SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 607–611 (1981)

A Stable Quicksort

DALIA MOTZKIN

Department of Computer Science, Western Michigan University, Kalamazoo, Michigan 49008, U.S.A.

SUMMARY

A sorting algorithm, called Stable Quicksort, is presented. the algorithm is comparable in speed with the Quicksort algorithm, but is stable. The experimental evidence presented support the theoretical evaluation of the performance of Stable Quicksort.

KEY WORDS Sorting Quicksort Stable sorting Efficient sorting

INTRODUCTION

Algorithms for stable sorting have been developed by Dewar,[1] Horvath,[5] Preparata,[9] Pardo,[10] Rivest[11] and others. A bibliography of stable sorting algorthms can be found in Reference 10. Pardo[10] developed an efficient algorithm for a stable merge sort with $0(N \cdot \log_2 n)$sorting time.

The stable sorting algorithm developed here is based on QUICKSORT, which is one of the best internal sorting methods as far as average computing time and space economy is concerned.

The Quicksort algorithm was first introduced by Hoare.[2,3] The algorithm is both efficient and elegant. It has gained wide acceptance. Many modifications and improvements have been developed. Its performance has been thoroughly analysed. Some of the interesting variants are described in References 12 and 14 to 16. General descriptions and analyses can be found in References 4 and 6. The history and full survey of the many variants is given in Reference 14. Sedgewick[13, 14] and Loeser[7, 8] have provided performance evaluation and analysis of some of the important versions of Quicksort. The above list of references is not exhaustive.

One drawback of Quicksort is that the sort is unstable. The stability property is necessary for certain applications. In this paper, we will describe an algorithm we call Stable Quicksort, which is similar to Quicksort, but it is stable, i.e. it preserves the relative ordering of equal keys. The novel feature of the algorithm, which makes stability possible without a significant time penalty, is that the keys to be sorted are stored in a linked list structure rather than an array. We will present the algorithm for positioning one key, explain why it preserves the relative ordering of equal keys and discuss its performance.

THE ALGORITHM

The sorting scheme for Stable Quicksort is presented below. The algorithm is approximately as fast as Quicksort, but stable. The general idea of Quicksort, i.e. that of partitioning is present here too, but the partitioning process is slightly different.

0038–0644/81/060607–05$01.00
© 1981 by John Wiley & Sons, Ltd.

Received 30 June 1980
Revised 4 November 1980

Let the keys of the file to be sorted be k(1), k(2), ..., k(*n*) (*n* being the number of keys). Let a key, say K, be the key controlling the first partitioning process (in the implimentation below, K is the first key of the file to be processed). At the end of the first partitioning process, the key K will become the *s*th key, and the following will be satisfied:

The *i*th key is less than K if $i < s$.

The *j*th key is greater than or equal to K if $j > s$.

The partitioning processes in Stable Quicksort differs from standard Quicksort in that the partitioning process here is effected by a one directional scan rather than the usual alternating scan, which compared keys from both ends of the file to the key controlling the partitioning process. Here, as the file is scanned, keys smaller than K, the key controlling the partitioning process, are deleted from the file and added to the tail of a new file, which ultimately becomes the left sub-file. Thus, at the end of the process, the keys in the left sub-file are in their original order relative to one another, and the same is true also of the right sub-file. As we mentioned above, this process does not impose any significant time penalty, since the keys are in a linked list structure rather than in an array, thus, there is no moving of lists of keys as the keys are 'inserted' at the tail of the left sub-file, but rather relinking of the keys. After the partitioning has been made, the two sub-files are sorted independently using the same procedure.

Consider, for example, the following set of keys:

k(1)	k(2)	k(3)	k(4)	k(5)	k(6)
15	5	7	20	3	17

The initial set of links is:

L(0)	L(1)	L(2)	L(3)	L(4)	L(5)	L(6)
1	2	3	4	5	6	0

If 15 is selected to be K, the key controlling the partitioning process, then at the end of the first partitioning stage, the keys remain in their original locations and the linked list becomes:

L(0)	L(1)	L(2)	L(3)	L(4)	L(5)	L(6)
2	4	3	5	6	1	0

The algorithm for positioning one element is described below.

