BIT 27 (1987), 324 329

# SEARCHABILITY IN MERGING AND IMPLICIT DATA STRUCTURES †

J. IAN MUNRO and PATRICIO V. POBLETE *

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

**Abstract.**

We introduce the notion of *searchability* as a property of an in place merging algorithm. We show that a pair of sorted arrays can be merged in place in linear time, so that a search can be performed in logarithmic time at any point during the merging process. We apply this method to devise an implicit data structure which can support searches in $O(\log^2 n)$ time in the worst case, and $O(\log n)$ on the average, and insertions in $O(\log n)$ time, in the worst case.

*CR categories:* E.1, F.2.2.

## 1. Introduction.

In studying the process of merging sorted blocks of data, three properties have been considered in the literature: (i) minimizing the number of comparisons [4], (ii) performing the merge in place [6] (also in [5, ex. 5.2.4-10]), and (iii) maintaining stability [3], [10]. We introduce the notion of *searchability* as a property of a merging algorithm. A merging algorithm is said to support $f(n)$ searchability if, at any stage in the process, a search for an arbitrary element can be performed in time $f(n)$. The standard merging algorithm is $O(\log n)$ searchable. Like stability, this property is of greater interest in considering in place merging schemes. For instance, Kronrod's in place merging algorithm [6] is only $\Theta(n^{1/2})$ searchable, since it completely "randomizes" a block of $n^{1/2}$ elements. Searchability of merge algorithms can be viewed as a paradigm for the more general issue of performing basic operations while reorganizing a database.

The main result of this paper is an in place merging algorithm that is $O(\log n)$ searchable, and that runs in time $O(n)$. Our interest in this problem was sparked by the development of an implicit (i.e. pointer free) data structure for performing searches and insertions. We were relatively content with an algorithm that ran in time $O(n \log n)$, until we found one that runs in linear time. The slower method

† This research was partly supported by NSERC under grant A8237 and presented in preliminary form at the 10th International Colloquium on Automata, Languages and Programming.

* On leave from the University of Chile.

Received March 1987.

is described here, too, since it is used as a component of the faster one. The notions of Wong [11] were useful in the development of these algorithms.

We use the merge algorithm to construct an implicit data structure that supports insertions in time $O(\log n)$ and searches in time $O(\log^2 n)$ in the worst case, and $O(\log n)$ in the average.

## 2. A slow algorithm.

LEMMA 1. *Two sorted arrays of n elements can be merged in place using $O(n \log n)$ time and $O(1)$ pointers, in such a manner that a search can be conducted at any time in $O(\log n)$ comparisons.*

PROOF: Assume that $A[1..n]$ and $B[1..n]$ are to be merged, and that $B$ immediately follows $A$. To simplify the presentation, we first assume $n$ is a power of 2. All logarithms are taken to base 2, unless otherwise noted.

**Procedure** Slow_merge

**begin**
  **for** $i = 1$ **to** $\log n - 1$ **do**
  **begin**
    $A$ and $B$ at this point are in sorted order and each can be viewed as $n/2^{i-1}$ blocks of $2^{i-1}$ elements of consecutive ranks in $A \cup B$.
    In a single scan ($n/2^{i-2}$ comparisons and $O(n)$ moves) concatenate pairs of blocks of consecutive rank in $A \cup B$, so that $A$ and $B$ can each be viewed as $n/2^i$ blocks of $2^i$ elements of consecutive rank.
    This is done by repeatedly finding the three blocks of smallest elements. At least two of them will be contiguous. Move the two blocks of smaller elements into that segment of $2^i$ locations and the third to the remaining block. (If we remember the location of the block with the smallest elements from the previous iteration, we need only two comparisons to find the second and third.)

  **end**
  Exchange $A$ and $B$ if necessary.
**end**

It should be clear that this algorithm requires about $O(n)$ comparisons but an unfortunate $O(n \log n)$ moves. Throughout Slow_merge both $A$ and $B$ remain in sorted order with one exception. In the process of concatenating two blocks, one may have a point of non-monotonicity. This leads to a $3 \log n$ search algorithm.

When $n$ is a power of 2, all blocks are the same size, and the concatenation of two blocks can be done by swapping. When $n$ is not a power of two, we

form blocks of approximately the same size, and we can concatenate the two blocks of smallest elements onto the locations occupied by the two contiguous blocks by using the reversal operation, and the fact that $\beta\alpha = (\alpha^R\beta^R)^R$.

