Computing 49, 1–9 (1992)

Computing

Best Case Lower Bounds for Heapsort

Y. Ding and **M. A. Weiss,** Miami

Received May 14, 1991

Abstract — Zusammenfassung

Best Case Lower Bounds for Heapsort. The performance of Heapsort algorithms on arbitrary input is examined. It is proved that an $n \log n - O(n)$ lower bound on the number of comparisons holds for a set of Heapsort algorithms, including Williams-Floyd's algorithm, Carlsson's bottom-up linear or binary insertion algorithm, and all up-down algorithms, on any input.

AMS Subject Classifications: 68Q05, 68Q25

Key words: Analysis of algorithms, sorting, comparisons, lower bound, heapsort

Untere Schranken von Heapsort für den besten Fall. Dieser Artikel untersucht die Komplexität von Heapsort Algorithmen für willkürliche Eingaben. Es wird bewiesen, daß für die Anzahl der Vergleiche auf jeden Fall eine untere Schranke vom Typ nlogn-$O(n)$ gilt, und zwar in einr Klasse von Heapsort Algorithmen, die den Williams-Floyd-Algorithmus, den Carlsson-Algorithmus mit linearem oder binärem Einfügen und alle up-down Algorithmen enthält.

1. Introduction

Heapsort is a simple but efficient sorting algorithm that was proposed by Williams [Wi64] and adapted by Floyd [Fl64] originally. The advantages of Heapsort are that it uses no extra space and is guaranteed to have $O(n \log n)$ worst case running time [Kn73]. As one of the approximately optimal sorting algorithms, it has been widely used and studied.

It is easily seen that the original Williams-Floyd Heapsort requires at most $2n \log n + O(n)$ comparisons.[1] (We have proved that this is also a *lower bound.*) To reduce this bound several variations of Heapsort have been proposed and their upper bounds in the worst case as well as in the average case have been published [Ca87a, Ca87b, Ca91, GM86, MR89, We90, XY90]. To our knowledge, however, the only nontrivial worst case lower bound that has been published is in [FSU91] about Carlsson's bottom-up linear-insertion algorithm [Ca87a]. In [DW91] we proved tight worst case lower bounds for a set of other Heapsort algorithms. On

[1] Since we will only discuss the coefficient of the terms of which the order is higher than n (linear), we will express all other terms in the referred results with $O(n)$.

the other hand, the *best case* performance of Heapsort algorithms has not been studied. Although this may be trivial for some other sorting algorithms, our results in this paper will show that it is not the case for Heapsort.

In section 2 a brief introduction of known Heapsort algorithms and their worst case performance is presented. Section 3 shows that $n \log n - O(n)$ comparisons are necessary for William-Floyd's algorithm to sort any n distinct elements; section 4 generalizes this result to other algorithms.

2. Basic Concepts and Worst Case Bounds

A *heap* is a complete binary tree, represented as an array in which each node satisfies the *heap condition*: the value of a node should be larger than (or equal to) the values of its children if it has any. It is easy to see that the root of the heap is the first element of the array, and the parent of an element in position i is in position $\lfloor i/2 \rfloor$, while the (possible) children of the node in position i are in positions $2i$ and $2i + 1$.

The basic idea of Heapsort is to build a heap containing the elements to be sorted, then remove the largest value (at the top) and reconstruct it back to a heap repeatedly until the heap becomes empty. There is standard algorithm to build a heap [Fl64, Ca87a] in $\Theta(n)$ time (for more precise upper and lower bounds see [GM86]); the difference among various Heapsort algorithms is the method used to reconstruct the heap.

There are basically two approaches to reconstruct the heap after the removal of the top element. The original Williams-Floyd Heapsort can be outlined as the following:

1. build a heap of the n elements somehow;
2. repeatedly do the following n times:
   (a) remove and output the element at the top;
   (b) move the last element in the array to the top;
   (c) iteratively adjust the array to restore the heap condition (in the array with 1 less element): if the parent is smaller than its larger child, then swap them.

We will refer to this algorithm as *top-down Heapsort* since the last element is added into the heap from the top. Since each (possible) swapping may require two comparisons, the upper bound for this algorithm is $2n \log n + O(n)$. This is also the worst case lower bound [DW91].

