Heapsort with $n \log(n+1) + n - 2\log(n+1) - 2$ Key Comparisons Using $\lfloor n/2 \rfloor$ Additional Bits

S. Haldar

RUU-CS-93-14
April 1993

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Heapsort with $n \log(n+1) + n - 2\log(n+1) - 2$ Key Comparisons Using $\lfloor n/2 \rfloor$ Additional Bits

S. Haldar

Technical Report RUU-CS-93-14
April 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Heapsort with $n \log(n+1) + n - 2\log(n+1) - 2$ Key Comparisons Using $\lfloor n/2 \rfloor$ Additional Bits*

S. Haldar
Department of Computer Science
University of Utrecht, PO Box 80.089
3508 TB Utrecht, The Netherlands

Abstract

Heapsort is one of the most studied sorting algorithms. In the classical heapsort of Floyd [3], $2n \log n$ key comparisons are needed in the worst case to sort n elements. A variant of heapsort, called *bottom-up-heapsort* [9], uses $1.5n \log n$ key comparisons in the worst case. McDiarmid and Reed [8] propose an interesting variant of bottom-up-heapsort using $2\lfloor (n-1)/2 \rfloor$ additional bits. Later, Wegener [10] shows that with this variation n elements can be sorted within $n \log n + 1.1n$ key comparisons in the worst case. Furthermore, it uses $O(n \log n)$ 2-bit variable comparisons. In this paper we propose another variant of bottom-up-heapsort using $\lfloor n/2 \rfloor$ additional bits. This variant uses $n \log(n+1) + n - 2\log(n+1) - 2$ key comparisons in the worst case to sort $n = 2^k - 1$ elements. No additional bit comparisons are needed.

Index Terms: Bottom-up-heapsort; heapsort; key comparison; sorting; worst case analysis.

1 Introduction

Sorting is a fundamental problem in computer science. A *sorting algorithm* describes how an input sequence $\{a_1, a_2, \ldots, a_n\}$ of n elements can be transformed into an output sequence $\{b_1, b_2, \ldots, b_n\}$ such that each b is a distinct element in the input sequence and for $1 \leq i < n$, $b_i \leq b_{i+1}$. The elements will also be referred to as *keys* in this paper.

There are many sorting algorithms available in the literature. We are interested in those (sequential) algorithms in which key comparison is the only means to order any two elements.

*This research is partially supported by NWO through NFI Project ALADDIN under Contract Number NF 62-376, and partially by ESPRIT through Project ALCOM II under Basic Research Action Number 7141.

The lower bound of key comparisons in the worst case, as well as in the average case, is $\log n! \approx n \log n - 1.44n + \Theta(\log n)$. With respect to this lower bound result, *merge sort* and *insertion sort* are quite efficient. But, the merge sort uses n extra space to store keys, and the insertion sort does $\Theta(n^2)$ key transports (see [7]).

Among the key comparison based sorting algorithms, *quicksort* [6] has been widely used as a general purpose sorting algorithm. Quicksort uses $(n^2 - n)/2$ comparisons in the worst case, and $1.386294n \log n - 2.845569n + 1.3863 \log n + 1.154$ in the average case [4]. There are many variants of the original quicksort available in the literature. The best of them, called *clever quicksort*, uses $1.188252n \log n - 2.255385n + 1.18825 \log n + 2.507$ comparisons in the average case.

Another widely studied sorting algorithm *heapsort* is originally proposed by William [11]. (Heapsort is of particular interest in this paper.) A heap of size n is an array $a[1..n]$ containing n elements satisfying the following conditions: (1) each component of the array stores exactly one element; (2) the array represents a binary tree, completely filled on all levels except possibly at the lowest, which is filled from the left up to a point; (3) the root of the tree is $a[1]$; (4) for a node i in the binary tree, $a[i]$ is its key, $parent(i) = \lfloor i/2 \rfloor$ is its parent, and $2i$ and $2i+1$ are its children, if they exist; (5) the heap property is, for all $2 \leq i \leq n$, $a[parent(i)] \geq a[i]$. Thus, the largest element in a heap is always at the root of the heap.

There are two phases in any heapsort algorithm. First, the input array is transformed into a heap. Secondly, the element at the root is exchanged with the last element of the heap, and the heap is rearranged to build a new heap with one fewer element. This is the most important phase, and repeated ($n-2$ times) until the input array is entirely sorted.

The algorithm of William [11] uses $n \log n + O(n)$ comparisons in the worst case (and about $1.7645n$ for sufficiently large n in the average case [1]) to build a heap on n elements, and more than $2n \log n$ comparisons in the worst case to sort the elements. Later in the same year 1964, Floyd [3] improves William's algorithm. (The Floyd algorithm is shown in Figure 1.) His algorithm uses $2n$ comparisons in the worst case ($1.88n$ in the average case [7]) to build a heap. The sorting phase requires at most $2n \log n$ comparisons. (To sort $n = 2^k - 1$ elements, it uses a total of $2n \log(n+1) - 2n + 2\log(n+1) + O(1)$ comparisons.) The average case is hardly better than the worst case. Because of such high comparisons, the use of heapsort has been of less practical interest in comparison to quicksort, and restricted to implementing mainly priority queues.

In this paper we will study some variants of heapsort, which have been developed in recent years. These variants are as good as, in fact better than, quicksort. Lately, the heap creation time is improved by Gonnet and Munro [5] and McDiarmid and Reed [8]. The algorithm of Gonnet and Munro uses $1.625n$ comparisons in the worst case ($1.5803n$ in the average case) to build a heap, and the one of McDiarmid and Reed uses $2n$ comparisons in the worst case ($1.5203n$ in the average case). The latter algorithm uses $2\lfloor(n-1)/2\rfloor$ additional bits.

