An Improved HEAPSORT Algorithm with $n \log n - 0.788928n$ Comparisons in the Worst Case

Xiao-Dong Wang (王晓东) and Ying-Jie Wu (吴英杰)

College of Mathematics and Computer Science, Fuzhou University, Fuzhou 350002, China

E-mail: wangxd@fzu.edu.cn; yjwu@fzu.edu.cn

Received September 30, 2006; revised May 8, 2007.

Abstract A new variant of HEAPSORT is presented in this paper. The algorithm is not an internal sorting algorithm in the strong sense, since extra storage for $n$ integers is necessary. The basic idea of the new algorithm is similar to the classical sorting algorithm HEAPSORT, but the algorithm rebuilds the heap in another way. The basic idea of the new algorithm is it uses only one comparison at each node. The new algorithm shift walks down a path in the heap until a leaf is reached. The request of placing the element in the root immediately to its destination is relaxed. The new algorithm requires about $n \log n - 0.788928 n$ comparisons in the worst case and $n \log n - n$ comparisons on the average which is only about $0.4n$ more than necessary. It beats on average even the clever variants of QUICKSORT, if $n$ is not very small. The difference between the worst case and the best case indicates that there is still room for improvement of the new algorithm by constructing heap more carefully.

Keywords data structures, analysis of algorithms, heaps, HEAPSORT

1 Introduction

Sorting is one of the most fundamental problems in computer science. In this paper only general and sequential sorting algorithms are studied. All results should be compared with the simple lower bound\cite{1}

$$\log(n!) = n \log n - n \log e + \Theta(\log n) \approx n \log n - 1.442695n,$$

for the worst and average case numbers of comparisons of general sorting algorithms. With respect to this lower bound, sorting by merging and sorting by insertion and binary search are very efficient. HEAPSORT\cite{2,3} needs $2n \log n$ comparisons. HEAPSORT is in almost all cases less efficient than QUICKSORT. All versions of QUICKSORT are inefficient in the worst case but efficient in the average case.

Let $H(n) = 1 + 1/2 + \cdots + 1/n$ be the $n$-th harmonic number, $Q(n)$ the average number of comparisons of QUICKSORT, and $CQ(n)$ the average number of comparisons of the best-of-three variant of QUICKSORT called CLEVER-QUICKSORT. Then\cite{4}

$$Q(n) = 2(n+1)H(n) - 4n \approx 1.386n \log n - 2.846n$$

and for $n \geq 6$

$$CQ(n) \approx 1.188n \log n - 2.255n.$$

Because of these results HEAPSORT has been considered for a long time only for theoretical reasons. Carlsson\cite{5} presented a new variant of HEAPSORT whose average and worst-case complexity is $n \log n + \Theta(n \log \log n)$. This algorithm does not beat CLEVER QUICKSORT on average for $n \leq 10^{10}$. Another variant of HEAPSORT is called BOTTOM-UP-HEAPSORT\cite{3,6}. The worst-case number of comparisons of the algorithm is about $1.5n \log n - 0.4n$\cite{4}.

In this paper a new variant of HEAPSORT algorithm is presented. The new algorithm is not an internal sorting algorithm in the strong sense, since extra storage for $n$ integers is necessary. The basic idea of the new algorithm is similar to the classical sorting algorithm HEAPSORT, but the algorithm rebuilds the heap in another way. The idea of the new algorithm is it uses only one comparison at each node. With one comparison we can decide which child of the node just considered contains the larger element and this child is promoted directly to the position of its parent. In this way, the algorithm shift walks down a path in the heap until a leaf is reached. In the new algorithm, the request of placing the element in the root immediately to its destination is relaxed. Therefore the algorithm saves the comparisons for rebuilding the heap.

In Section 2 we present the algorithm and discuss some details of its implementation. In Section 3 we prove that the worst-case number of comparisons
of the new algorithm is remarkably small: it can be bounded by $n \log n - 0.788928n$. In Section 4 we estimate the average complexity of the new algorithm. The number of comparisons of the new algorithm is about $n \log n - n$ on average. The new algorithm is compared with other practical sorting algorithms. We finish the paper with conclusions in Section 5.

