JOURNAL OF COMPUTER AND SYSTEM SCIENCES 33, 66–74 (1986)

An Implicit Data Structure Supporting Insertion, Deletion, and Search in $O(\log^2 n)$ Time*

J. IAN MUNRO

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

Received April 1, 1985; revised May 1, 1986

We introduce a data structure that requires only one pointer for every k data values and supports searches in $O(\log n)$ time and insertions and deletions in $O(k + \log n)$ time. By choosing k to be $O(\log n)$, pointers can be encoded into the relative order of data values and this technique can be used to support insert, delete and search in $O(\log^2 n)$ time while using only the storage needed for the data. © 1986 Academic Press, Inc.

1. THE PROBLEM AND THE RESULTS

Pointers are routinely used to indicate relationships between keys in a data structure. Their use is very often crucial in implementing flexible and efficient algorithms. Their explicit representation, however, can contribute very significantly to the space requirements of a structure. A natural question to ask is whether pointers are inherently necessary for structures as efficient as AVL [1] trees in maintaining a dictionary (operations of insert, delete, and search). One might think so; however, we note that the heap [2, 9] is an ideal structure for a priority queue. It uses no pointers and requires work comparable to that implied by the information theoretic lower bound. In this paper we focus on implicit (or pointer free) data structures for the dictionary problem. This problem was first explicitly studied by Munro and Suwanda [7]. By the "usual" pairing function, they suggested keeping n data values in consecutive locations but thinking of them as occupying the locations above the minor diagonal in a square array sorted by row and by column. Maintaining the data in this way, they showed searches and updates can be performed in $O(n^{1/2})$ time and the structure can grow or shrink smoothly. They also demonstrated that, if the elements are kept in any fixed partial order by their values, the product of search and update times must be $\Omega(n)$, i.e., their structure is more or less optimal in that class. They concluded by suggesting the idea of storing components of a structure in an arbitrary cyclic shift of sorted order. Such an organization is not a partial

* This work was supported by Natural Sciences and Engineering Research Council of Canada Grant A-8237, AT&T Bell Laboratories, and, at the University of Washington, National Science Foundation Grant MCS7609212A.

0022-0000/86 $3.00

order, and they were able to reduce the search time to $O(\log n)$ with an insert/delete cost of $O(n^{1/2} \log n)$, or alternatively $O(n^{1/3} \log n)$ for any operation. Frederickson [5] extended the use of rotated lists to achieve $O(\log n)$ search and $O(2^{\sqrt{2 \log n}} \log^{3/2} n)$ insert and delete time. This paper solves the main problem left open in [7] by demonstrating that there is an implicit data structure for the dictionary problem that runs in polylog time. The search time is a bit more than Frederickson's, but the update cost is a dramatic improvement. In this work, as in previous work on the problem, it is assumed all data values are distinct. This assumption is not necessary for the proof of Theorem 1, but is required for the titular result, Theorem 2.

THEOREM 1. *There is a data structure that supports searches in $O(\log n)$ time and insertions and deletions in $O(k + \log n)$ time and requires at most $n + k^2$ data locations and $k + O(n/k)$ pointers counters and flags. Furthermore, this structure can occupy a contiguous segment of memory.*

This technique will be presented in Section 2. It should be drawn to the reader's attention that this theorem is different from the auxiliary theorem of the preliminary forms of this paper [6] (also sketched in Sect. 4). The result proven here leads to faster algorithms at the cost of a small increase in storage. The main advantage of the approach taken here is, however, that the update algorithms are much simpler.

The spirit of Theorem 1 is that k can be parameterized in terms of the anticipated value of n. This leads to a requirement of $o(n)$ pieces of structural (i.e., nondata) information. Such structures have been called semi-implicit [7]. To convert this semi-implicit data structure to a fully implicit one, a number of points must be addressed. First, the issue of k potentially nonfull nodes must be addressed. Second, pointers/counters in the range $1,\ldots, n$ are encoded by the relative order of $2 \lg n$ data values in a node. Finally, a mechanism must be found to, effectively, permit k to change in response to changes in n. These are discussed in Section 3 and lead to the main result of this paper:

THEOREM 2. *There is an implicit data structure that supports searches, insertions, and deletions in $O(\log^2 n)$ time.*

2. A SEMI-IMPLICIT STRUCTURE

A natural approach to the semi-implicit version of our problem is to lump k data values of consecutive rank into a single node along with a constant number of pointers, flags, and counters. These nodes can be arranged in an AVL tree [1] or some similar structure. Searches, of course, are trivial. The difficulty is that, as elements are inserted, single values, rather than k consecutive values, have to be appended.