Algorithm to position one element

Variable Description

KEY—an array containing the keys.

SUCC(J)—the index of the key which succeeds KEY(J). (SUCC(J) is obtained through a linked list, i.e. the successor of KEY(J) is in the Jth location of the linked list array).

ADJACENTL—The index of the key preceding the sub-file to be sorted (i.e. the index of the key which is adjacent, at the left, to the sub-file to be partitioned).

ADJACENTR—The index of the key adjacent from the right to the file to be partitioned (i.e. the index of the key immediately succeeding the file).

LAST—The index of the last key of the sub-file, before the partitioning process is started.

P—The index of the key controlling the partitioning process.
RUN—The index of the key currently being compared to KEY(P) (running index).
PREDP—The index of the predessor of KEY(P).
PREDRUN—The index of the predessor of KEY(RUN).

The input to the algorithm is the array KEY, the linked list, and the three values: ADJACENTL, P, and LAST. At the beginning, before the first partitioning, ADJACENTL = 0, P = 1, LAST = N and SUCC(ADJACENTL) = 1. At the end of each partitioning stage, the left sub-file is delimited by the keys whose indices are SUCC(ADJACENTL), and PREDP: the right sub-file is delimited by the keys whose indices are SUCC(P) and PREDRUN. For each sub-file which contains more than one key, an entry is added to the stack. This entry is composed of three indices: the indices of the first and last keys of the sub-file and the index of the key preceding the first key of the sub-file. The procedure is repeated for each sub-file until the stack is empty.
The Algorithm:

```
ADJACENTR ← SUCC(LAST)
PREDP ← ADJACENTL
RUN ← SUCC(P)
WHILE RUN ≠ ADJACENTR DO
  IF KEY(RUN) < KEY(P)
    THEN SUCC(PREDP) ← RUN
      IF SUCC(P) = RUN
        THEN SUCC(P) ← SUCC(RUN)
        ELSE SUCC(PREDRUN) ← SUCC(RUN)
      PREDP ← RUN
      RUN ← SUCC(RUN)
      SUCC(PREDP) ← P
    ELSE PREDRUN ← RUN
      RUN ← SUCC(RUN)
End of WHILE
End of partitioning of one sub-file
```

PROOF OF THE STABILITY OF THE 'STABLE QUICKSORT'

Lemma. 'Stable Quicksort' is stable.

Denote the keys of the unsorted file by $k_1, k_2, \ldots, k_n$.

We will prove the following by induction. Let the keys k_i, k_j satisfy $k_i = k_j$, $i < j$. Then at the end of any partitioning stage k_i precedes k_j

Proof

First consider the first partitioning step.

Let k_1 be the partitioning key. If $k_1 \leqslant k_i, k_j$, then k_i, k_j will not change their relative positions during this partitioning step. At the end, when k_1 is positioned in place, both k_i and k_j will be in the right hand side unsorted sub-file with k_i still preceding k_j.

If $k_1 > k_i, k_j$ then, according to the algorithm, k_i and k_j will be positioned to the left of k_1, but k_i will be placed there first and, therefore, it will precede k_j.

If $i = 1$, then k_i is the key controlling the partitioning. Since $k_j = k_i$, k_j will not be repositioned during this partitioning step and will remain to the right of k_i. Therefore, again, k_i and k_j will preserve their original ordering at the end of the first partitioning process.

Suppose that at the end of the mth partitioning step the induction hypothesis holds. Consider the $m+1$ partitioning step. If k_i and k_j are not in the same sub-file at the end of the mth partitioning step, then, since the partitioning procedure repositions records only within the same sub-file (splitting it to two new sub-files), the relative ordering of k_i, k_j will not be affected by this partitioning and k_i will precede k_j at the end of the $m+1$ partitioning step regardless of which sub-file was processed during this partitioning step.

If k_i and k_j are in the same sub-file to be sorted during the $m+1$ partitioning step, then the same reasoning as used for the first partitioning step would apply here too. Finally, if k_i and k_j are in the same sub-file and their sub-file is not being partitioned during the $m+1$ partitioning step, then obviously k_i and k_j are not affected by the partitioning and k_i will still precede k_j at the end of the $m+1$ partitioning step. This concludes the proof by induction, i.e. at the end of the final partitioning step k_i still preceded k_j. This proves that Stable Quicksort preserves the ordering of any two equal keys.

