The Ultimate Heapsort

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
Electronic mail: `jyrki@diku.dk`

Abstract. A variant of Heapsort—named Ultimate Heapsort—is presented that sorts n elements in-place in $\Theta(n \log_2(n+1))$ worst-case time by performing at most $n \log_2 n + \Theta(n)$ key comparisons and $n \log_2 n + \Theta(n)$ element moves. The secret behind Ultimate Heapsort is that it occasionally transforms the heap it operates with to a two-layer heap which keeps small elements at the leaves. Basically Ultimate Heapsort is like Bottom-Up Heapsort, but due to the two-layer heap property an element taken from a leaf has to be moved towards the root only $O(1)$ levels, on an average.

Let $a[1..n]$ be an array of n elements each consisting 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 $a[\lfloor i/2 \rfloor]$ is larger than or equal to that of element $a[i]$. That is, a heap is a pointer-free representation of a left complete 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 $a[\lfloor i/2 \rfloor]$, $a[2i]$ 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.

Heapsort is a classical sorting method that is described in almost all algorithmic textbooks. 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 $a[1..n]$ rearranged in sorted order, i.e., the key of $a[i]$
   should be smaller than or equal to that of $a[i+1]$, for all $i \in \{1, 2, \ldots, n-1\}$.
1. Rearrange $a[1..n]$ into a heap.
2. for $i \leftarrow n$ step -1 until 2 do
      Exchange $a[1]$ and $a[i]$.
      Remake $a[1..i-1]$ into a heap.
```

According to Williams [28], who invented Heapsort and the heap data structure, Heapsort is a marriage of Treesort developed by Floyd [14] and Tournament-sort described, for instance, by Iverson [17, Section 6.4] (see also [15]). Both Treesort and Tournament-sort run in $\Theta(n \log_2(n+1))$ worst-case time and carry out $n \log_2 n + \Theta(n)$ key comparisons and $n \log_2 n + \Theta(n)$ element moves. For example, Treesort uses $2n + \Theta(1)$ pointers and $2n$ additional locations for elements, whereas Heapsort requires only $\Theta(1)$ extra space for its operation. In the form programmed by Williams [28], Heapsort sorts an array of size n in $\Theta(n \log_2(n+1))$ time by performing about

$3n \log_2 n$ key comparisons and $4\frac{3}{4}n \log_2 n$ element moves (cf. [23]) in the worst case. Soon after the publication of Heapsort, Floyd [16] improved it by showing that (1) the construction of a heap for n elements can be accomplished in-place in $\Theta(n)$ time by performing at most $2n$ key comparisons and $2n$ element moves, and (2) n elements can be sorted in-place in $\Theta(n \log_2(n+1))$ time by performing at most $2n \log_2 n + \Theta(n)$ key comparisons and $n \log_2 n + \Theta(n)$ element moves.

Floyd's results have been improved further. Now there exists an in-place algorithm that constructs a heap of size n in linear time by performing only $1.6316n + O(\log^2 n)$ key comparisons in the worst case [8] (see also [5]). Moreover, the number of element moves can be easily reduced to, say $1.625n$. As to Heapsort, the number of key comparisons performed has been gradually reduced to $\frac{3}{\log_2 3} n \log_2 n + \Theta(n)$ [21, Exercise 5.2.3-28], $\frac{3}{2}n \log_2 n + \Theta(n)$ [2, 11, 23, 25], $\frac{4}{3}n \log_2 n + \Theta(n)$ [29, 30], $\frac{7}{6}n \log_2 n + \Theta(n)$ [26], $n \log_2 n + n \log_2 \log_2 n + \Theta(n)$ [3, 13], and $n \log_2 n + n \log_2^* n + \Theta(n)$ [4, 12]. Here $\log_2^* n$ is defined as $1 + \log_2^*(\log_2 n)$, if $n > 2$, and otherwise it is 1. It is even possible to reduce the number of key comparisons to $n \log_2 n + \Theta(n)$—without increasing the running time or the number of element moves—if extra storage for n bits is available [6, 10, 24] or if this behaviour is only called for in the average case [23, 25].

In this paper a new variant of Heapsort—named Ultimate Heapsort—is presented that guarantees $\Theta(n \log_2(n+1))$ worst-case time behaviour, uses $\Theta(1)$ extra space, and performs at most $n \log_2 n + \Theta(n)$ key comparisons and $n \log_2 n + \Theta(n)$ element moves. This resolves a long-standing open problem but it must be conceded that the solution is mainly of theoretical interest.