We will outline an interesting way to overcome this problem that has eluded previous investigations.

The data structure consists of $k-1$ pointers (*header* $[1,\ldots, k-1]$) and a potentially unbounded array of nodes of identical format. Logically these nodes are viewed as forming a primary structure (the AVL tree alluded to) and a secondary structure of $k-1$ doubly linked lists with the *header* $[i]$ referring to the first node of the ith list. Hence we will refer to the nodes as AVL or list nodes depending on their current use. All utilized nodes contain a full complement of k data values, with the possible exception of the first node in each of the linked lists. Each node (see also Fig. 1) contains:

k indexable data locations,

4 pointers (*left*, *right*, *coh*, and *manip*) each capable of referring to a node in the structure,

1 *counter* which holds an integer in the range $[0, k]$, and 2 flags.

The data values of each AVL node are of consecutive rank among the elements currently stored in the structure. By convention the smallest k data values are stored in the leftmost AVL node. The key invariant is

> *At most* $k-1$ *data values may fall between the largest value of one AVL node and the smallest of the next.*

A set of i $(i \leqslant k-1)$ such data values that fall after those of an AVL node (but before those of the next) are stored (consecutively and in increasing order) in the ith linked list. We will refer to such a "handful" of i elements as the *maniple* of that AVL node. While the elements of a maniple are stored in sorted order, no particular order is maintained on the maniples in a list. Observe that as i $(<k)$ typically will not divide k, a maniple may start in one list node and finish in the first few locations of the next. On the other hand, a list node typically contains several maniples (the first and last may not be entirely present), and so we speak of the set of maniples that start in a list node as a *cohort*. (We will resist referring to the lists as legions.)

In the AVL nodes, *left* and *right* refer to the children of the node in the tree, while *manip* refers to the list node containing the smallest element of the maniple of the AVL node. A null *manip* pointer indicates no elements fall between the current AVL node and the next. The *counter* is used to indicate the position within the list

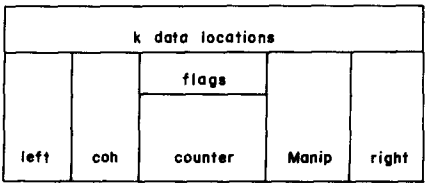

FIG. 1. The format of a node.

node in which the smallest member of this maniple is found. In order to maintain the property that only the first so many locations of memory are used by our structure, it is necessary, at times, to move entire nodes and hence all references to them. To facilitate the moving of list nodes it is convenient to keep the AVL nodes of the maniples of each cohort in a circular list. The pointer *coh* is employed for this purpose. The two flags are used to indicate the state of the local balance of the AVL tree.

In each list node, *counter* indicates the list number (maniple size), one flag is employed to indicate whether the node is the first in the list, the other is not used; *left* and *right* perform their expected tasks of referring, respectively, to the previous and next nodes of the list; *coh* and *manip* are both idle. Figure 2 illustrates the structure of a list and its connections with AVL nodes.

The growth and shrinkage of a list will occur at its first node, the only node in a list that need not be completely filled. Hence the role of the pointer *header* [i] in referring to the first node in list i is crucial. The fact that the maniple size (and hence list number) is stored in each list node, permits access, through the relevant header pointer, to the first node of the list in constant time.

