PERGAMON Computers and Mathematics with Applications 39 (2000) 193–197

An International Journal
computers & mathematics
with applications

www.elsevier.nl/locate/camwa

The Heap-Mergesort

R. A. CHOWDHURY, S. K. NATH AND M. KAYKOBAD
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh
kaykobad@buet.edu

(Received January 1999; accepted April 1999)

Abstract—In this paper, we present a new mergesort algorithm which can sort $n(= 2^{h+1} - 1)$ elements using no more than $n\log_2(n+1) - (13/12)n - 1$ element comparisons in the worst case. This algorithm includes the heap (fine heap) creation phase as a pre-processing step, and for each internal node v, its left and right subheaps are merged into a sorted list of the elements under that node. Experimental results show that this algorithm requires only $n\log_2(n+1) - 1.2n$ element comparisons in the average case. But it requires extra space for n LINK fields. © 2000 Elsevier Science Ltd. All rights reserved.

Keywords—Mergesort, Fine heap, Heap-mergesort, Complexity.

1. INTRODUCTION

Heap is a very important data structure frequently used in representing priority queues and in algorithmic design problems [1–5]. Heapsort algorithm was introduced by Williams [6] and later modified by Floyd [7]. This modified algorithm requires $2n\log_2 n - 2n$ comparisons to sort $n(= 2^{h+1} - 1)$ elements. The most efficient heapsort variants so far are a pure in-place algorithm identified by Gu and Zhu [8] with at most $n\log_2 n + n\log_2(\log_2 n) + O(n)$ element comparisons, and a not-in-place variant based on fine-heaps by Carlsson *et al.* [9]. The fine-heap can be implemented using $n/2$ additional bits, and it can be used to sort n elements using $n\log_2 n + 0.91667n$ element comparisons in the worst case. In comparison, previously identified bound is given by $n\log_2 n + n$ [10,11].

Mergesort can sort n elements with at most $n\lceil\log_2 n\rceil - 2^{\lceil\log_2 n\rceil} + 1$ comparisons using space $2n$ [12]. So mergesort is more efficient but uses much more space. It should be noted that mergesort can be implemented using links [2]. The best in-situ mergesort is due to Katajainen *et al.* [13]. It can sort n elements with at most $n\log_2 n + O(n)$ comparisons and $\varepsilon n\log_2 n$ element transpositions.

In this paper, we present a mergesort algorithm based on merging heaps and use the concept of fine-heaps and the improvement in the heap creation phase by Carlsson *et al.* [9] to achieve $n\log_2(n+1) - (13/12)n - 1$ element comparisons and $O(n)$ element movements in the worst case.

The authors would like to thank the referees for helpful suggestions and comments. The authors also wish to thank M. Murshed of the Australian National University for his support.

PII: S0898-1221(00)00075-4

The proposed algorithm requires extra space for n LINK fields. Note that, the lower bound for both the average and worst case number of comparisons of general (comparison-based) sorting algorithms is $\log_2(n!) \approx n \log_2 n - 1.442695n$.

2. THE HEAP-MERGESORT ALGORITHM

In the preprocessing step of this algorithm, we need to create a heap with LINK fields. The LINK field of each internal node will point to the smaller of its children if it has two children, otherwise the LINK field will point to the left child. The LINK fields of the leaves will contain 0. The heap creation process is identical to that used in creating fine-heaps [9] and requires the same number of comparisons. But each of the LINK fields must be a word of size $\lceil \log_2 n \rceil + 1$, where n is the number of elements in the heap.