Another approach is to add the last element into the heap from the bottom: the empty top is first filled with its larger child therefore the empty position is moved down one level, and this procedure is repeated until the empty position is on the bottom level, then the last element is put into it and swapped with element(s) of higher level(s) if necessary. This technique has been used in [Ca87a, Ca87b, GM86, XY90] (some refer it as "finding a path of maximum sons").

In the algorithm analyzed by Carlsson [Ca87a], the last element, after being placed into the empty position at the bottom level, is compared with its parent and if the

latter is smaller then they are swapped and this procedure is performed iteratively. This algorithm will be referred to as *bottom-up Heapsort*, although the empty position is not necessarily moved down physically. [Ca87a] shows that the average cost is $n\log n + O(n)$. This algorithm has also been published later by [MR89, We90]; [We90] reports a $\frac{3}{2}n\log n + O(n)$ upper bound for it (the same result can also be obtained as a corollary of our results in this paper). [FSU91] proves a worst case lower bound of $\frac{5}{4}n\log n - O(n\log\log n)$ by constructing a special heap for it. [MR89] uses extra space to store the "knowledge" of the order of elements gained from previous comparisons, obtaining an upper bound of $n\log n + O(n)$; this, however, makes the algorithm not an in-place one. We will not discuss this variation.

Based on the fact that the elements on the path from the top to the empty bottom position are sorted, Carlsson, and Gonnet and Munro suggest that binary search be used in the procedure of determining the final position of the last element, instead of the original linear insertion [Ca87b, GM86]. This leads an upper bound of $n\log n + n\log\log n + O(n)$ [Ca87b]; in [DW91] we proved the same lower bound. We will refer to this algorithm as the *binary insertion Heapsort*. It is, of course, a (conceptually) bottom-up one, since the "path of maximum sons" on which the binary insertion is performed must be determined first.

In [GM86] Gonnet and Munro also indirectly propose another algorithm that has the best known upper bound for Heapsort (if [MR89] is excluded). The idea is: first "move" the empty position down $r = \lceil \log n - \log\log n \rceil$ levels instead of to the bottom, then check whether the last element can be inserted back into these levels; if yes, binary insertion is used, while if no, the algorithm is recursively applied to the subheap with the current empty position as the top. Each reconstruction here requires only $\lceil \log n \rceil + \log^* n \pm O(1)$ comparisons and this is proved to be a tight bound for reconstruction; the upper bound for this algorithm can then be derived out as $n\log n + n\log^* n + O(n)$. Again, we have shown in [DW91] this is also a lower bound. This algorithm will be referred as the *multi-step binary insertion bottom-up Heapsort*. Recently Carlsson [Ca91] improves this method by selecting a slightly different r which makes the single *delete-root* (including heap reconstruction) optimal in the worst case; this will slightly improve the above algorithm, but only in the linear term.

Another class of Heapsort algorithms is originally from Xunrang and Yuzhang [XY90] which combines the ideas of the top-down algorithm with the straight forward bottom-up algorithm by first "moving" down the empty position $r = \frac{2}{3}h$ levels, where h is the height of the heap, and then putting the last element there, which will stay there, move upward, or move downward according to its value as compared with the one of its parent or its children. This algorithm has a $\frac{4}{3}n\log n + O(n)$ worst case upper bound; this again is also a lower bound [DW91].

By allowing binary-insertion instead of linear-insertion when upward insertion is necessary, and with $r = \lceil \log n - \frac{1}{2}\log\log n \rceil$ to balance the cost, in [DW91] we improved this up-down algorithm, obtaining a tight worst case bound of $n\log n + \frac{1}{2}n\log\log n + O(n)$.

We will refer to all algorithms of this type, i.e. the combination of top-down and bottom-up approaches, as *up-down Heapsort*.

3. A Best Case Lower Bound

As we have seen in the last section, *bad* inputs exist that force a high lower bound for Heapsort algorithms. One may then ask whether there are *good* inputs that make an algorithm spend significantly less. Although the answer is trivially "yes" if we count the input of n identical elements, the answer for at least some algorithms, as we will show, is "no" if the elements are all distinct.