When a new node is required, the next one in free storage is simply taken. When a node is no longer required, however, we want to release the last node in the currently utilized portion of memory. The contents of this last node must be transferred to the vacated one. This is easily done in $O(k)$ time. The trick is to update all pointers *to* that node. If the node is in the AVL tree, $O(\log n)$ time suffices to find its parent and the task is quickly finished. List nodes, however, can be referred to

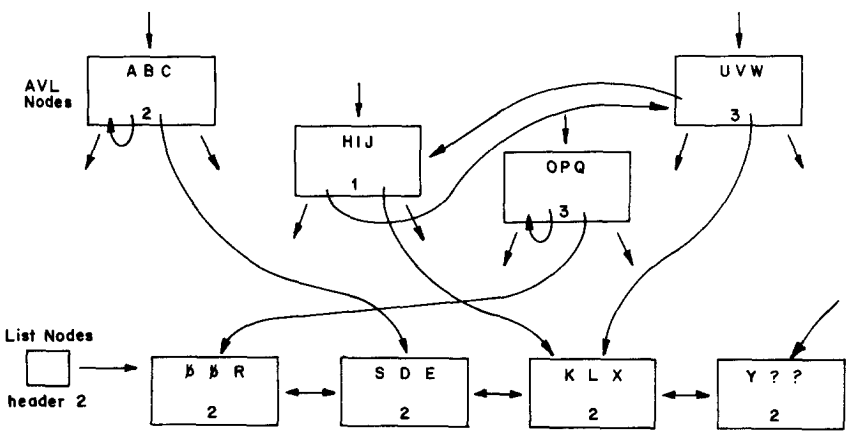

FIG. 2. A segment of the data structure (with $k = 3$) showing several AVL nodes and a portion of the list of maniples of size 2: the short arrows on the AVL nodes indicate tree connections; data values are shown across the top of the nodes; the counter is 2 in each list node and in the AVL node indicates the position of the relevant maniple in its list node; the circular cohort list of AVL nodes HIJ and UVW is of size two, while the others are singular as indicated; the connections from AVL nodes to list nodes are indicated by the manip pointers: the ?? in the rightmost node indicates the continuation of the list.

not only by their two list neighbors, but by up to k *manip* pointers in the AVL tree. An AVL tree search of the last data value in the node finds one of the relevant AVL nodes. The cohort cycle is then found using the *coh* pointers. In either case the task is performed in $O(k + \log n)$ time.

A search is performed with an AVL tree search and a binary search of portions of at most two nodes. It therefore requires $O(\log n)$ time. Updates are surprisingly easy. Insertion of the value x proceeds as follows:

Perform a search of the AVL tree to determine whether x falls between the maximum and minimum values of some AVL node (or is less than the smallest value in the structure).
If so, insert x into the relevant node, moving all values stored there and ultimately displacing the largest value from the node (call this displaced value y).
Otherwise, assign x to y; y must now be inserted into the secondary structure.
The location of the appropriate maniple is found as a byproduct of the AVL tree search. The size of the maniple (if nonzero) is found with the maniple. Let i denote the (pre-insertion) size of the maniple. Do the appropriate case:

$i = 0$. If the first node of the list referred to by *header* [1] is not full, insert y into that node. Set the *manip* pointer of the AVL node where the search ended to refer to this list node and adjust the relevant *coh* pointers.
If the first node of list 1 is full (or the list is null), insert a new node containing y at the head of this list, and set the *manip* pointer of the AVL node to refer to it and the *coh* pointer to refer the AVL node itself.

$i = k - 1$. Swap the maniple associated with y and the lead maniple of list $k - 1$. If the first node of list $k - 1$ is completely full, copy y and the $k - 1$ elements of its associated maniple into the next free node and insert that node into the AVL tree. The cohorts falling to either side of this new AVL node will be of size 0. Adjust the appropriate pointers and rebalance the AVL tree if necessary. The first node of list $k - 1$ now contains only 1 element.
Otherwise the first node of list $k - 1$ contains only the maniple of y or part of that maniple. Move y and all of its associated maniple into that lead node and insert that node into the AVL tree as stated. Adjust *header* $[k - 1]$ to refer to the next node of list $k - 1$ and delete any values that have just been copied.

Otherwise. Insert y and the i elements of its associated maniple into the front of list $i + 1$. This may require obtaining a new node and will require adjustments to the cohort cycle (*coh* pointers) associated with the node formerly at the front of list $i + 1$.
Move the first maniple of list i into the space vacated later in that list. Again, adjust appropriate references. This could release the first node of list i. If so transfer the contents of the last utilized node to that space and update the relevant references.

An insertion, then, may be performed in a constant number of AVL tree searches

(at a cost of $O(\log n)$ each), one AVL tree rebalancing (again $O(\log n)$), and $O(k)$ pointer changes and data moves. This leads to the claimed insertion cost of $O(k+\log n)$.