According to the property of fine heaps, we already know the smallest and the next larger element of the entire heap. So if we assume that the two subheaps of the root node (each with $2^h - 1$ elements, where $n = 2^{h+1} - 1$ is the total number of elements in the heap) are already sorted, the only thing that remains to get the sorted list of all the elements of the heap is to merge two sorted lists with $2^h - 2$ and $2^h - 1$ elements, respectively. The two subheaps have already been sorted using the same procedure and this process is continued recursively until we reach a heap with three elements. To eliminate recursion we may apply the merging process bottom-up starting from the $\lfloor n/2 \rfloor^{\text{th}}$ element of the heap and continuing up to the root. The pseudo-code of the algorithm is given in Figure 1.

```
procedure HEAP_MERGE_SORT(A, LINK, n)
// A[1 : n] is the array of the elements to be sorted in nondecreasing order. //
// After the sorting phase A[1] will contain the smallest element of the array, //
// LINK[1] will point to the 2nd smallest element, LINK[LINK[1]] to the 3rd, //
// LINK[LINK[LINK[1]]] to the 4th, etc. //

        element_type A[1 : n]
        integer i, n, LINK[0 : n]

        // We assume the existence of the following routine based on the works of //
        // Carlsson et al. [9] //
        call CREATE_FINE_HEAP_WITH_LINKS(A, LINK, n)

        for i ← ⌊n/2⌋ to 1 by −1 do
            if (LINK[i]mod 2 = 1)
                // The following routine is exactly the same as the MERGE1 routine //
                // described on page 120 of [2] which assumes n, A[1 : n] and LINK[0 : n] //
                // to be global. //
                call LINK_MERGE(LINK[LINK[i]], 2 * i, LINK[2 * i + 1])
            else
                if (2 * i + 1 ≤ n)
                    call LINK_MERGE(LINK[LINK[i]], 2 * i + 1, LINK[2 * i])
                endif
            endif
        repeat

end HEAP_MERGE_SORT
```

Figure 1. The heap-mergesort algorithm.

3. NUMBER OF ELEMENT COMPARISONS AND MOVEMENTS

For a heap with $n = 2^{h+1} - 1$ elements each of the two subheaps of the root node has $2^h - 1$ elements. We know that the element at the root is the smallest element of the heap. According to the property of fine heaps the LINK field of the root element points to the smaller of its two children. Hence, we already know the smallest and the next larger element of the heap. Let us assume that the two subheaps are already sorted and the next larger element is the root of the left (right) subheap. So, if we merge the $2^h - 2$ sorted elements (excluding the root) of the left (right) subheap with the $2^h - 1$ sorted elements of the right (left) subheap, we will get a sorted list of all the elements of the heap. This merging step requires $(2^h - 2) + (2^h - 1) - 1 = 2^{h+1} - 4 = n - 3$ element comparisons in the worst case. The final three elements are sorted according to the property of fine heaps.

Hence, the total number of element comparisons required during the merging phase is given by

$$
\begin{aligned}
C(n) &= (n-3) + 2C\left(\frac{n-1}{2}\right) \\
&= (n-3) + (n-7) + 2^2 C\left(\frac{n-3}{2^2}\right) \\
&= (n-3) + (n-7) + (n-15) + 2^3 C\left(\frac{n-7}{2^3}\right) \\
&= \left[(n+1) - 2^2\right] + \left[(n+1) - 2^3\right] + \left[(n+1) - 2^4\right] + \cdots + \left[(n+1) - 2^{m+1}\right] \\
&\quad + 2^m C\left(\frac{n-(2^m-1)}{2^m}\right) \\
&= m(n+1) - 2^2\left(1 + 2 + 2^2 + \cdots + 2^{m-1}\right) + 2^m C(3) \\
&\quad \left[\text{assuming } \frac{n-(2^m-1)}{2^m} = 3 \Rightarrow m = \log_2(n+1) - 2\right] \\
&= m(n+1) - 2^2\left(\frac{2^m-1}{2-1}\right) + 2^m(0) \\
&= m(n+1) - 2^{m+2} + 4 \\
&= [\log_2(n+1) - 2](n+1) - (n+1) + 4 \\
&= n\log_2(n+1) - 3n + \log_2(n+1) + 1.
\end{aligned}
$$

A fine heap can be created using at worst only $(23/12)n - \log_2(n+1) - 2$ element comparisons [9]. By including this heap creation cost, therefore, the worst case number of comparisons required by the algorithm is found to be

$$
\begin{aligned}
n\log_2(n+1) - 3n + \log_2(n+1) + 1 + \frac{23}{12}n - \log_2(n+1) - 2 \\
= n\log_2(n+1) - \frac{13}{12}n - 1 \cong n\log_2(n+1) - 1.083n - 1.
\end{aligned}
$$

The number of element movements in the heap creation phase will not exceed $O(n)$ and there is no element movement during the merging phase. Hence, the overall number of element movements will remain $O(n)$.

