{N})$	log N bits
Rotated list	θ(log N)	$O(\sqrt{N}$ log N)	log N bits
Rotated list with block pointers	θ(log N)	$O(\sqrt{N})$	$O(\sqrt{N}$ log N) bits
Rotated list with auxiliary table	θ(log N)	$O(\sqrt{N})$	$O(\log^2 N)$ bits
Combination of Biparental heap and rotated list	$O(N^{1/3}$ log N)	$O(N^{1/3}$ Log N)	O(log N) bits

Table 1: Costs of the Implicit Structures Discussed

7. References

[1] Bentley, J.L., D. Detig, L.Guibas, J. Saxe, "An Optimal Data Structure for Minimal Storage Dynamic Member Searching", unpublished manuscript.

[2] Borodin, A.B., L.J. Guibas, N.A. Lynch, A.C. Yao, "Efficient Searching via Partial Ordering", unpublished manuscript, (April 1979).

[3] Bondy, J.A., U.S.R. Murty, Graph Theory with Applications , American Elsevier Publishing Co., 1976.

[4] Knuth, D.E., The Art of Computer Programming , Vol. III, Sorting and Searching, Addison Wesley, 1973.

[5] Snyder, L., "On Uniquely Representable Data Structures", Proc. 18th IEEE Symposium on FOCS (1977), pp. 142-146.

[6] Suwanda, H., Ph.D. Thesis , Dept. of Computer Science, University of Waterloo (in preparation).

[7] Williams, J.W.J., "Algorithm 232: Heapsort", CACM 7(1964), pp. 347-348.