Hereinafter we assume all the elements of a heap are distinct.

We start from the classical Williams-Floyd top-down algorithm. Although a direct proof is not hard to give, we can obtain it using the result of [We90] where Lemma 1 in the following comes from.

Lemma 1. *$\frac{3}{2}n\log n + O(n)$ comparisons are sufficient in worst case to sort n elements using Carlsson's bottom-up linear-insertion algorithm.* ■

Since all known Heapsort algorithms generate the same subheap sequence (i.e. after each *delete-root* or *reconstruction* the resulting heap is always the same), we can count the number of comparisons involved in each step of the top-down algorithm by examining the counterpart of the bottom-up algorithm, based on the following fact:

Lemma 2. *For any heap of size n, if $f_{bu}(n)$ comparisons are required for the bottom-up linear-insertion algorithm to reconstruct the heap after a delete-root, and $f_{td}(n)$ comparisons for the top-down algorithm, then* $4 \times \lceil\log n\rceil - 7 \le 2f_{bu}(n) + f_{td}(n) \le 4 \times \lceil\log n\rceil$.

Proof. A heap of size n, after the last leaf is removed, has $\lceil\log n\rceil$ levels. If this last leaf is finally inserted into level k (define the root as level 1), by using the bottom-up linear-insertion algorithm, $(\lceil\log n\rceil - 2)$ or $(\lceil\log n\rceil - 1)$ comparisons are required to establish the *path of the maximum sons*, depending on where this path reaches the bottom; and $[(\lceil\log n\rceil - 1 - k) + 1]$ or $[(\lceil\log n\rceil - k) + 1]$ comparisons are required for locating the insertion position. Thus $2 \times \lceil\log n\rceil - k - 2 \le f_{bu}(n) \le 2 \times \lceil\log n\rceil - k$. By using the top-down algorithm, to get the same result, the required number of comparisons is between $2(k-2)$ and $2k$. Moreover, the two inequalities on the left side cannot become equalities simultaneously. Adding the two inequalities we then have

$$4 \times \lceil\log n\rceil - 7 \le 2f_{bu}(n) + f_{td}(n) \le 4 \times \lceil\log n\rceil.$$ ■

The following result is now an easy one:

Theorem 1. *To sort any n distinct elements, $n\log n - O(n)$ comparisons are necessary for top-down Heapsort.*

Proof. Denote the number of comparisons, involved in the whole sorting process *after* the initial heap construction, by using the top-down algorithm, as $F_{td}(n)$;

similarly define $F_{bu}(n)$ for the bottom-up algorithm. Given any input of size n, from Lemma 2 we have

$$\sum_{k=1}^{n} (4 \times \lceil \log k \rceil - 7) \leq 2 \times \sum_{k=1}^{n} f_{bu}(k) + \sum_{k=1}^{n} f_{td}(k) \leq 4 \times \sum_{k=1}^{n} \lceil \log n \rceil,$$

that is,

$$4 \times \sum_{k=1}^{n} \lceil \log k \rceil - 7n - 2F_{bu}(n) \leq F_{td}(n) \leq 4 \times \sum_{k=1}^{n} \lceil \log k \rceil - 2F_{bu}(n).$$

According to Lemma 1, $F_{bu}(n) \leq \frac{3}{2} n \log n + O(n)$ for any input, therefore

$$\begin{aligned} F_{td} &\geq 4 \times \sum_{k=1}^{n} \lceil \log k \rceil - 7n - 2F_{bu}(n) \\ &\geq 4 \times \sum_{k=1}^{n} (\log k) - 7n - 2 \times \frac{3}{2} n \log n - O(n) \\ &\geq 4\left(n \log n - \frac{3}{2} n \right) - 7n - 3n \log n - O(n) \\ &\geq n \log n - O(n). \end{aligned}$$

Since $\Theta(n)$ comparisons are required to build the initial heap, the total number of comparisons involved in sorting any n distinct elements is therefore at least $n \log n - O(n)$. ■

Remark. The relationship established by Lemma 2 can be used in the analysis of both algorithms. It explains why a "worst case input" for the top-down algorithm, as we constructed in [DW91], is a "best case input" for the bottom-up linear-insertion algorithm. For the same reason, the "bad heap" in [FSU91] is an example of a "good heap" for the top-down algorithm—only $\frac{3}{2} n \log n + O(n \log \log n)$ comparisons are required. If a "better" heap for the top-down algorithm can be constructed, a higher lower bound for the bottom-up algorithm can then be established. Also, if Theorem 1 is proved directly (as we will briefly do in next section), Lemma 1 can then be obtained as a corollary.