It is clear that a heap can be built by $O(n)$ comparisons. So the critical part of any heapsort is the sorting phase. A variant of classical heapsort is presented by Carlson [2] to reduce the number of key comparisons in the sorting phase. In the heap creation phase, the algorithm of Carlson uses $1.82n - O(\log n)$ comparisons in the worst case. It uses $(n+1)(\log(n+1) + \log\log(n+1))$ comparison in the sorting phase, thereby requiring a total of $(n+1)(\log(n+1) + \log\log(n+1) + 1.82) + O(\log n)$ comparisons to sort n elements. It has been pointed out in [9] that this algorithm is better than the clever quicksort on average if $n > 10^{16}$, and hence, is less practical. Another variant of classical heapsort, called *bottom-up-heapsort* (BUH, in short), is proposed by Wegener [9]. This algorithm uses on the average $1.649302n + \Theta(\log n)$ comparison to create a heap, and to sort n elements it uses $1.5n\log n - 0.4n$ comparisons in the worst case and $n\log n + nf(n)$, $f(n) \in [0.355, 0.39]$, in the average case. It is better than the quicksort if $n \geq 400$, and the clever quicksort if $n \geq 1600$. In [8], McDiarmid and Reed propose a variant of BUH, and conjecture that n elements can be sorted using $n\log n + O(n)$ comparisons for this variant. (We call this variant MDR-heapsort.) Later, Wegener [10] proposes an algorithm for this variant that uses $(n+1)\log n + 1.086072n$ comparison in the worst case. (If $n = 2^k - 1$, it uses $(n+1)\log n + n$ comparison in the worst case.) This algorithm uses $2\lfloor(n-1)/2\rfloor$ additional bits, two bits per internal node, and does $O(n\log n)$ 2-bit variable comparisons. For $n \geq 1000$, the worst case number of comparisons is smaller than the average case number of comparisons in the quicksort.

In this paper we present a better algorithm for the heapsort variant suggested by McDiarmid and Reed. Our algorithm uses $\lfloor(n-1)/2\rfloor$ additional bits. It uses at the most $2n$ comparisons in the heap creation phase. It uses a total of $n\log(n+1) + n - 2\log(n+1) - 2$ key comparisons in the worst case to sort $n = 2^k - 1$ elements, and does not do any additional 2-bit variable comparisons. This algorithm is better than the clever quicksort.

The rest of the paper is organized as follows. The bottom-up-heapsort and its variant suggest by McDiarmid and Reed are presented in Section 2. The proposed algorithm is presented in Section 3, and its worst case analysis in Section 4. Section 5 concludes the paper.

2 Bottom-up-heapsort and its variant

Bottom-up-heapsort (BUH, in short) [9] works like a heapsort, but it rearranges the remaining heap in a different way. The algorithm is shown in Figure 2. Every time bottom-up-rearrange procedure is called, it always looks for the leaf that it can reach by starting at the root and going always to the child containing greater element. Let us call this leaf the *special leaf* and the corresponding path the *special path*. Then, it starts climbing up the special path starting from the special leaf. The climbing process continues until it finds an element in the special path, which is greater than root element. Let the position found be j. Then all elements $root = a[i], \ldots, a[j]$ in the special path are cyclically left shifted. As the bottom-up-rearrange constructs the same heap as the rearrange (Cf. Figure 1), the correctness of BUH follows from the correctness of the classical heapsort. It has been shown in [9] that BUH uses $1.5n \log n + O(n)$ key comparisons in the worst case to sort n elements.

To reduce the number of comparisons, a new variant of BUH is proposed by McDiarmid and Reed [8]. They have only presented the algorithm for the heap creation phase. The complete algorithm is found in [10] (Cf. Figure 3). This algorithm uses $2\lfloor (n-1)/2 \rfloor$ additional bits, 2 bits per internal node for storing three values: u (unknown), l (left) and r (right). For each internal node j, a 2-bit variable $info[j]$ is used for this purpose. Initially all $info$ is u. If $info[j] = u$, nothing is known about the greater child. If $info[j] = l$ (or r) then the left child contains greater key than the right one (or vice versa). During the leaf search, for node j, if $info[j] \neq u$, then it takes the appropriate branch (greater child) indicated by the $info[j]$ value, that is, there is no need of a key comparison (for its children). If $info[j] = u$, then, obviously, it does need to do a key comparison (as done in BUH) for its children. In the interchange procedure, during the cyclic shift, the corresponding $info$ variables are set to u. It has been shown that the algorithm uses $(n+1)\log n + 1.086072n$ key comparisons in the worst case. In addition to the key comparisons, it also does $O(n \log n)$ 2-bit $info$ comparisons.

3 The proposed algorithm

In this section we present an efficient technique of implementing the algorithm presented in the above section. Our algorithm uses fewer additional bits for internal nodes, one bit per node, that is, a total of $\lfloor n/2 \rfloor$ additional bits. Here, the $info$ variables are 1-bit variables, and store 0 or 1. In the leaf search phase, it does not do any comparisons, neither key nor $info$. The $info$ values are cleverly used in selecting greater child, either left or right child. For a node j, $info[j] = 0$ if $a[2j] \geq a[2j+1]$, and 1 otherwise. We define a function $left(j) = 2j + info[j]$. That is, $left(j)$ always indicates the greater child of j, if it exists. The function $left(j)$ can be efficiently implemented by a real computer as follows: $2j$ is computed by left shifting j by one bit; and then set the least significant bit of $2j$ to *info*$[j]$. Thus, the algorithm does not need to do any key or *info* comparisons during the leaf search. During the cyclic shift, when the elements are moved up the tree, they are compared with their respective right sibling, and the comparison results are stored in the *info* variables of their respective parents. Thus, the algorithm is able to get rid of the undesirable undefined values for *info* variables as used in MDR-heapsort. The complete algorithm is presented in Figure 4.

The correctness of the proposed algorithm is obvious, since rearrange, bottom-up-rearrange, mdr-rearrange and rebuild all construct the same heap. In addition, the proposed algorithm is quite efficient and easy to implement. The worst case analysis is presented in the next section.

4 Worst case analysis

For the sake of simplicity we would assume that there are $n = 2^k - 1$ elements in the input array. Let us define *level* of the root of a heap to be 1. The *level* of the children of each node i is $level(i) + 1$. The *height* h of a heap is the maximum of all *levels* of the nodes in the heap. The $height(i)$ of a node i in the heap is $h - level(i) + 1$. That is, all the leaf nodes are at height 1. As $n = 2^k - 1$, the initial height of the heap is k.