4. EXPERIMENTAL RESULTS

In Table 1, we present the average number of comparisons for the algorithm presented in this paper along with that obtained for the classical mergesort algorithm. For each n, the result is the average of that for 100 iterations on random data set. "Heap-Mergesort 1" denotes the heap-mergesort algorithm which uses the straight forward method of creating fine-heaps (using $2n - \log_2(n+1) - 1$ element comparisons in the worst case [9]) and "Heap-Mergesort 2"

denotes the heap-mergesort algorithm which uses the improved method of fine-heap creation [9]. It should be noted here that both "Mergesort" and "Heap-Mergesort 1" have similar worst case complexity ($n \log_2 n - n + O(1)$) whereas "Heap-Mergesort 2" is better ($n \log_2 n - (13/12)n + O(1)$) in the worst case.

Table 1. Average number of comparisons required by Mergesort and Heap-Mergesort.

$n = 2^{h+1} - 1$	Mergesort	Heap-Mergesort 1	Heap-Mergesort 2
1023	8935	8928	8947
2047	19922	19917	19951
4095	43967	43926	43999
8191	96130	96051	96197
16383	208648	208505	208793

The results show that for $n = 2^{h+1} - 1$, "Heap-Mergesort 1" requires fewer comparisons than that in "Mergesort" and "Mergesort" requires fewer comparisons than that in "Heap- Mergesort 2". This occurs because the heap creation phase of "Heap-Mergesort 2" is designed for performing better in the worst case, not on the average while that of "Heap-Mergesort 1" performs better in the average case. For example, "Heap-Mergesort 2" requires exactly 28 comparisons for creating three fine heaps of size 7 each, whereas "Heap-Mergesort 1" can do the same using fewer comparisons on the average.

Even "Heap-Mergesort 1" will require slightly higher number of comparisons on the average if n is not near or equal to $2^{h+1} - 1$. This is because linear-merge is optimal for merging lists with equal or almost equal number of elements. If the heap is not full then in some merging steps the two subheaps to be merged will not have equal or nearly equal number of elements. For nonfull heaps, however, the number of comparisons may be reduced by modifying this algorithm appropriately to be able to use the concept of minimum comparison merging [14].

5. CONCLUSION

In this paper, we have introduced a new variant of the mergesort algorithm with a better worst case behavior than that of the classical one. The worst case number of comparisons required by this algorithm is less than that of the traditional algorithm by about $0.083n$ comparisons and is only $0.34n$ comparisons off from the theoretic lower bound. However, for the reasons given in Section 4, this improvement will not be reflected in its average case behavior, though in that case a nonsignificant improvement over the classical mergesort algorithm can be achieved by choosing appropriate techniques for heap creation and merging.

REFERENCES

1. A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, (1974).
2. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Galgotia Publications Pvt. Ltd., New Delhi, (1995).
3. D.E. Knuth, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA, (1973).
4. K. Mehlhorn and A. Tsakalidis, Data structures, In *Handbook of Theoretical Computer Science*, (Edited by J. van Leeuwen), Ch. 6, pp. 301–241, Elsevier, The Netherlands, (1990).
5. H. Noltemeier, Dynamic partial orders and generalized heaps, *Computing (Suppl.)* **7**, 125–139, (1990).
6. J.W.J. Williams, Algorithm 232, *CACM* **7** (6), 347–348, (June 1964).
7. R.W. Floyd, Algorithm 245, Treesort, *CACM* **7** (12), 701, (December 1964).
8. X. Gu and Y. Zhu, Optimal heapsort algorithm, *Theoretical Computer Science* **163**, 239–243, (1996).
9. S. Carlsson, J. Chen and C. Mattsson, Heaps with bits, *Theoretical Computer Science* **164**, 1–12, (1996).
10. C.J.H. McDiarmid and B.A. Reed, Building heaps fast, *Journal of Algorithms* **10**, 352–365, (1989).
11. I. Wegener, The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$, *Information and Computation* **97**, 86–96, (1992).

12. K. Mehlhorn, *Data Structures and Algorithms, 1: Sorting and Searching*, Springer Verlag, Heidelberg, (1984).
13. J. Katajainen, T. Pasanen and J. Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3**, 27–40, (1996).
14. F.K. Hwang and S. Lin, A simple algorithm for merging two disjoint linearly ordered sets, *SIAM J. Computing* **1**, 31–39, (1972).