The present investigation was triggered off by the recent analyses of the best case of Heapsort [2, 23] and the worst case of Bottom-Up Heapsort [11, 23, 25]. These analyses show clearly that the normal heap property is not fully satisfactory, since it still permits Bottom-Up Heapsort to carry out as many as $\frac{3}{2}n \log_2 n - O(n \log_2 \log_2 n)$ key comparisons in the worst case [11, 23]. On an average Bottom-Up Heapsort performs only $n \log_2 n + \Theta(n)$ key comparisons [23, 25]. Even if there exists some pathological inputs that cannot be handled efficiently by Bottom-Up Heapsort, these are rare. Therefore, I started to search for a stronger heap property that would make these pathological cases impossible. The two-layer heap concept to be defined next turns out to have the characteristics I was looking for.

Let $a[1..n]$ be an array of size n. Further, let 2^d be the largest power of 2 for which $2^d \leq n$. A *two-layer heap* stored in $a[1..n]$ is a heap with the additional property that, for all $j \in \{1, 2, \ldots, 2^d - 1\}$ and $k \in \{2^d, 2^d + 1, \ldots, n\}$, the key of element $a[j]$ is larger than or equal to that of element $a[k]$. Fig. 1 gives an example of a two-layer heap storing integers.

1	2	3	4	5	6	7	8	9	10	11	12
60	38	18	31	12	10	16	6	10	9	2	5

Fig. 1. A two-layer heap of size 12.

The elements in an array can be rearranged to form a two-layer heap as follows:

```
   Input: Array $a[1..n]$ of $n$ elements.
   Output: The elements in $a[1..n]$ rearranged into a two-layer heap.
1. Find element $x$ that has the $2^d$th largest key of the elements in $a[1..n]$, where $2^d$
   is the largest power of 2 smaller than or equal to $n$. This can be done, e.g., by
   using the in-place selection algorithm of Lai and Wood [22].
2. Partition $a[1..n]$ into three pieces $a[1..\ell]$, $a[\ell+1..\ell+e]$, and $a[\ell+e+1..n]$ such
   that the key of every element in $a[1..\ell]$, $a[\ell+1..\ell+e]$, and $a[\ell+e+1..n]$ is
   larger than, equal to, and smaller than that of $x$, respectively. Here the 3-way
   partitioning algorithm described, e.g., by Dijkstra [9, Chapter 14] or Wegner [27]
   could be used.
3. Rearrange $a[1..\ell]$ into a heap, e.g., by using the standard heap-construction al-
   gorithm [16].
```

Lemma 1. *A two-layer heap for n elements can be constructed in-place in $O(n)$ time by performing fewer than $9n+\ell$ key comparisons and $22n$ element moves, where ℓ is the number of elements with a key larger than that of the partitioning element x.*

Proof. That the running time of the algorithm described is linear is obvious since all the subroutines run in linear time. The selection of the partitioning element x requires fewer than $7n$ key comparisons and $19n$ element moves [22]. It is not difficult to organize the partitioning such that every element with a key larger than that of x causes one comparison and one move, while every other element causes two comparisons and three moves. Thus the partitioning requires $2n-\ell$ comparisons and $3n-2\ell+1$ moves in total. The construction of the heap requires at most 2ℓ comparisons and 2ℓ moves [16]. Hence, the total number of key comparisons is bounded by $9n+\ell$ and that of element moves by $22n$. ■

The reader is encouraged to device a method for constructing a two-layer heap that improves the constant factors given in Lemma 1. Especially, a better in-place method for selecting the kth largest element of a *multiset* is needed that performs the partitioning at the same time.

We need also a subroutine that picks up those elements from a heap that have a key equal to the key of the element stored currently at the root. The elements picked up are to be moved at the end of the array storing the heap. It is easy to write a recursive procedure that traverses the heap in depth-first order and carries out the movements required. The recursion can be removed from this procedure without using any space for a recursion stack. The details are given below:

```
   Input: A heap of $n$ elements stored in $a[1..n]$.
   Output: The elements in $a[1..n]$ rearranged such that, if the key of $a[j]$ is smaller
   than that of $a[n]$, then the same holds for all $i \in \{1,2,\ldots,j\}$.
