A New Variant of In-Place Sort Algorithm

Xiaodong Wang\textsuperscript{a,b\*}, Yingjie Wu\textsuperscript{b}, Daxin Zhu\textsuperscript{a}

\textsuperscript{a}Quanzhou Normal University, Quanzhou 362000, China
\textsuperscript{b}Fuzhou University, Fuzhou 350000, China

Abstract

This paper presents a new variant of in place sort algorithm, an improved Heapsort algorithm. In the worst case, the improved algorithm requires \(n \log n + n \log^* n + O(n)\) comparisons and \(n \log n + O(n)\) moves.

© 2011 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of Harbin University of Science and Technology. Open access under CC BY-NC-ND license.

Keywords: heap; algorithm; complexity

1. Introduction

In the in-place sort algorithm Heapsort, the array to be sorted has \(n\) elements which are initially arranged in the form of a heap (heap creation phase) with the property that (1) the \(i\)-th element of the array will have the \((2i)\)-th and the \((2i+1)\)-th elements (if they exist) as its left and right children respectively and (2) every element in the heap is larger than or equal to the larger of its two children. The heap creation phase requires \(O(n)\) time. After the heap is so created, the largest element will be at the root. Then the root element changes place with the last element in the array and the heap is rearranged with one element less. That means the last position of the heap is considered to be deleted. This is repeated \(n\) times, each time at \(O(\log n)\) cost. [1,3,8]
The rearrangement procedure proceeds as follows. At the beginning, the root contains a former leaf element. This element is swapped with the larger of its new children and this is repeated until it is larger than both of its children or it is a leaf. In the procedure two comparisons are made at each level. Therefore, the heapsort algorithm requires $2n \log n + O(n)$ comparisons and $n \log n + O(n)$ moves.

2. A New Algorithm

In this paper, we present a different rearrangement procedure. In our new procedure, we first search the destination position for the former leaf element along the sinking path on which the sinking element would sink in the original rearrangement procedure. After its destination is identified, the former leaf element is moved directly from the root to its destination position along the sinking path.

Suppose the array $A[1..n]$ is arranged in the form of a heap.

Suppose $A[k] (k \leq n)$ is an arbitrary node of the heap $A$. The elements $A[1], A[i_1], \ldots, A[k]$ on the path from $A[1]$ to $A[k]$ are in sorted order.

For each location $i (1 \leq i \leq n)$ of the array $A$, if we mark it with $m(i)$ such that
\[
m(i) = \begin{cases} 
0, & i = 2j, j = 1,2,\ldots,n/2; \\
1, & i = 2j-1, j = 1,2,\ldots,n/2.
\end{cases}
\]
then, every path of the heap $A$ from the root $A[1]$ to node $A[k]$ can be uniquely represented by its mark sequence $\{m(1),\ldots,m(k)\}$ [2,5,7].

For any positive integer $k (1 \leq k \leq n)$, if its binary representation is denoted by
\[
k = \sum_{i=0}^{\lfloor \log k \rfloor} b_i(k)2^i, \quad b_i(k) \in \{0,1\},
\]
then the sequence $\{b_{\lfloor \log k \rfloor}(k),\ldots,b_0(k)\}$ is the mark sequence of the path from $A[1]$ to $A[k]$ on the heap $A$, and vice versa.

This can be proved by induction on the path length $h = \lfloor \log k \rfloor$.

The theorem is trivially true if the length of the path from $A[1]$ to $A[k]$ is 0 or 1. Suppose the theorem is true when the path length is less than $h$. In the case of the path length is $h = \lfloor \log k \rfloor$, the parent of the node $A[k]$ is $A[\lfloor k/2 \rfloor]$. From the binary representation of $k$ we know,
\[
k/2 = \sum_{i=1}^{\lfloor \log k \rfloor} b_i(k)2^{i-1} + b_0(k)/2 = \sum_{i=0}^{\lfloor \log k \rfloor-1} b_{i+1}(k)2^i + b_0(k)/2
\]
Therefore, \[ \left\lfloor \frac{k}{2} \right\rfloor = \sum_{i=0}^{\lceil \log k \rceil - 1} b_{i+1}(k)2^i . \]

Since the length of the path from A[1] to A[\left\lfloor \frac{k}{2} \right\rfloor] is \( h - 1 \), we have \( (b_{\lceil \log k \rceil}(k), \ldots, b_{1}(k)) \) is the mark sequence of the path from A[1] to A[\left\lfloor \frac{k}{2} \right\rfloor]. On the other hand, \( b_0(k) = 0 \) if \( k \) is even, and \( b_0(k) = 1 \) if \( k \) is odd. It follows that \( (b_{\lceil \log k \rceil}(k), \ldots, b_{1}(k), b_0(k)) \) is the mark sequence of the path from A[1] to A[k].

Suppose \( A[i_1], A[i_2], \ldots, A[i_k] \) is a path of the heap \( A \) from \( A[i_1] \) to \( A[i_k] \). For any \( 1 \leq j \leq k \), we have \( i_j = i_k \text{shr} (k - j) \), where \text{shr} is shiftright operation.