Deletions are performed in a manner completely analogous to insertions, from which Theorem 1 follows.

3. A Fully Implicit Structure

There are two issues that must be addressed in converting the structure of the preceding section to a fully implicit one. First, the $k-1$ potentially nonfull nodes must be represented in a different manner. Second, the pointers, counters, and flags must be encoded in a relative ordering of the data values in a node.

The first difficulty is handled in a rather straightforward manner. Data values that would have been stored in a nonfull node are kept in the last $(k-1)^2$ or fewer locations, ending in position n. Such elements are stored in order of their list number and, among those from the same list, in sorted order. The $k-1$ list headers are replaced by two pointers each, one to the last full node of the relevant list and one to the location in which these last few values begin. Modifications of search and update procedures to accommodate this minor structural change are straightforward. The only significant impact on the runtime of these procedures is that insertions and deletions may involve shifting all $O(k^2)$ data values of the formerly nonfull nodes. All $k-1$ pointers to the first value in each subsection of this suffix may also have to be modified as a consequence of one update. The insertion and deletion costs are increased to $O(k+\log n)$ operations of comparing data values or following or updating pointers plus $O(k^2)$ movements of data values.

Encoding pointers and counters takes a little more care. First, assume that $\lceil \lg n \rceil$ can be treated as fixed. Under this assumption, it is obvious a pointer or counter can be encoded in the relative order of $2\lceil \lg n \rceil$ data values. The values are stored more or less in sorted order, however, odd–even pairs of consecutive elements are kept in increasing order to indicate a 0 and reversed to denote a 1 in the binary representation of an integer between 0 and $n-1$. Reading or updating a pointer can, then, be performed in $O(\log n)$ time. Such a structure still supports a slightly modified binary search.

Using this trick, we choose $k = c\lceil \lg n \rceil + d$ (for appropriate constants c and d) and so represent the four pointers, one counter, and two flags of a node in the semi-implicit structure. The ordering of the data values in the first $4\lceil \lg n \rceil (k-1)$ is used to encode pointers to the last full node of each of the list and the location of the first (smallest) value in each of the logical nonfull nodes. Under the assumption that $\lceil \lg n \rceil$ does not change, this permits searches, insertions, and deletions to be performed in $O(\log^2 n)$ time.

The final issue is handling the growth or shrinkage of n. Following Frederickson [3], this is done by maintaining $O(\log \log n)$ distinct structures of sizes 2^{2^i} ($i =$ some constant,..., $\lfloor \lg \lg n \rfloor$) plus one more of the appropriate size. In any such

structure, the logarithm of the number of elements in it and all smaller ones is fixed to within a factor of 2, and hence the node size may be fixed. The cost of a search in each of these structures is dominated by that of a search in the largest, due to the double exponential growth of the structure size. Insertions are always made on the last structure, and a deletion is accomplished by moving an element from the last structure into the one in which the element to be deleted is found.

Theorem 2 now follows.

4. Another Approach

The solution presented here differs from that of preliminary versions of this paper [6]. The approach taken there leads to a slightly weaker semi-implicit structure ($O(k \log n)$ update cost, but no difficulties with nonfull nodes). The same $O(\log^2 n)$ bound is proven for the fully implicit structure. As this alternative approach may be of some interest we will sketch the general approach, but refer the reader to the conference proceedings or the more detailed technical report for the details of the structure and the update algorithms.

Again the semi-implicit structure consists of an AVL tree and a sequence of doubly linked lists. Again AVL tree nodes each contain k values of consecutive rank, and have pointers to aid in a search that continues into the linked list portion of the structure. Here the similarity between this and the previously described structure ends.

The AVL tree is referred to as level 0; the lists are levels 1 through $O(\log n)$. The data values in any node are of contiguous ranks among the values at that or any subsequent level. The crucial invariant on the structure is:

> *The elements in the leftmost node at any level precede all those at subsequent levels. Ignoring details regarding the bottom two or three levels, there are at least k and at most $3k-1$ data values in subsequent levels which are greater than the values in a given node but less than those in the following node at its level (if such a right neighbor exists).*

From this invariant it follows that at least a quarter and at most half of the elements at or below a given level are actually at that level. It follows, then, that there are between $\lg n$ and $\lg n/\lg(4/3)$ levels. Each node at level i will contain a gap pointer to the node at level $i+1$ that contains the smallest value greater than those in the given node or contains the largest value at level $i+1$ smaller than this value. A few other pieces of information such as counters and back pointers prove useful as well. Indeed, the requirements for structural information are comparable to that of Section 2, although only the single node at the last level may be nonfull.