Heap building phase

Let $H(k)$ be the maximum number of key comparisons required to create a complete heap of height k. It is clear from the proposed algorithm that a heap of height k is build from two heaps of height $k - 1$ and one element. In each invocation of build-heap, one key comparison is done in that procedure. No key comparisons are done in leaf-search procedure. The number of key comparisons done in bottom-up-search and interchange procedures is at least $k - 1$ and at most k. Thus, the algorithm uses $k + 1$ comparison in the worst case to build a heap of height k from two heaps of height $k - 1$. Then, we have the following.

$$H(k) = \begin{cases} 2H(k-1) + (k+1), & \text{for } k > 1 \\ 0 & \text{for } k = 1. \end{cases}$$

That is,

$$\begin{aligned} H(k) &= \textstyle\sum_{i=0}^{k-2} (k+1-i)2^i \\ &= (k+1)\textstyle\sum_{i=0}^{k-2} 2^i - \textstyle\sum_{i=0}^{k-2} i2^i \\ &= (k+1)(2^{k-1} - 1) - 2((k-2)2^{k-1} - (k-1)2^{k-2} + 1) \\ &= 2^{k+1} - k - 3. \end{aligned}$$

Selection phase

Let $T(k)$ be the maximum number of key comparisons required in the sorting phase for a complete heap of height k. There are 2^{k-1} elements at height 1, 2^{k-2} elements at height 2, and so on. For an element, if the leaf-search stops at level l, then to insert the element at appropriate position we need exactly l comparisons (see rebuild procedure). When an element of level l is selected, the corresponding leaf-search terminates at level l or $l-1$. For the worst case, we assume that it terminates at level l. Note that for the last element of level l the leaf-search always terminates at level $l-1$. If the leaf-search for the penultimate element of level l terminates at level l, we need not do any key comparison with the last element of level l. Hence, for the last two elements of each level l, the algorithms does at the most $l-1$ comparisons. Finally, it does not require to do any key comparisons for level 1 and 2. Then, we have the following.

$$\begin{aligned}
T(k) &= \{k(2^{k-1}-2)+2(k-1)\}+\{(k-1)(2^{k-2}-2)+2(k-2)\}+\cdots+\{3(2^{3-1}-2)+2(3-1)\}\\
&= \textstyle\sum_{i=3}^{k}(i(2^{i-1}-2)+2(i-1))\\
&= \textstyle\sum_{i=3}^{k}(i2^{i-1}-2i+2i-2))\\
&= \textstyle\sum_{i=3}^{k} i2^{i-1}-\sum_{i=3}^{k} 2\\
&= \textstyle\sum_{i=1}^{k} i2^{i-1}-2.2^{1}-1-\sum_{i=3}^{k} 2\\
&= k2^{k+1}-(k+1)2^{k}+1-5-2(k-2)\\
&= k2^{k}-2^{k}-2k.
\end{aligned}$$

Hence, the total number of key comparisons required in the worst case to sort $n=2^k-1$ elements is

$$\begin{aligned}
H(k)+T(k) &= 2^{k+1}-k-3+k2^{k}-2^{k}-2k\\
&= k2^{k}+2^{k}-3k-3\\
&= (n+1)\log(n+1)+n-3\log(n+1)-2\\
&= n\log(n+1)+n-2\log(n+1)-2.
\end{aligned}$$

5 Conclusion

An efficient algorithm for a variant of heapsort has been presented. The algorithm uses only $\lfloor n/2 \rfloor$ additional bits, and it could be considered as almost internal sorting algorithm. This algorithm is better than quicksort. It is to be noted that in a heap the first two elements, the root and its left child, are greatest elements in the heap. So, they can be output in one pass. Then, instead of taking one leaf element at a time in the selection phase, we could take two leaf elements (of known order) to rebuilt a new heap with two fewer elements. This might further reduce the number of key comparisons. Finally, we would conjecture that n elements could be sorted using some variant of (almost internal) heapsort within $n \log n$ key comparisons.

References

[1] B. Bollobas and I. Simon, 'Repeated random insertion into a priority queue', J. of Algorithms, Vol.6(4), 1985, 466–477.

[2] S. Carlson, 'A variant of heapsort with almost optimal number of comparisons', Information Processing Letters, Vol.24(4), 1987, 247–250.

[3] R. Floyd, 'Algorithm 245: Treesort', Communications of the ACM, Vol.7(12), 1964, 701.

[4] G. Gonnet, 'Handbook of algorithms and data structures', Addison-Wesley, Reading, MA, 1984.

[5] G. Gonnet and J. Munro, 'Heap on heaps', SIAM J. of Computing, Vol.15(4), 1986, 964–971. (Also in Proc. of 9th ICALP, 1982, 282–291.)

[6] C. Hoare, 'Quicksort', Computer Journal, Vol.5, 1962, 10–15.

[7] D. Knuth, 'The art of computer programming, Vol.III, sorting and searching', Addison-Wesley, Reading, MA, 1973.

[8] C. McDiarmid and B. Reed, 'Building heaps fast', J. of Algorithms, Vol.10(3), 1989, 352–365.

[9] I. Wegener, 'Bottom-up-heap sort, a new variant of heap sort beating on average quick sort (if n is not very small)', Proc. of Mathematical Foundation of Computer Science, 1990, LNCS 452, 516–522.

[10] I. Wegener, 'The worst case complexity of McDiarmid and Reed's variant of bottom-up-heap sort in less than $n \log n + 1.1n$', Proc. of the Symp. on Theoretical Aspects of Computer Science, 1991, LNCS 480, 137–147.