The binary representation of \( i_k \) and \( i_j \) can be written as

\[
i_k = (b_{\lceil \log i_1 \rceil}, b_{\lceil \log i_2 \rceil-1}, \ldots, b_0), \quad i_j = (c_{\lceil \log i_1 \rceil}, c_{\lceil \log i_2 \rceil-1}, \ldots, c_0)\]

where, \( b_i \in \{0,1\}, \quad i = 0, \ldots, \lceil \log i_k \rceil, \quad c_j \in \{0,1\}, \quad j = 0, \ldots, \lceil \log i_j \rceil \)

Since \( A[i_j] \) is on the path from root \( A[1] \) to \( A[i_k] \), it follows from theorem 2 that

\[
b_{\lceil \log i_j \rceil} = c_{\lceil \log i_j \rceil} = 1, \quad \lceil \log i_k \rceil = \lceil \log i_j \rceil + k - j, \quad b_i = c_{i+j-k}, \quad k - j \leq i \leq \lceil \log i_k \rceil .
\]

Thus,

\[
i_k = (c_{\lceil \log i_1 \rceil}, c_{\lceil \log i_2 \rceil-1}, \ldots, c_0, b_{k-j-1}, b_{k-j-2}, \ldots, b_0).
\]

Now it follows that \( i_k \text{shr} (k - j) = (c_{\lceil \log i_1 \rceil}, c_{\lceil \log i_2 \rceil-1}, \ldots, c_0) = i_j \).

From results above we know, elements on the path from A[1] to A[k] are in sorted order and the location of the i-th element on the path in array A can be determined by shiftright operation in \( O(1) \) time.


Once the destination position \( m \) for A[n] is determined, A[n] can be moved directly to its destination position by the following procedure.

\begin{verbatim}
procedure insert(position,bottom)
begin
   temp:=a[1];
   if position=1 then current:=position
   else
\end{verbatim}
begin
for i:=log(position)-1 downto 0 do
begin
  current:=position shr i;
  a[current div 2]:=a[current]
end;
end;
a[current]:=a[bottom];
a[bottom]:=temp
end;

Based on the discussions above we can describe our new rearrangement procedure by a single statement as follows
insert(search(1,n),n);

In the statement function search(1,n) returns the destination position for A[n] in the new heap A[1..n-1] in a recursive manner. Suppose the height of the current heap is $h$. The algorithm finds the path of maximum children down $f(h)$ levels to A[i]. If A[n]>A[i] then performs a binary search with A[n] along the path of length $f(h)$ to find its destination position recursively apply the method starting at location A[i]. Since the number of comparisons required by the procedure is at least linear on $h$, we take $f(h) = h - \log h$ to balance $h - f(h)$ and $\log(f(h))$.

3. Complexity

The correctness of new algorithm can be proved by induction on the for statement of the algorithm.

The cost of the algorithm can be divided into two parts.

(1). The cost of the heap creation phase which requires $O(n)$ time.

(2). The cost of for statement of the algorithm which is determined by the cost of insert procedure.

The algorithm requires $T(h) \leq h + \log^* h + 1$ comparisons in the worst case, where $h$ is the height of the current heap and $\log^* x = 0$ for $x < 1$ and $\log^* n = \log^* (\log n) + 1$.

The correctness of the algorithm can be proved by induction on the depth of recursion. In the algorithm, we spend one comparison per level to find the path of maximum children down $h - \log h$ levels to A[i], and then spend $\log(h - \log h) + 1$ comparisons to perform a binary search with A[n] along this path to its destination position if A[n]>A[i], or otherwise we recursively apply the method starting at location A[i]. This leads to the recurrence
\[ T(h) = h - \log h + 1 + \begin{cases} 
\log(h - \log h) + 1 \\
T(\log h) 
\end{cases} \]

where \( T(h) \) denotes the number of comparisons required by the algorithm.

Since \( T(\log h) > \log(h - \log h) + 1 \), we have \( T(h) \leq h - \log h + 1 + T(\log h) \). Let \( k_{\text{max}} \) be the maximum depth of recursion. In other words \( k_{\text{max}} = \log^* h \). Therefore,

\[ T(h) \leq h - \log h + 1 + \log h - \log \log h + 1 + \ldots = h + k_{\text{max}} + 1 = h + \log^* h + 1. \]

From theorem 4 we know, the number of comparisons required by the \textbf{for} statement of the algorithm is bounded above by

\[ \sum_{i=2}^{n} (\log i + \log^* (\log i) + 1) = n \log n + n \log^* n + O(n). \]

The number of moves required by the algorithm is clearly

\[ \sum_{i=2}^{n} \lfloor \log(i - 1) \rfloor = n \log n + O(n). \]

Finally we can conclude that the improved algorithm requires \( n \log n + n \log^* n + O(n) \) comparisons and \( n \log n + O(n) \) moves in the worst case.

References

[6] Z. Li, and Bruce A. Reed, Heap Building Bounds, LNCS 3608(2005), 14-23.