Again, searches are easily performed in $O(\log n)$ comparisons and pointers followings. The key issue in updates is that the data values falling between those of two consecutive AVL nodes (or indeed two consecutive nodes of one of the linked

lists), may be spread among one or two nodes in each of the subsequent levels. A single insertion or deletion can cause changes in as many as three nodes in each level. It is not hard to see that such updates can be performed in time polynomial in k and $\log n$. Some care and complication, however, seem necessary to achieve the claimed $O(k \log n)$ update time. Hence the approach presented in Section 2 seems preferable for the purposes of this paper.

5. Conclusions

Do pointers give you more than simpler code and a constant factor in run time? Granted these are important features, but one feels pointers must give more in the dictionary problem. I make no conjecture either way. Suwanda and I were genuinely surprised when we realized that organization other than a partial order was crucial. A few researchers have conjectured that Frederickson's scheme was optimal and that no polylog solution for the dictionary problem existed. He has demonstrated its optimality in a restricted class [4]. The space saving construction is interesting, and its essence may be of use in other environments. The fully implicit structure, however, is the main result. Given that it effectively encodes pointers, and must charge $O(\log n)$ for each use, one sees that the approach we have taken will go no farther.

A few other results follow quickly. If data values have explicit probabilities of access, one can take advantage of our structures (particularly the $\log \log n$ separate structures for the fully implicit version). The sequence structures of doubly exponentially increasing size can also be used if the data values come from underlying but unknown distribution. A "self organizing implicit" structure along the lines of Frederickson's [3] can be formed. There are several related, interesting, and perhaps tractable problems:

- Give a better upper bound on the number of pointers necessary to support $O(\log n)$ searches and polylog updates. Can this be reduced to, say, $O(n^{1/2})$?
- Show, as Allan Borodin has suggested, that no implicit structure can support searches in $O(\log n)$ time while requiring only a constant (!) number of *moves* for an update.[1]
- Give an easily implementable polylog solution to the implicit dictionary problem. This would be of interest even if the bound held only for the average case. Munro and Poblete [8] have suggested a candidate, but the analysis of its behavior remains open.

[1] *Note added in proof.* This has been proven true. See Borodin *et al., in* "Proceedings, ICALP," 1986.

REFERENCES

1. G. M. ADEL'SON-VEL'SKIĬ AND Y. M. LANDIS, An algorithm for the organization of information, *Dokl. Akad. Nauk. SSSR* **146** (1962), 263–266.
2. R. W. FLOYD, Algorithm 245: Treesort 3, *Comm. ACM* **7** (1964), 701.
3. G. N. FREDERICKSON, Self organizing heuristics for implicit data structures, *SIAM J. Comput.* **13** (1984), 277–291.
4. G. N. FREDERICKSON, Recursively rotated orders and implicit data structures: A lower bound, *Theoret. Comput. Sci.* **29** (1985), 75–85.
5. G. N. FREDERICKSON, Implicit data structures for the dictionary problem, *J. Assoc. Comput. Mach.* **30** No. 1 (1983), 80–94.
6. J. I. MUNRO, An implicit data structure for the dictionary problem that runs in polylog time, *in* "Proceedings 25th Annual IEEE Sympos. FOCS," Oct. 1984, pp. 369–374; Expanded version, Research Report CS-84-20, Department of Computer Science, University of Waterloo, Aug. 1985.
7. J. I. MUNRO AND H. SUWANDA, Implicit data structures for fast search and update, *J. Comput. System Sci.* **21** (1980), 236–250.
8. J. I. MUNRO AND P. V. POBLETE, Searchability in merging and implicit data structures, *in* "Proceedings, ICALP," pp. 527–535, Jul. 1983, Lecture Notes in Comput. Sci. Vol. 154, Springer-Verlag, New York/Berlin, 1983.
9. J. W. J. WILLIAMS, Algorithm 232: Heapsort, *Comm. ACM* **7** (1964), 347–348.