## 3. A fast algorithm.

THEOREM 1. *Two sorted arrays of $n$ elements can be merged in place using $O(n)$ time and $O(1)$ pointers, in such a manner that a search can be conducted at any time in $O(\log n)$ comparisons.*

PROOF: We make use of the fact that two blocks of data can be merged using a third block as a "scratch area." This third block has to be at least the size of the smallest of the blocks being merged. The merge proceeds as usual, but the "move" operation is replaced by "swap." As a result, its data values are not destroyed, but they are permuted from their orginal order.

**Procedure** Searchable_merge

**begin**

1) Find the $2\log n$ smallest elements of the whole set, and move them to the first $2\log n$ locations of $A$, leaving $A$ and $B$ permuted from their original form but each still in sorted order. This is done in time $O(n)$ as follows: In $O(\log n)$ comparisons (actually $O$ (loglog $n$) suffice) determine the $r$ elements of $B$ to be moved. The elements of ranks $2\log n - r + 1$ through $2\log n$ of $A$ are merged with the bottom $n-r$ elements of $B$ into (all of) $B$, using the first $r$ elements of $B$ as a scratch area. This puts $B$ in proper form but leaves the top $r$ elements of the original $B$ in unknown order in locations $2\log n - r + 1$ through $2\log n$ of $A$. Sorting the first $2\log n$ elements of $A$ completes this step.
2) Scan the $n - 2\log n$ remaining elements of $A$ and the $n$ elements of $B$ from top to bottom, logically forming blocks of $\log n$ elements of consecutive ranks in $A \cup B$. This can be done in a single pass using the first $2\log n$ locations of $A$ as a scratch area.
3) Re-sort the first $2\log n$ locations of $A$.
4) View each conceptual block of $A$ and $B$ as consisting of a first element, *header*$[j]$ the next $\log n - 2$ elements, *middle*$[j]$, and the last, *trailer*$[j]$. Apply Slow_merge to the $2n/\log n$ elements $\{$*header*$[i]\}$. At this point, these "headers" are in their final positions.
5) **for** $i = 1$ **to** $2n/\log n$ **do**
   **while** *middle*$[i]$ is not in its final position (i.e. between *header*$[i]$ and *header*$[i+1]$) **do**
   By binary search on the headers, find the final position of *middle*$[i]$ and swap *middle*$[i]$ with that block.

6) Apply Slow_merge to $\{trailer[i]\}$
**end**

Each of the 6 labeled steps requires at most linear time and so the entire algorithm is linear. Note that when Slow_merge is called, it is applied to lists of $n/\log n$ elements. In step 5) each iteration of the **while** loop puts one block in its final position. Therefore, there are $O(n/\log n)$ iterations, each with a cost of $O(\log n)$ comparisons and $O(\log n)$ moves. The $O(\log n)$ searchability follows by arguments similar to those applied to Slow_merge. Note that during step 5) searches are performed by two binary searches, on the headers and on the trailers.

For simplicity we have assumed that $\log n$ divides $n$. If this is not the case, we can form blocks of size $\lfloor \log n \rfloor$, and enlarge the initial block of $2 \log n$ elements to hold also the elements that did not fit in any of the blocks. Also, the previous algorithm can be easily generalized to handle the case of merging sets of different sizes.

## 4. An application to implicit data structures.

An implicit data structure [9] is an array of $n$ data elements organized in some fashion to support appropriate operations without the use of pointers. Clearly a sorted list is a very effective implicit structure for searching. A sorted list is, of course, disastrous for insertions and deletions. If the operations insert, delete and find are to be supported, then a rather complex structure suggested by Munro [7] can be used to perform these operations in $O(\log^2 n)$ time. Bentley et al. [1] have considered a restricted version of this problem in which no deletions are permitted. They are able to achieve searches in $O(\log^2 n)$ comparisons in the worst case and $O(\log n)$ on the average, with a conceptually straightforward method. Their scheme may require $\Theta(n)$ time for a single insertion, but is guaranteed not to take more than $\Theta(n \log n)$ time for a sequence of $n$ insertions. We build on their scheme and our merging algorithm to achieve $O(\log n)$ behavior in the worst case for insertions.

The basic idea of the Bentley *et al.* scheme is to retain up to $\log n$ sorted subarrays or blocks, one of length $2^i$ if the $i$th digit in the binary representation of $n$ is a 1. A search is performed by applying binary search to the blocks in decreasing order by size. Insertion is similar to binary addition: a new element is a new block of length 1. Each time two blocks of length $2^i$ appear, they are merged into a single block of length $2^{i+1}$. It follows that, although increasing the structure from $n = 2^k - 1$ to $2^k$ elements will spawn $k$ merges and $\Theta(n)$ work, the "amortized" number of comparisons per insertion is $O(\log n)$. In order to avoid additional storage in the merge phase, they employ Kronrod's algorithm [6].