[11] J. Williams, 'Algorithm 232: Heapsort', Communications of the ACM, Vol.7, 1964, 347–348.

```
Procedure rearrange($i, m$);
    if $i > \lfloor m/2 \rfloor$ then return;
    if $i = \lfloor m/2 \rfloor$ then $min := \min\{a[i], a[2i]\}$;
    if $i < \lfloor m/2 \rfloor$ then $min := \min\{a[i], a[2i], a[2i+1]\}$;
    if $min = a[i]$ then return {already in heap form}
    elseif $min = a[2i]$ then
        exchange $a[i]$ and $a[2i]$;
        rearrange($2i, m$)
    else
        exchange $a[i]$ and $a[2i+1]$;
        rearrange($2i+1, m$)
end-of-rearrange;

Procedure heapsort($a[1..n]$);
    for $i := \lfloor n/2 \rfloor, ..., 1$ do rearrange($i, n$); {heap creation phase}
    for $m := n, ..., 2$ do                            {sorting phase}
        exchange $a[1]$ and $a[m]$;
        if $m \neq 2$ then rearrange($1, m-1$);
end-of-heapsort;
```

Figure 1: The classical Floyd heapsort.

```
Procedure leaf-search($m, i, j$); {search for the special leaf}
   $j := i$;
   while $2j < m$ do
      if $a[2j] \geq a[2j+1]$ then $j := 2j$ else $j = 2j+1$; {take greater child}
   if $2j = m$ then $j := m$;
end-of-leaf-search; {$j$ is a special leaf}

Procedure bottom-up-search($i, j$);
   while $i < j$ and $a[i] \geq a[j]$ do $j := \lfloor j/2 \rfloor$;
end-of-bottom-up-search;

Procedure interchange($i, j$); {can be efficiently implemented as done in [9]}
   if $i = j$ then return;
   $temp := a[j]$; $a[j] := a[i]$;
   while $j > i$ do
      exchange $temp$ and $a[\lfloor j/2 \rfloor]$
      $j := \lfloor j/2 \rfloor$;
end-of-interchange;

Procedure bottom-up-rearrange($i, m$);
   leaf-search($m, i, j$);
   bottom-up-search($i, j$);
   interchange($i, j$);
end-of-rearrange;

Procedure bottom-up-heapsort($a[1..n]$);
   for $i := \lfloor n/2 \rfloor, ..., 1$ do bottom-up-rearrange($i, n$); {heap creation phase}
   for $m := n, ..., 2$ do                                   {sorting phase}
      exchange $a[1]$ and $a[m]$;
      if $m \neq 2$ then bottom-up-rearrange($1, m-1$);
end-of-heapsort;
```

Figure 2: Bottom-up-heapsort.

```
var
  info : array [1..2⌊(n − 1)/2⌋] of (l, r, u);

Function parent(i) := ⌊i/2⌋;

Procedure leaf-search(m, i, j);
  j := i;
  while 2j < m do
     if info[j] = l then j := 2j
     elseif info[j] = r then j := 2j + 1
     elseif a[2j] ≥ a[2j + 1] then info[j] := l; j := 2j
         else info[j] := r; j := 2j + 1
  if 2j = m then j := m;
end-of-leaf-search; {j is a special leaf}

Procedure bottom-up-search(i, j);
  while i < j and a[i] ≥ a[j] do j := ⌊j/2⌋;
end-of-bottom-up-search;

Procedure interchange(i, j); {can be efficiently implemented as done in [9]}
  if i = j then return;
  temp := a[j]; a[j] := a[i];
  while j > i do
     info[parent(j)] := u
     exchange temp and a[parent(j)];
     j := parent(j);
end-of-interchange;

Procedure mdr-rearrange(i, m);
  leaf-search(m, i, j);
  bottom-up-search(i, j);
  interchange(i, j);
end-of-rearrange;

Procedure mdr-heapsort(a[1..n]);
  for i := ⌊(n − 1)/2⌋, ..., 1 do info[i] := u;
  for i := ⌊n/2⌋, ..., 1 do mdr-rearrange(i, n); {heap creation phase}
  for m := n, ..., 2 do                          {sorting phase}
     exchange a[1] and a[m];
     if m ≠ 2 then mdr-rearrange(1, m − 1);
end-of-heapsort;
```

Figure 3: MDR-heapsort.

```
var
  $info$ : array $[1..\lfloor n/2 \rfloor]$ of 0..1;

Function left($i$) := $2i + info[i]$;

Function parent($i$) := $\lfloor i/2 \rfloor$;

Procedure leaf-search($m, i, j$);
  $j := i$;
  while $2j \leq m$ do $j$ := left($j$);
end-of-leaf-search; {$j$ is a special leaf}

Procedure bottom-up-search($i, j$);
  while $i < j$ and $a[i] \geq a[j]$ do $j := parent(j)$;
end-of-bottom-up-search;

Procedure interchange($i, j$); {can be efficiently implemented as done in [9]}
  if $i = j$ then return;
  $temp := a[j]$; $a[j] := a[i]$;
  while $j > i$ do
    $info[parent(j)]$ := if $a[2 * parent(j)] \geq a[2 * parent(j) + 1]$ then 0 else 1;
    {if $2 * parent(j) + 1 > m$ **then** $info[parent(j)] := 0$};
    exchange $temp$ and $a[parent(j)]$;
    $j := parent(j)$;
end-of-interchange;

Procedure rebuild($i, m$);
  leaf-search($m, i, j$);
  bottom-up-search($i, j$);
  interchange($i, j$);
end-of-rebuild;

Procedure build-heap($i, m$);
  if $2i = m$ then $info[i] := 0$;
  if $2i < m$ then $info[i]$ := if $a[2i] \geq a[2i + 1]$ then 0 else 1;
  leaf-search($m, i, j$);
  bottom-up-search($i, j$);
  interchange($i, j$);
end-of-build-heap;

Procedure proposed-heapsort($a[1..n]$);
  for $i := \lfloor n/2 \rfloor, ..., 1$ do build-heap($i, n$); {heap creation phase}
  for $m := n, ..., 2$ do                            {sorting phase}
    exchange $a[1]$ and $a[m]$;
    $info[\lfloor m/2 \rfloor] := 0$;
    if $m \neq 2$ then rebuild($1, m - 1$);
end-of-heapsort;
```

Figure 4: Proposed algorithm.