2 New Algorithm

Let $a[1..n]$ be an array of $n$ elements of a key and some information associated with this key. This array is a (maximum) heap if, for all $i \in \{2, \ldots, n\}$, the key of element $a[i/2]$ is larger than or equal to that of element $a[i]$. That is, a heap is a pointer free representation of a binary tree, where the elements stored are partially ordered according to their keys. Element $a[1]$ with the largest key is stored at the root. Elements $a[i/2], a[2i]$ and $a[2i + 1]$ (if they exist) are respectively stored at the parent, the left child and the right child of the node at which element $a[i]$ is stored. If a node has no children then the node is a leaf, otherwise the node is an internal node.

HEAPSORT is a classical sorting algorithm that is described in almost all algorithmic textbooks. Generally the HEAPSORT algorithm can be divided into two phases: the heap creation phase and the selection phase. HEAPSORT sorts the given elements in ascending order with respect to their keys as follows:

Input: Array $a[1..n]$ of $n$ elements.

Output: The elements in $a[1..n]$ in sorted order.

void HEAPSORT()

{ buildheap();
 for (int $i = n; i > 1; i = i - 1$)
 { swap(a[1], a[i]);
   heapify(i - 1);
 }
}

In the heap creation phase, the algorithm build-heap rearranges the input array $a[1..n]$ into a heap. In the selection phase, the algorithm heapify starts at the root and stops if the element at the node just considered is not smaller than the elements at its two children. Otherwise it interchanges the element at the node just considered with the larger element of the two children elements and considers the corresponding child. Algorithm heapify needs two comparisons at each node in order to compute the maximum of the three elements at the node and its two children.

The basic idea of the new algorithm is similar to HEAPSORT, but the algorithm heapify in another way. The new heapsort algorithm RANK_HEAPSORT can be described as follows:

void RANK_HEAPSORT()

{ buildheap();
 for (int $i = n; i > 1; i = i - 1$) shift(i);
   rearrange();
 }

The core of the new algorithm is the algorithm shift. The algorithm heapify uses two comparisons at each node, while the algorithm shift uses only one comparison at each node. With one comparison we can decide which child of the node just considered contains the larger element and this child is promoted directly to the position of its parent. In this way, the algorithm shift walks down a path in the heap until a leaf is reached. This path will be called special path, and the last element on this path will be called special leaf.

When the special leaf $a[k]$ is reached in the algorithm shift($i$), the elements in $a[0]$ and $a[k]$ are swapped and the rank of the element in $a[k]$ is now $i$. To record the rank of the element in $a[k]$, an extra array rank[1..n] is used. In the new algorithm, the request of placing the element in $a[0]$ immediately to its destination $a[i]$ is relaxed. The element in $a[0]$ is placed in $a[k]$ instead and its rank is stored in rank[$k$]. The values of $\text{rank}[i], i \in \{1, \ldots, n\}$, are initialized with value 0. Once the value of $\text{rank}[i], i \in \{1, \ldots, n\}$, is set to a value larger than 1, then the element in $a[k]$ is no longer an element of the current heap. In this way, the array rank[1..n] is also used to indicate the elements of the current heap. The value of rank[0] is set to $n + 1$ and hence the element in $a[0]$ is always not an element of the current heap.

In the classical algorithm HEAPSORT, elements $a[i/2], a[2i]$ and $a[2i + 1]$ (if they exist) are respectively stored at the parent, the left child and the right child of the node at which element $a[i]$ is stored. In our new algorithm RANK_HEAPSORT, the relations are no longer hold. The unranked elements of the array still have the property that the key of an element is larger than or equal to that of its children. That is $a[\text{rank}[i/2]] \geq a[\text{rank}[i]]$ for all $i$. We call unranked elements of this array a pseudo heap.

The algorithm shift can be described as follows:

void shift(int index)

{ int $i = 0$;
   while(internal($i$))
   { $i = \text{maxchild}(i); a[i/2] = a[i];$}
   $a[i] = a[0]; \text{rank}[i] = index;$
 }

In the algorithm, the function internal($i$) is used to test whether the node $a[i]$ is an internal node or not.