4. More Best Case Lower Bounds

It is trivial to see that the $n \log n + O(n)$ best case lower bound holds for bottom-up linear/binary-insertion algorithms:

Theorem 2. *To sort any n distinct elements, $n \log n - O(n)$ comparisons are necessary for bottom-up linear-insertion Heapsort as well as bottom-up binary-insertion Heapsort.*

Proof. Given a heap of size n, to establish a *path of maximum sons* in the heap (excluding the last leaf) requires at least $\lceil \log n \rceil - 2$ comparisons. Therefore the total cost for the whole sorting process is at least

$$\sum_{k=3}^{n} [\lceil \log n \rceil - 2] \geq n \log n - \frac{3}{2}n - 2n = n \log n - \frac{7}{2}n,$$

which is of course a lower bound for the whole algorithm. ■

Remark. Although we have counted only the cost for establishing the path of maximum sons, the total cost for other operations may be only $O(n)$; this is at least true for the linear-insertion algorithm.

The above argument does not work on the multi-step binary-insertion algorithm on which only an $n \log n - n \log \log n - O(n)$ bound can be obtained.

As for up-down algorithms, although this bound is not that trivial, the proof is not hard. We assume the turn point (or the value of r) of the up-down algorithm is determined by some function $\mathscr{H}(h)$ of the number of levels h instead of n ($0 \leq \mathscr{H}(h) < h$)[2]; i.e. to ensure that the subheaps with same number of levels will have a turn point at the same level. In practice this assumption is reasonable. In the case $\mathscr{H}(h) \equiv 0$, it is a top-down algorithm; if $\mathscr{H}(h) \equiv h - 1$, it is a bottom-up algorithm.

Under this assumption, we have

Theorem 3. *To sort any n distinct elements, $n \log n - O(n)$ comparisons are necessary for any up-down algorithm no matter what strategy is used for the up actions.*

Before the proof we need the following lemma:

Lemma 3. *In a heap of n distinct elements, there are at least $\lceil n/4 \rceil$ leaves which are among the $\lfloor n/2 \rfloor$ smallest elements.*

Proof. Denote the set of the $\lfloor n/2 \rfloor$ smallest elements as S. By removing all elements that are not in S we get a forest of k small heaps ($k \geq 1$). It is easy to see (1) all the leaves of these heaps are also leaves of the original heap; and (2) for any heap the number of internal nodes is at least the same as the number of leaves, while the equality holds only when the heap has an even number of elements. Therefore among the elements in S, there are at least $(k - 1)$ more leaves than internal nodes if n is even, and at least k more if n is odd. If the two numbers are not equal, the number of leaves will be at least $\lceil (\lfloor n/2 \rfloor + 1)/2 \rceil \geq \lceil n/4 \rceil$. The two numbers can be equal only when n is even and $k = 1$; but this implies that the only reduced heap itself also contains an even number of elements, i.e. n is divisible by 4, thus there are still $n/4 = \lceil n/4 \rceil$ leaves in S. ■

Proof of Theorem 3. To simplify the proof, we assume $n = 2^h - 1$. In this case, during the first $\lceil n/2 \rceil = 2^{h-1}$ iterations of the sorting phase, $r = \mathscr{H}(h)$ does not change its value. We count the number of comparisons involved in this process.

First assume $r = 0$; that is, the case of the first half of the top-down algorithm. The above lemma says initially there are $\lceil n/4 \rceil$ leaves from the set of smallest elements S; after this process all elements in S are still in the heap, but those leaves (hereinafter called *ex-leaves*) have all changed their positions.

---

[2] Actually $h = \lceil \log n \rceil$ if the last leaf is included in n.