Heapsort with $n \log(n+1) + n - 2\log(n+1) - 2$ Key Comparisons Using $\lfloor n/2 \rfloor$ Additional Bits

S. Haldar

RUU-CS-93-14
April 1993

Heapsort with $n\log(n+1)+n-2\log(n+1)-2$ Key Comparisons Using $\lfloor n/2 \rfloor$ Additional Bits

S. Haldar

Technical Report RUU-CS-93-14
April 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Heapsort with $n \log(n+1) + n - 2\log(n+1) - 2$ Key Comparisons Using $\lfloor n/2 \rfloor$ Additional Bits*

S. Haldar
Department of Computer Science
University of Utrecht, PO Box 80.089
3508 TB Utrecht, The Netherlands

Abstract

Heapsort is one of the most studied sorting algorithms. In the classical heapsort of Floyd [3], $2n \log n$ key comparisons are needed in the worst case to sort n elements. A variant of heapsort, called *bottom-up-heapsort* [9], uses $1.5n \log n$ key comparisons in the worst case. McDiarmid and Reed [8] propose an interesting variant of bottom-up-heapsort using $2\lfloor (n-1)/2 \rfloor$ additional bits. Later, Wegener [10] shows that with this variation n elements can be sorted within $n \log n + 1.1n$ key comparisons in the worst case. Furthermore, it uses $O(n \log n)$ 2-bit variable comparisons. In this paper we propose another variant of bottom-up-heapsort using $\lfloor n/2 \rfloor$ additional bits. This variant uses $n \log(n+1) + n - 2\log(n+1) - 2$ key comparisons in the worst case to sort $n = 2^k - 1$ elements. No additional bit comparisons are needed.

Index Terms: Bottom-up-heapsort; heapsort; key comparison; sorting; worst case analysis.

1 Introduction

Sorting is a fundamental problem in computer science. A *sorting algorithm* describes how an input sequence $\{a_1, a_2, \ldots, a_n\}$ of n elements can be transformed into an output sequence $\{b_1, b_2, \ldots, b_n\}$ such that each b is a distinct element in the input sequence and for $1 \leq i < n$, $b_i \leq b_{i+1}$. The elements will also be referred to as *keys* in this paper.

There are many sorting algorithms available in the literature. We are interested in those (sequential) algorithms in which key comparison is the only means to order any two elements.

*This research is partially supported by NWO through NFI Project ALADDIN under Contract Number NF 62-376, and partially by ESPRIT through Project ALCOM II under Basic Research Action Number 7141.

The lower bound of key comparisons in the worst case, as well as in the average case, is $\log n! \approx n \log n - 1.44n + \Theta(\log n)$. With respect to this lower bound result, *merge sort* and *insertion sort* are quite efficient. But, the merge sort uses n extra space to store keys, and the insertion sort does $\Theta(n^2)$ key transports (see [7]).

Among the key comparison based sorting algorithms, *quicksort* [6] has been widely used as a general purpose sorting algorithm. Quicksort uses $(n^2 - n)/2$ comparisons in the worst case, and $1.386294n \log n - 2.845569n + 1.3863 \log n + 1.154$ in the average case [4]. There are many variants of the original quicksort available in the literature. The best of them, called *clever quicksort*, uses $1.188252n \log n - 2.255385n + 1.18825 \log n + 2.507$ comparisons in the average case.

Another widely studied sorting algorithm *heapsort* is originally proposed by William [11]. (Heapsort is of particular interest in this paper.) A heap of size n is an array $a[1..n]$ containing n elements satisfying the following conditions: (1) each component of the array stores exactly one element; (2) the array represents a binary tree, completely filled on all levels except possibly at the lowest, which is filled from the left up to a point; (3) the root of the tree is $a[1]$; (4) for a node i in the binary tree, $a[i]$ is its key, $parent(i) = \lfloor i/2 \rfloor$ is its parent, and $2i$ and $2i + 1$ are its children, if they exist; (5) the heap property is, for all $2 \leq i \leq n$, $a[parent(i)] \geq a[i]$. Thus, the largest element in a heap is always at the root of the heap.

There are two phases in any heapsort algorithm. First, the input array is transformed into a heap. Secondly, the element at the root is exchanged with the last element of the heap, and the heap is rearranged to build a new heap with one fewer element. This is the most important phase, and repeated ($n - 2$ times) until the input array is entirely sorted.

The algorithm of William [11] uses $n \log n + O(n)$ comparisons in the worst case (and about $1.7645n$ for sufficiently large n in the average case [1]) to build a heap on n elements, and more than $2n \log n$ comparisons in the worst case to sort the elements. Later in the same year 1964, Floyd [3] improves William's algorithm. (The Floyd algorithm is shown in Figure 1.) His algorithm uses $2n$ comparisons in the worst case ($1.88n$ in the average case [7]) to build a heap. The sorting phase requires at most $2n \log n$ comparisons. (To sort $n = 2^k - 1$ elements, it uses a total of $2n \log(n + 1) - 2n + 2 \log(n + 1) + O(1)$ comparisons.) The average case is hardly better than the worst case. Because of such high comparisons, the use of heapsort has been of less practical interest in comparison to quicksort, and restricted to implementing mainly priority queues.

In this paper we will study some variants of heapsort, which have been developed in recent years. These variants are as good as, in fact better than, quicksort. Lately, the heap creation time is improved by Gonnet and Munro [5] and McDiarmid and Reed [8]. The algorithm of Gonnet and Munro uses $1.625n$ comparisons in the worst case ($1.5803n$ in the average case) to build a heap, and the one of McDiarmid and Reed uses $2n$ comparisons in the worst case ($1.5203n$ in the average case). The latter algorithm uses $2\lfloor(n-1)/2\rfloor$ additional bits.