1. $(i,j) \leftarrow (0,1)$; $x \leftarrow a[1]$; $y \leftarrow a[n]$; $h \leftarrow n$;
2. **while** $(i,j) \neq (1,0)$ **do**
      **if** $j \geq h$ **then** $(i,j) \leftarrow (j,i)$
```

```
    else if $i = \lfloor j/2 \rfloor$ then
        if the key of $a[j]$ is equal to that of $x$ then $(i, j) \leftarrow (j, 2j)$
        else $(i, j) \leftarrow (j, i)$;
    else if $i = 2j$ then $(i, j) \leftarrow (j, 2j + 1)$
    else if $i = 2j + 1$ then
        repeat $a[h] \leftarrow a[j]$; $h \leftarrow h - 1$; $a[j] \leftarrow a[h]$
        until the key of $a[j]$ is not equal to that of $x$ or $j = h$;
        $(i, j) \leftarrow (j, \lfloor j/2 \rfloor)$;
3. $a[h] \leftarrow y$.
```

Lemma 2. *Let x be the root of a heap stored in $a[1..n]$. Further, let m denote the number of those elements the key of which is equal to that of x. These m elements can be gathered to the end of array $a[1..n]$ in-place in $\Theta(m)$ time by performing at most $3m + 1$ key comparisons and $2m + 2$ element moves.*

Proof. The procedure described above visits at most $2m + 1$ nodes at the top of the heap, i.e., the nodes that contain an element with a key equal to x and the children of those nodes. When a node is visited for the first time, one element comparison is carried out. Moreover, when an element is moved to the end of the array, one extra comparison is necessary to check whether the new element also has a key equal to that of x. In total, at most $3m + 1$ comparisons are performed. All the m elements have to be moved to the hole maintained and a new hole have to be dug beside the old one. The two extra moves are needed for digging the initial hole at the end of the array and for filling the final hole. ■

Now we are ready for describing Ultimate Heapsort. Assume that the input array to be sorted is of size n. The given elements are sorted in $d - 1$ *rounds*, where $d = \lceil \log_2(n + 1) \rceil$. In each round about one half of the remaining elements is sorted. Let n_i denote the number of remaining elements prior to round i, $i \in \{1, 2, \ldots, d-1\}$. Of course, $n_1 = n$. Moreover, assume that 2^{d_i} is the largest power of 2 that is smaller than or equal to n_i.

In the *first phase* of the ith round, $i \in \{1, 2, \ldots, d - 1\}$, the remaining n_i elements are built into a two-layer heap. Let x be the element with the 2^{d_i}th largest key in this heap. Further, let ℓ_i be the number of those elements in the heap that have a key larger than that of x. In the *second phase* of the ith round, the ℓ_i elements with a key larger than that of x are removed from the heap. The removed elements are exchanged with the last ℓ_i elements of the heap as normally. In the *third phase* of the ith round, $n_i - 2^{d_i} - \ell_i$ of the elements with a key equal to that of x are gathered together and moved at the end of the sub-array containing the remaining elements. Prior to the next round, n_{i+1} is updated to $2^{d_i} - 1$. The computations carried out in one round are summarized below:

Input: Array $a[1..n]$ and element x whose key is the 2^dth largest of the elements in $a[1..n]$. Here 2^d is the largest power of 2 for which $2^d \leq n$. Let ℓ denote the number of elements with a key larger than that of x.
Output: The elements in $a[1..n]$ rearranged so that the key of $a[j]$ is smaller than or equal to that of x, for all $j \in \{1, 2, \ldots, 2^d - 1\}$, and $a[2^d..n]$ is in sorted order.

```
1. Rearrange $a[1..n]$ into a two-layer heap.
2. for $j \leftarrow n$ step $-1$ until $n-\ell$ do
       Exchange $a[1]$ and $a[j]$.
       Remake $a[1..j-1]$ into a heap.
3. Gather $n$-$2^d$-$\ell$ of the elements with a key equal to that of $x$ to the end of $a[1..j]$.
```

To be more specific, the correction of the heap in the second phase is organized as in Bottom-Up Heapsort. First, a copy of the last leaf is taken; let z be this copy. Second, a *hole* is created at the root by removing the current maximum and moving it at the place occupied earlier by the last leaf. Third, the hole is sifted down along the path of maximum children until the leaf level is reached. Fourth, the hole is sifted up along the same path until an element is found which is *larger than* or *equal to* z. Finally, z is moved into this hole after which the next iteration can be started.