If in one iteration, the *last leaf* is finally inserted at level k (define root as level 1) which is not the bottom level, then it must have been compared with $k-1$ pairs of other elements that are larger, and a pair (or a single, if it is a single child leaf) that are smaller; we can record this by charging the *last leaf* $2k$ (or $2k-1$) points respectively.

If it is finally posited at a leaf position, the charge will be $2(k-1)$ or $2(k-1)-1$, since no element is smaller than the *last leaf* on its way down. If this is an old leaf position, then this element will be moved again later when it becomes the *last leaf* again; the element that was originally here and that has been moved one level up since it is larger than the *last leaf*, however, will not become the *last leaf* during the first $\lceil n/2 \rceil = 2^{h-1}$ iterations. Thus we can record the charge on this element, and it can be regarded as $2k$ or $2k-1$ since the charged element is one level higher than the actual one.

All ex-leaves must have been charged after this process since they have all been moved out of the bottom which has ceased existence. For an ex-leaf at level k, the charge is at least $2k$ if level k does not contain any leaves of the remaining heap, and $2k-3$ if it does. Thus the total charge will be minimized if we assume that all charges are of its minimum value and the ex-leaves are at levels as high as possible. This will give a total charge, denoted as $\mathscr{S}_0$, of

$$\mathscr{S}_0 \geq \left\{ \sum_{k=1}^{m} [2k \times 2^{k-1}] \right\} + (\lceil n/4 \rceil - 2^m + 1) \times (2m-1),$$

where $m = \left\lfloor \log \frac{n}{4} \right\rfloor$ is the number of levels that contain only non-leaf elements in the remaining heap. Summing up the right hand side we get

$$\mathscr{S}_0 \geq \tfrac{1}{2} n \log n - O(n),$$

which is a lower bound for the number of comparisons during this process[3].

If $r > 0$, when r is very large (say, $r = h - C$ for some constant C) the proof is trivial. If this is not the case, the heap can be divided into three segments—the *upper levels* which are higher than level $r+1$, the level $r+1$, called the *middle level*, and the *lower levels* which are lower than the middle level. Also, we call the upward insertion part of the algorithm the *up* part, and the downward insertion using the top-down algorithm the *down* part.

It is not hard to see the following facts:

- an ex-leaf, once moved into the upper levels, can never get back to the lower levels;
- when the *down* part of the algorithm is in use, the actions of the ex-leaves are the same as they are in the top-down algorithm, except that the starting level is not at the root, but at level r;

---

[3] This is in fact part of a direct proof of Theorem 1; by summing up the cost for each phase the lower bound can be derived out. [We90] uses a similar argument to prove Lemma 1.

- an element at the middle level may be moved up or down; but if it was an ex-leaf, it must have been involved in two comparisons in order to be posited at this level[4].

Therefore we can still charge the ex-leaves that are involved in a *down* part of the algorithm as in the case $r = 0$. After the first $\lceil n/2 \rceil = 2^{h-1}$ iterations, the charge for each ex-leaf at level k which is among the lower levels, should be at least $m - 2r$ if the charge in the case $r = 0$ is m.

There may be ex-leaves that once were at the lower levels but then moved into the upper levels; they may also be charged once, but we just ignore that.

For the ex-leaves at the middle level, since they must have been involved in at least two comparisons that are not charged elsewhere, we can charge them this amount.

Now we may estimate the total charge to all the ex-leaves in the middle and lower levels. To minimize it as we did before, we assume as many ex-leaves as possible are in the upper levels for which the charges are ignored; as for those in the middle or lower levels, we assume they are as high as possible to minimize the charge. A conservative way is to subtract $2r$ for each of the $\lceil n/4 \rceil = 2^{h-2}$ ex-leaves from the total charge $\mathscr{S}_0$ for the case $r = 0$ (this, if there is any ex-leaf at an upper level, will over estimate the loss of charge); under this estimation the loss of charge, as compared with $\mathscr{S}_0$, is at most $2r \times 2^{h-2}$. On the other hand, the extra total cost for establishing a partial *path of maximum sons* is $2^{h-1} \times r$; therefore the total number of comparisons we can count has remained unchanged in this case. Denoting the number of comparisons involved in the sorting phase of an up-down algorithm as $\mathscr{C}(n)$, then for $n = 2^h - 1$,

$$\begin{aligned} \mathscr{C}(n) &\geq \mathscr{C}(\lfloor n/2 \rfloor) + \tfrac{1}{2} n \log n - O(n) \\ &= n \log n - O(n). \end{aligned}$$