It is clear that a heap can be built by $O(n)$ comparisons. So the critical part of any heapsort is the sorting phase. A variant of classical heapsort is presented by Carlson [2] to reduce the number of key comparisons in the sorting phase. In the heap creation phase, the algorithm of Carlson uses $1.82n - O(\log n)$ comparisons in the worst case. It uses $(n+1)(\log(n+1) + \log\log(n+1))$ comparison in the sorting phase, thereby requiring a total of $(n+1)(\log(n+1) + \log\log(n+1) + 1.82) + O(\log n)$ comparisons to sort n elements. It has been pointed out in [9] that this algorithm is better than the clever quicksort on average if $n > 10^{16}$, and hence, is less practical. Another variant of classical heapsort, called *bottom-up-heapsort* (BUH, in short), is proposed by Wegener [9]. This algorithm uses on the average $1.649302n + \Theta(\log n)$ comparison to create a heap, and to sort n elements it uses $1.5n\log n - 0.4n$ comparisons in the worst case and $n\log n + nf(n)$, $f(n) \in [0.355, 0.39]$, in the average case. It is better than the quicksort if $n \geq 400$, and the clever quicksort if $n \geq 1600$. In [8], McDiarmid and Reed propose a variant of BUH, and conjecture that n elements can be sorted using $n\log n + O(n)$ comparisons for this variant. (We call this variant MDR-heapsort.) Later, Wegener [10] proposes an algorithm for this variant that uses $(n+1)\log n + 1.086072n$ comparison in the worst case. (If $n = 2^k - 1$, it uses $(n+1)\log n + n$ comparison in the worst case.) This algorithm uses $2\lfloor(n-1)/2\rfloor$ additional bits, two bits per internal node, and does $O(n\log n)$ 2-bit variable comparisons. For $n \geq 1000$, the worst case number of comparisons is smaller than the average case number of comparisons in the quicksort.

In this paper we present a better algorithm for the heapsort variant suggested by McDiarmid and Reed. Our algorithm uses $\lfloor(n-1)/2\rfloor$ additional bits. It uses at the most $2n$ comparisons in the heap creation phase. It uses a total of $n\log(n+1) + n - 2\log(n+1) - 2$ key comparisons in the worst case to sort $n = 2^k - 1$ elements, and does not do any additional 2-bit variable comparisons. This algorithm is better than the clever quicksort.

The rest of the paper is organized as follows. The bottom-up-heapsort and its variant suggest by McDiarmid and Reed are presented in Section 2. The proposed algorithm is presented in Section 3, and its worst case analysis in Section 4. Section 5 concludes the paper.

2 Bottom-up-heapsort and its variant

Bottom-up-heapsort (BUH, in short) [9] works like a heapsort, but it rearranges the remaining heap in a different way. The algorithm is shown in Figure 2. Every time bottom-up-rearrange procedure is called, it always looks for the leaf that it can reach by starting at the root and going always to the child containing greater element. Let us call this leaf the *special leaf* and the corresponding path the *special path*. Then, it starts climbing up the special path starting from the special leaf. The climbing process continues until it finds an element in the special path, which is greater than root element. Let the position found be j. Then all elements $root = a[i], \ldots, a[j]$ in the special path are cyclically left shifted. As the bottom-up-rearrange constructs the same heap as the rearrange (Cf. Figure 1), the correctness of BUH follows from the correctness of the classical heapsort. It has been shown in [9] that BUH uses $1.5n \log n + O(n)$ key comparisons in the worst case to sort n elements.

To reduce the number of comparisons, a new variant of BUH is proposed by McDiarmid and Reed [8]. They have only presented the algorithm for the heap creation phase. The complete algorithm is found in [10] (Cf. Figure 3). This algorithm uses $2\lfloor (n-1)/2 \rfloor$ additional bits, 2 bits per internal node for storing three values: u (unknown), l (left) and r (right). For each internal node j, a 2-bit variable $info[j]$ is used for this purpose. Initially all *info* is u. If $info[j] = u$, nothing is known about the greater child. If $info[j] = l$ (or r) then the left child contains greater key than the right one (or vice versa). During the leaf search, for node j, if $info[j] \neq u$, then it takes the appropriate branch (greater child) indicated by the $info[j]$ value, that is, there is no need of a key comparison (for its children). If $info[j] = u$, then, obviously, it does need to do a key comparison (as done in BUH) for its children. In the interchange procedure, during the cyclic shift, the corresponding *info* variables are set to u. It has been shown that the algorithm uses $(n+1)\log n + 1.086072n$ key comparisons in the worst case. In addition to the key comparisons, it also does $O(n \log n)$ 2-bit *info* comparisons.

3 The proposed algorithm

In this section we present an efficient technique of implementing the algorithm presented in the above section. Our algorithm uses fewer additional bits for internal nodes, one bit per node, that is, a total of $\lfloor n/2 \rfloor$ additional bits. Here, the *info* variables are 1-bit variables, and store 0 or 1. In the leaf search phase, it does not do any comparisons, neither key nor *info*. The *info* values are cleverly used in selecting greater child, either left or right child. For a node j, $info[j] = 0$ if $a[2j] \geq a[2j+1]$, and 1 otherwise. We define a function $left(j) = 2j + info[j]$. That is, $left(j)$ always indicates the greater child of j, if it exists. The function $left(j)$ can be efficiently implemented by a real computer as follows: $2j$ is computed by left shifting j by one bit; and then set the least significant bit of $2j$ to *info*$[j]$. Thus, the algorithm does not need to do any key or *info* comparisons during the leaf search. During the cyclic shift, when the elements are moved up the tree, they are compared with their respective right sibling, and the comparison results are stored in the *info* variables of their respective parents. Thus, the algorithm is able to get rid of the undesirable undefined values for *info* variables as used in MDR-heapsort. The complete algorithm is presented in Figure 4.

The correctness of the proposed algorithm is obvious, since rearrange, bottom-up-rearrange, mdr-rearrange and rebuild all construct the same heap. In addition, the proposed algorithm is quite efficient and easy to implement. The worst case analysis is presented in the next section.

4 Worst case analysis

For the sake of simplicity we would assume that there are $n = 2^k - 1$ elements in the input array. Let us define *level* of the root of a heap to be 1. The *level* of the children of each node i is $level(i) + 1$. The *height* h of a heap is the maximum of all *levels* of the nodes in the heap. The $height(i)$ of a node i in the heap is $h - level(i) + 1$. That is, all the leaf nodes are at height 1. As $n = 2^k - 1$, the initial height of the heap is k.