Lemma 3. *Let n be the size of a two-layer heap. Further, let $d = \lceil \log_2(n+1) \rceil$ and let ℓ be the number of the elements that have a key larger than that of the partitioning element x. These ℓ elements can be removed from the heap in reversed sorted order in $\Theta(\ell \log_2 n + n)$ time and during the removal process at most $\ell(d-1)+2^d$ key comparisons and $\ell(d+1)+2^d$ element moves are carried out.*

Proof. We shall analyse the number of key comparisons carried out during the removals. The number of element moves performed as well as the actual running time are closely related to this quantity, so the verification of that part of the lemma is left for the reader.

Due to the two-layer property of the heap an element at a leaf on its bottommost level has a key that is smaller than or equal to that of an element stored in any of the internal nodes. Therefore, it is very probable that during a sift-up only a few elements are visited. However, the heap is changing all the time so there are some elements that can traverse close to the root. The formal proof about the number of comparisons carried out should take this into account.

Consider now the situation prior to any removals. We say that the elements whose key is larger than that of the partitioning element x are *red*. All the other elements are called *white*. That is, in the second phase all red elements are removed from the heap, whereas the white elements stay in the heap. Assume now that there are some red elements, since otherwise nothing is done in the second phase. It is easy to see that the algorithm maintains the following invariant: on any path from the root to a leaf there appears some red elements followed by some white elements. In particular, an element at the bottommost level can never be red.

Let n, d, and ℓ be as stated in the lemma. We define the *height* of an element to be d minus the distance between the node storing it and the root. Assume now that each comparison costs 1 krone. To pay the climbs down and up the heap, we give $d-1$ kroner to each of the ℓ elements at the bottommost level that are to be replaced during the removal process. Moreover, we give h kroner for each red element if its height is h. If a white element is moved into the place of some bottommost element having some money, the just inserted element inherits the money.

The climb down the heap can be paid by the white element taken from the last leaf, since during this operation $d-1$ or $d-2$ comparisons are carried out. When a hole is climbing up, the climb is stopped by a white or a red element. The comparisons performed can be paid by the red element removed from the root. This element had d kroner, so it can also donate 1 krone for each red element on the maximum path above the hole. After getting this donation, a red element that was moved one position towards the root has still h kroner if h is its current height.

In total, we need at most

$$\ell(d-1) + \sum_{j=1}^{d} j \cdot 2^{d-j-1}$$

kroner to pay all the comparisons, which is bounded by $\ell(d-1)+2^d$. This concludes the proof of the lemma. ■

Let us now put the results of the previous lemmas together.

Theorem 4. *Ultimate Heapsort sorts n elements in-place in $\Theta(n\log_2(n+1))$ worst-case time by performing fewer than $n\log_2 n + 32n + O(\log_2 n)$ key comparisons and $n\log_2 n + 62n + O(\log_2 n)$ element moves.*

Proof. Consider first the number of comparisons performed. As earlier, let n_i denote the number of elements left prior to the ith round, 2^{d_i} the rank of the partitioning element, and ℓ_i the number of those elements that have a key larger than that of the partitioning element. Let $d = \lceil \log_2(n+1) \rceil$. According to the lemmas proved, the cost of the ith round, $i \in \{1, 2, \ldots, d-1\}$, is

$$9n_i + \ell_i + \ell_i d_i + 2^{d_i+1} + 3(n_i - 2^{d_i} - \ell_i) + 1.$$

By using the facts that $n_1 = n$, $n_i = 2^{d_i+1}-1$ for $i \geq 2$, $d_i = d - i$, and $d \leq \log_2 n + 1$, we get that the total number of key comparisons performed is bounded by $n\log_2 n + 32n + O(\log_2 n)$.

The actual running time is closely related to the number of key comparisons carried out. Therefore, we conclude that the algorithm runs in $\Theta(n\log_2(n+1))$ worst-case time. The number of element moves can be calculated in the same manner as key comparisons. This quantity turns out be bounded by $n\log_2 n + 62n + O(\log_2 n)$. ■

The goal in this paper has been to show that there exists a variant of Heapsort which sorts n elements in-place in $\Theta(n\log_2(n+1))$ worst-case time by performing at most $n\log_2 n + \Theta(n)$ key comparisons and $n\log_2 n + \Theta(n)$ element moves. Basically, Ultimate Heapsort operates like Bottom-Up Heapsort but it is able to handle all inputs equally efficiently (or should I say inefficiently). This strengthens the conviction that Bottom-Up Heapsort is the way how Heapsort should be implemented. Due to the large constant factor in the linear term of the running time, the current implementation of Ultimate Heapsort will not beat Bottom-Up Heapsort in practice. However, as with the first selection algorithm deviced by Blum et al. [1], I believe that the performance of Ultimate Heapsort can be improved considerably.