To convert the $O(\log n)$ average insertion cost to a worst case bound, we must (i) spread the merging cost over many insertions and (ii) maintain $\Theta(\log n)$

searchability while doing so. One way to achieve the former subgoal is to apply what Bentley and Saxe [2] have dubbed the "online binary transform". The cost of merging is spread over several insertions in a manner that can be viewed as counting in a *redundant binary system, using the digits* 0, 1 *and* 2. The presence of a 2 in a given position indicates that the two corresponding blocks are being merged. If we delay, as much as possible, the expansion $2 \rightarrow 10$, then there will always be at least one block of each possible size, and it is not hard to see that the merging of two blocks of size $n$ can be spread over $n$ steps. We can therefore use a "time sharing" arrangement, where, for each insertion, we spend an $O(\log n)$ "time slice", doing a constant amount of work on each of the $O(\log n)$ "active" merges. In practice one would clearly use $O(\log n)$ words of $\Theta(\log n)$ bits each to monitor the progress of the merges, thus leading to a "semi-implicit" data structure. One could, of course, carry out a purely implicit implementation. A cache of $O(\log n)$ elements can be used to encode each pointer. The cost of encoding and decoding this information is kept under control by adopting the policy of always working on the the smallest uncompleted merge. This implies that if a "large" number of merges are worked on, then "most" of them are small and so less time is required to decode their status. Hence decoding is not a dominant issue.

The maintenance of searchability during the process follows by using the merging algorithm presented in the preceding section. Hence:

THEOREM 2: *There is an implicit data structure under which insertions can be performed in* $O(\log n)$ *steps and searches require* $O(\log n)$ *time on the average and* $O(\log^2 n)$ *in the worst case.*

## 5. Conclusions.

We have introduced the notion of $O(\log n)$ searchability as a desirable property of an (in place) merging scheme. In addition to showing that this new property can be achieved in linear time we have shown the usefulness of the concept by demonstrating an implicit data structure requiring only $O(\log^2 n)$ comparisons for a search and $O(\log n)$ for an insertion in the worst case. Our scheme requires only $O(\log n)$ time to search, on the average. This compares favourably with the data structure in [7], that requires $\Theta(\log^2 n)$ on the average, as well as in the worst case, for both operations, although the latter method does support deletions. In [8] we proposed a scheme for handling deletions in our data structure, that seems to perform well on the average, but for which no analysis is available. The reader is referred to [8] for a description of the method, and discussion of simulation results.

**Acknowledgement.**

We thank Gaston Gonnet, Pedro Celis, Joe Culberson and the other members of the Data Structuring Group for a number of productive discussions on the mathematical and experimental aspects of this work, and Vitus Chan for some preliminary experimentation.

## REFERENCES

1. J. L. Bentley, D. Detig, L. Guibas and J. B. Saxe, *An optimal data structure for minimal-storage dynamic member searching*, Carnegie-Mellon University, 1978.
2. J. L. Bentley and J. B. Saxe, *Decomposable searching problems I. Static-to-dynamic transformation*, Journal of Algorithms 1, 4 (Dec. 1980), 301–358.
3. E. C. Horvath, *Stable sorting in asymptotically optimal time and extra space*, Journal of the ACM 25, 2 (April 1978), 177–199.
4. F. K. Hwang and S. Lin, *A simple algorithm for merging two disjoint linearly ordered sets*, SIAM Journal on Computing 1, 1 (March 1972), 31–39.
5. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.
6. M. A. Kronrod, *An optimal ordering algorithm without a field of operation*, Dok. Akad. Nauk SSSR 186 (1969), 1256–1258.
7. J. I. Munro, *An implicit data structure supporting insertion, deletion and search in* $O(\log^2 n)$ *time*, Journal of Computer and System Sciences 33, 1 (August 1986), 66–74.
8. J. I. Munro and P. V. Poblete, *Searchability in merging and implicit data structures*, Proceedings 10th International Conference on Automata, Languages and Programming, Barcelona, July 1983.
9. J. I. Munro and H. Suwanda, *Implicit structure for fast search and update*, Journal of Computer and System Sciences 21, 2 (Oct. 1980), 236–250.
10. L. Trabb Pardo, *Stable sorting and merging with optimal space and time bounds*, SIAM Journal on Computing 6, 2 (June 1977), 351–372.
11. J. K. Wong, *Some simple in-place merging algorithms*, BIT 21 (1981), 157–166.