Heap building phase

Let $H(k)$ be the maximum number of key comparisons required to create a complete heap of height k. It is clear from the proposed algorithm that a heap of height k is build from two heaps of height $k - 1$ and one element. In each invocation of build-heap, one key comparison is done in that procedure. No key comparisons are done in leaf-search procedure. The number of key comparisons done in bottom-up-search and interchange procedures is at least $k - 1$ and at most k. Thus, the algorithm uses $k + 1$ comparison in the worst case to build a heap of height k from two heaps of height $k - 1$. Then, we have the following.

$$H(k) = \begin{cases} 2H(k-1) + (k+1), & \text{for } k > 1 \\ 0 & \text{for } k = 1. \end{cases}$$

That is,

$$\begin{aligned} H(k) &= \textstyle\sum_{i=0}^{k-2}(k+1-i)2^i \\ &= (k+1)\textstyle\sum_{i=0}^{k-2} 2^i - \sum_{i=0}^{k-2} i2^i \\ &= (k+1)(2^{k-1}-1) - 2((k-2)2^{k-1} - (k-1)2^{k-2} + 1) \\ &= 2^{k+1} - k - 3. \end{aligned}$$

Selection phase

Let $T(k)$ be the maximum number of key comparisons required in the sorting phase for a complete heap of height k. There are 2^{k-1} elements at height 1, 2^{k-2} elements at height 2, and so on. For an element, if the leaf-search stops at level l, then to insert the element at appropriate position we need exactly l comparisons (see rebuild procedure). When an element of level l is selected, the corresponding leaf-search terminates at level l or $l-1$. For the worst case, we assume that it terminates at level l. Note that for the last element of level l the leaf-search always terminates at level $l-1$. If the leaf-search for the penultimate element of level l terminates at level l, we need not do any key comparison with the last element of level l. Hence, for the last two elements of each level l, the algorithms does at the most $l-1$ comparisons. Finally, it does not require to do any key comparisons for level 1 and 2. Then, we have the following.

$$
\begin{aligned}
T(k) &= \{k(2^{k-1}-2)+2(k-1)\}+\{(k-1)(2^{k-2}-2)+2(k-2)\}+\cdots+\{3(2^{3-1}-2)+2(3-1)\} \\
&= \textstyle\sum_{i=3}^{k}(i(2^{i-1}-2)+2(i-1)) \\
&= \textstyle\sum_{i=3}^{k}(i2^{i-1}-2i+2i-2)) \\
&= \textstyle\sum_{i=3}^{k} i2^{i-1} - \sum_{i=3}^{k} 2 \\
&= \textstyle\sum_{i=1}^{k} i2^{i-1} - 2.2^{1} - 1 - \sum_{i=3}^{k} 2 \\
&= k2^{k+1}-(k+1)2^{k}+1-5-2(k-2) \\
&= k2^{k}-2^{k}-2k.
\end{aligned}
$$

Hence, the total number of key comparisons required in the worst case to sort $n = 2^k - 1$ elements is

$$
\begin{aligned}
H(k)+T(k) &= 2^{k+1}-k-3+k2^{k}-2^{k}-2k \\
&= k2^{k}+2^{k}-3k-3 \\
&= (n+1)\log(n+1)+n-3\log(n+1)-2 \\
&= n\log(n+1)+n-2\log(n+1)-2.
\end{aligned}
$$

5 Conclusion

An efficient algorithm for a variant of heapsort has been presented. The algorithm uses only $\lfloor n/2 \rfloor$ additional bits, and it could be considered as almost internal sorting algorithm. This algorithm is better than quicksort. It is to be noted that in a heap the first two elements, the root and its left child, are greatest elements in the heap. So, they can be output in one pass. Then, instead of taking one leaf element at a time in the selection phase, we could take two leaf elements (of known order) to rebuilt a new heap with two fewer elements. This might further reduce the number of key comparisons. Finally, we would conjecture that n elements could be sorted using some variant of (almost internal) heapsort within $n \log n$ key comparisons.

References

[1] B. Bollobas and I. Simon, 'Repeated random insertion into a priority queue', J. of Algorithms, Vol.6(4), 1985, 466–477.

[2] S. Carlson, 'A variant of heapsort with almost optimal number of comparisons', Information Processing Letters, Vol.24(4), 1987, 247–250.

[3] R. Floyd, 'Algorithm 245: Treesort', Communications of the ACM, Vol.7(12), 1964, 701.

[4] G. Gonnet, 'Handbook of algorithms and data structures', Addison-Wesley, Reading, MA, 1984.

[5] G. Gonnet and J. Munro, 'Heap on heaps', SIAM J. of Computing, Vol.15(4), 1986, 964–971. (Also in Proc. of 9th ICALP, 1982, 282–291.)

[6] C. Hoare, 'Quicksort', Computer Journal, Vol.5, 1962, 10–15.

[7] D. Knuth, 'The art of computer programming, Vol.III, sorting and searching', Addison-Wesley, Reading, MA, 1973.

[8] C. McDiarmid and B. Reed, 'Building heaps fast', J. of Algorithms, Vol.10(3), 1989, 352–365.

[9] I. Wegener, 'Bottom-up-heap sort, a new variant of heap sort beating on average quick sort (if n is not very small)', Proc. of Mathematical Foundation of Computer Science, 1990, LNCS 452, 516–522.

[10] I. Wegener, 'The worst case complexity of McDiarmid and Reed's variant of bottom-up-heap sort in less than $n \log n + 1.1n$', Proc. of the Symp. on Theoretical Aspects of Computer Science, 1991, LNCS 480, 137–147.