If $n \neq 2^h - 1$, r may change its value once during this period. Since the argument for the case of $r = 0$ will not be affected, and actually the charges can reflect the number of comparisons already involved even if it is evaluated after any iteration, we can perform the above counting in two steps, each for one value of r, and compare with the corresponding steps in the case $r = 0$. This will result in an asymptotically same result as above. We omit the detailed discussion here. ■

5. Summary

In this paper we establish a *best case lower bound* for a group of Heapsort algorithms. We have in fact proved that even partial use of the top-down algorithm will cause an $n \log n - O(n)$ lower bound on any input. On the other hand, heavy use of the bottom-up algorithm, which asks for a *path of maximum sons*, forces a same or similar bound. Since Gonnet-Munro's multi-step algorithm neither generates

---

[4] One comparison is used to decide to use the *up* part of the algorithm, and the other is used to make it stay at the middle level.

a complete path of maximum sons in all cases, nor uses the top-down insertion method, the proof in section 4 cannot cover it. However, we don't believe the trivial $n \log n - n \log \log n - O(n)$ bound is tight, since this algorithm is definitely unable to work in its best case all the time according to the analysis in the proof of Theorem 3.

Based on this paper, to some extent we can say that Heapsort performs uniformly on all input. This is a property many other types of sorting algorithms do not have.

References

[Ca87a] Carlsson, S.: Average-case results on heapsort. BIT *27*, 2 (1987).

[Ca87b] Carlsson, S.: A variant of heapsort with almost optimal number of comparisons. Information Processing Letters *24*, 4 (1987).

[Ca91] Carlsson, S.: An optimal algorithm for deleting the root of a heap. Information Processing Letters *37*, 2 (1991).

[Do84] Doberkat, E.: An average case analysis of Floyd's algorithm to construct heaps. Information and Control *61*, 2 (1984).

[DW91] Ding, Y., Weiss, M.: On heapsort algorithms that are good in both worst-case and average-case, Tech. Report, School of Computer Science, Florida International University, Feb. 1991.

[Fl64] Floyd, R.: Algorithm 245: Treesort. Comm. ACM *7*, 12 (1964).

[Fr88] Frieze, A.: On the random construction of heaps. Information Processing Letters *27* 2, (1988).

[FSU91] Fleischer, R., Sinha, B., Uhrig, C.: A lower bound for the worst case of bottom-up-heapsort, personal communication, Jan. 1991.

[GM86] Gonnet, G., Munro, J.: Heaps on heaps. SIAM J. Comput. *15*, 4 (1986).

[Ha91] Hayward, R.: Average case analysis of heap building by repeated insertion. J. Algorithms *12*, 1 (1991).

[Kn73] Knuth, D.: The art of computer programming, Vol. 3. Reading, MA: Addison-Wesley 1973.

[MR89] McDiarmid, C., Reed, B.: Building heaps fast. J. Algorithms *10*, 3 (1989).

[PS75] Porter, T., Simon, I.: Random insertion into a priority queue structure. IEEE Trans. Software Engineering *SE-1*, 3 (1975).

[We90] Wegener, I.: Bottom-up-heapsort, a new variant of heapsort beating on avergage quicksort (if n is not very small). Berlin, Heidelberg, New York: Springer 1990 (MFCS'90, Lecture Notes in Computer Science 452).

[Wi64] Williams, J.: Algorithm 232: Heapsort. Comm. ACM *7*, 6 (1964).

[XY90] Xunrang, G., Yuzhang, Z.: A new heapsort algorithm and the analysis of its complexity. The Computer Journal *33*, 3 (1990).

Mr. Yuzheng Ding
Computer Science Department
University of California, Los Angeles
405 Hilgard Avenue
Los Angeles, CA 90024
U.S.A.

Email: eugene@cs.ucla.edu

Dr. Mark Allen Weiss
School of Computer Science
Florida International University
University Park
Miami, Fl 33199
U.S.A.

Email: weiss@scs.fiu.edu