Clearly, any set of more than two equal keys would preserve its relative ordering. Thus, the sort is stable.

PERFORMANCE EVALUATION

The average expected time, T, required to sort a file of n records using the Stable Quicksort would be $T \leqslant k \cdot n \cdot \log_e n$ for $n > 2$, where k is a constant. the notation and the proof found in Reference 4 would apply here too, since the proof is based on the partitioning procedure which is also the basic idea of Stable Quicksort. The experimental results support the theoretical evaluation. The following is a summary of the experimental results:

The Stable Quicksort was examined on all $n!$ permutations of n keys for $n = 7$, $n = 8$, $n = 9$ and the averages were calculated (see Table I).

Table I

Number of keys	Average number of comparisons	Average number of times a key was moved	Average number of partitioning steps	Average size of stack
7	13·4	6·7	4·3	1·9
8	16·9	8·4	5	2·6
9	20·5	10·2	5·6	2·9

Note, however, that this algorithm requires some additional space to store the links.

CONCLUDING REMARKS

It has been shown that 'Stable Quicksort' is a stable sort, with performance similar to traditional quicksort. This algorithm requires some additional space for the storage links. Two important variants of Quicksort (surveyed in reference 14 are: (1) sorting smaller sub-files first, and (2) using insertion sort for smaller sub-files. techniques for incorporating these variants into Stable Quicksort, by adding counters and by using linked list insertion sort, are being investigated now.

REFERENCES

1. Robert B. K. Dewar, 'A stable minimum storage algorithm', *Information Processing Letts.*, **2** 162–164 (1974).
2. C. A. R. Hoare, 'Partition (Algorithm 63); Quicksort (Algorithm 64); and Find (Algorithm 65)', *Comm. ACM,* **4** (7) 321–322 (1961).
3. C. A. R. Hoare, 'Quicksort', *Computer J.*, **5** (4) 10–15 (1962).
4. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, California, 1976, pp. 347–350.
5. E. C. Horvath, 'Efficient minimum extra space stable sorting', *Proc. 6th Annual Symp. of Theory of Computing* (SIGACT 6), Association for Computing Machinery, New York, 1974, pp. 194–215.
6. D. E. Knuth, 'Sorting and searching', in *The Art of Computer Programming 3*, Addison-Wesley, 1972.
7. R. Loeser, 'Some performance tests of "quicksort" and descendents', *Comm. ACM,* **17** (3), 143–152 (1974).
8. R. Loeser, 'Survey on Algorithms 34, 422, and Quicksort', *ACM Transactions on Mathematical Software*, **2** (3), 290–299 (1976).
9. F. P. Preparatà, 'A fast stable sorting algorithm with absolutely minimum storage', *Theorem of Computer Science*, **1**, 185–190 (1975).
10. L. T. Pardo, 'Stable sorting and merging with optimal space and time bounds', *Siam J. Comput.*, **6** (2), 351–371 (1977).
11. Ronald Rivest, 'A fast stable minimum storage algorithm, Rep. 43', *Institute of Recherche d'Informatique et d'Automatique*, Rocquencourt, France, Dec. 1973.
12. R. D. Scowen, 'Quickersort (Algorithm 271)', *Comm. ACM,* **8** (11) (Nov. 1965).
13. R. Sedgewick, *Quicksort with Equal Keys*, Technical Report No. CS-8, Computer Science Program and Division of Applied Mathematics, Brown University, August 1975.
14. R. Sedgewick, 'The analysis of Quicksort programs', *Acta Informatica,* **7**, 327–355 (1977).
15. R. C. Singleton, 'An efficient algorithm for sorting with minimal storage (Algorithm 347)', *Comm. ACM*, **12** (3), 185–187 (1969).
16. M. N. van Emden, 'Increasing the efficiency of quicksort (Algorithm 402)', *Comm. ACM,* **13** (11), 693–694 (1970).