[11] J. Williams, 'Algorithm 232: Heapsort', Communications of the ACM, Vol.7, 1964, 347–348.

```
Procedure rearrange($i, m$);
  if $i > \lfloor m/2 \rfloor$ then return;
  if $i = \lfloor m/2 \rfloor$ then $min := \min\{a[i], a[2i]\}$;
  if $i < \lfloor m/2 \rfloor$ then $min := \min\{a[i], a[2i], a[2i+1]\}$;
  if $min = a[i]$ then return {already in heap form}
  elseif $min = a[2i]$ then
      exchange $a[i]$ and $a[2i]$;
      rearrange($2i, m$)
  else
      exchange $a[i]$ and $a[2i+1]$;
      rearrange($2i+1, m$)
end-of-rearrange;

Procedure heapsort($a[1..n]$);
  for $i := \lfloor n/2 \rfloor, ..., 1$ do rearrange($i, n$); {heap creation phase}
  for $m := n, ..., 2$ do                          {sorting phase}
      exchange $a[1]$ and $a[m]$;
      if $m \neq 2$ then rearrange($1, m-1$);
end-of-heapsort;
```

Figure 1: The classical Floyd heapsort.

```
Procedure leaf-search($m, i, j$); {search for the special leaf}
  $j := i$;
  while $2j < m$ do
     if $a[2j] \geq a[2j+1]$ then $j := 2j$ else $j = 2j + 1$; {take greater child}
  if $2j = m$ then $j := m$;
end-of-leaf-search; {$j$ is a special leaf}

Procedure bottom-up-search($i, j$);
  while $i < j$ and $a[i] \geq a[j]$ do $j := \lfloor j/2 \rfloor$;
end-of-bottom-up–search;

Procedure interchange($i, j$); {can be efficiently implemented as done in [9]}
  if $i = j$ then return;
  $temp := a[j]$; $a[j] := a[i]$;
  while $j > i$ do
     exchange $temp$ and $a[\lfloor j/2 \rfloor]$
     $j := \lfloor j/2 \rfloor$;
end-of-interchange;

Procedure bottom-up-rearrange($i, m$);
  leaf-search($m, i, j$);
  bottom-up-search($i, j$);
  interchange($i, j$);
end-of-rearrange;

Procedure bottom-up-heapsort($a[1..n]$);
  for $i := \lfloor n/2 \rfloor, ..., 1$ do bottom-up-rearrange($i, n$); {heap creation phase}
  for $m := n, ..., 2$ do                                                {sorting phase}
     exchange $a[1]$ and $a[m]$;
     if $m \neq 2$ then bottom-up-rearrange($1, m-1$);
end-of-heapsort;
```

Figure 2: Bottom-up-heapsort.

```
var
  $info$ : array $[1..2\lfloor(n-1)/2\rfloor]$ of $(l, r, u)$;

Function parent($i$) := $\lfloor i/2 \rfloor$;

Procedure leaf-search($m, i, j$);
  $j := i$;
  while $2j < m$ do
     if $info[j] = l$ then $j := 2j$
     elseif $info[j] = r$ then $j := 2j + 1$
     elseif $a[2j] \geq a[2j+1]$ then $info[j] := l$; $j := 2j$
        else $info[j] := r$; $j := 2j + 1$
  if $2j = m$ then $j := m$;
end-of-leaf-search; {$j$ is a special leaf}

Procedure bottom-up-search($i, j$);
  while $i < j$ and $a[i] \geq a[j]$ do $j := \lfloor j/2 \rfloor$;
end-of-bottom-up-search;

Procedure interchange($i, j$); {can be efficiently implemented as done in [9]}
  if $i = j$ then return;
  $temp := a[j]$; $a[j] := a[i]$;
  while $j > i$ do
     $info[parent(j)] := u$
     exchange $temp$ and $a[parent(j)]$;
     $j := parent(j)$;
end-of-interchange;

Procedure mdr-rearrange($i, m$);
  leaf-search($m, i, j$);
  bottom-up-search($i, j$);
  interchange($i, j$);
end-of-rearrange;

Procedure mdr-heapsort($a[1..n]$);
  for $i := \lfloor(n-1)/2\rfloor, ..., 1$ do $info[i] := u$;
  for $i := \lfloor n/2 \rfloor, ..., 1$ do mdr-rearrange($i, n$); {heap creation phase}
  for $m := n, ..., 2$ do                                          {sorting phase}
     exchange $a[1]$ and $a[m]$;
     if $m \neq 2$ then mdr-rearrange($1, m-1$);
end-of-heapsort;
```

Figure 3: MDR-heapsort.

```
var
  $info$ : array $[1..\lfloor n/2 \rfloor]$ of 0..1;

Function left($i$) := $2i + info[i]$;

Function parent($i$) := $\lfloor i/2 \rfloor$;

Procedure leaf-search($m, i, j$);
  $j := i$;
  while $2j \leq m$ do $j :=$ left($j$);
end-of-leaf-search; {$j$ is a special leaf}

Procedure bottom-up-search($i, j$);
  while $i < j$ and $a[i] \geq a[j]$ do $j := parent(j)$;
end-of-bottom-up-search;

Procedure interchange($i, j$); {can be efficiently implemented as done in [9]}
  if $i = j$ then return;
  $temp := a[j]$; $a[j] := a[i]$;
  while $j > i$ do
    $info[parent(j)] :=$ if $a[2 * parent(j)] \geq a[2 * parent(j) + 1]$ then 0 else 1;
    {if $2 * parent(j) + 1 > m$ then $info[parent(j)] := 0$};
    exchange $temp$ and $a[parent(j)]$;
    $j := parent(j)$;
end-of-interchange;

Procedure rebuild($i, m$);
  leaf-search($m, i, j$);
  bottom-up-search($i, j$);
  interchange($i, j$);
end-of-rebuild;

Procedure build-heap($i, m$);
  if $2i = m$ then $info[i] := 0$;
  if $2i < m$ then $info[i] :=$ if $a[2i] \geq a[2i + 1]$ then 0 else 1;
  leaf-search($m, i, j$);
  bottom-up-search($i, j$);
  interchange($i, j$);
end-of-build-heap;

Procedure proposed-heapsort($a[1..n]$);
  for $i := \lfloor n/2 \rfloor, ..., 1$ do build-heap($i, n$); {heap creation phase}
  for $m := n, ..., 2$ do                                {sorting phase}
    exchange $a[1]$ and $a[m]$;
    $info[\lfloor m/2 \rfloor] := 0$;
    if $m \neq 2$ then rebuild($1, m - 1$);
end-of-heapsort;
```

Figure 4: Proposed algorithm.