Finally, I would like to point out that the performance of in-place Mergesort developed by Katajainen et al. [19] has been recently improved such that it will beat all known variants of Heapsort, both in theory [18] and in practice [20].

Acknowledgements. The author would like to thank Tomi Pasanen and Jukka Teuhola for helpful discussions. This work was partially supported by the Danish Natural Science Research Council under contract No. 9400952 (Project "Computational Algorithmics").

References

1. M. Blum, R. F. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, Time bounds for selection, *Journal of Computer and System Sciences* **7** (1973) 488–461.
2. B. Bollobás, T. I. Fenner, and A. M. Frieze, On the best case of Heapsort, *Journal of Algorithms* **20** (1996) 205–217.
3. S. Carlsson, A variant of HEAPSORT with almost optimal number of comparisons, *Information Processing Letters* **24** (1987) 247–250.
4. S. Carlsson, A note on HEAPSORT, *The Computer Journal* **35** (1992) 410–411.
5. S. Carlsson and J. Chen, Heap construction: Optimal in both worst and average cases?, *Proceedings of the 6th Annual International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science **1004**, Springer, Berlin/Heidelberg (1995) 254–263.
6. S. Carlsson, J. Chen, and C. Mattsson, Heaps with bits, *Theoretical Computer Science* **164** (1996) 1–12.
7. S. Carlsson and M. Sundström, Linear-time in-place selection in less than $3n$ comparisons, *Proceedings of the 6th Annual International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science **1004**, Springer, Berlin/Heidelberg (1995) 244–253.
8. J. Chen, A framework for constructing heap-like structures in-place, *Proceedings of the 4th International Symposium on Algorithms and Computation*, Lecture Notes in Computer Science **762**, Springer-Verlag, Berlin/Heidelberg (1993) 118–127.
9. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs (1976).
10. R. D. Dutton, Weak-heap sort, *BIT* **33** (1993) 372–381.
11. R. Fleischer, A tight lower bound for the worst case of Bottom-Up-Heapsort, *Proceedings of the 2nd International Symposium on Algorithms*, Lecture Notes in Computer Science **557**, Springer-Verlag, Berlin/Heidelberg (1991) 251–262.
12. G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM Journal on Computing* **15** (1986) 964–971.
13. X. Gu and Y. Zhu, Optimal heapsort algorithm, *Theoretical Computer Science* **163** (1996) 239–243.
14. R. W. Floyd, Algorithm 113: Treesort, *Communications of the ACM* **5** (1962) 434.
15. E. H. Friend, Sorting on electronic computer systems, *Journal of the ACM* **3** (1956) 134–168.
16. R. W. Floyd, Algorithm 245: Treesort 3, *Communications of the ACM* **7** (1964) 701.
17. K. E. Iverson, *A Programming Language*, John Wiley and Sons, New York (1962).
18. J. Katajainen and T. Pasanen, In-place sorting with fewer moves, in preparation.
19. J. Katajainen, T. Pasanen, and J. Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996) 27–40.

20. J. Katajainen and J. L. Träff, A meticulous analysis of mergesort programs, *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, Lecture Notes in Computer Science **1203**, Springer, 1997.
21. D. E. Knuth, *The Art of Computer Programming*, Volume 3/ *Sorting and Searching*, Addison-Wesley Publishing Company, Reading (1973).
22. T. W. Lai and D. Wood, *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **318**, Springer-Verlag, Berlin/Heidelberg (1988) 14–23.
23. R. Schaffer and R. Sedgewick, The analysis of Heapsort, *Journal of Algorithms* **15** (1993) 76–100.
24. I. Wegener, The worst case complexity of McDiarmid and Reed's variant of BOTTOM-UP HEAPSORT is less than $n \log n + 1.1n$, *Information and Computation* (1992) 86–96.
25. I. Wegener, BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if n is not very small), *Theoretical Computer Science* **118** (1993) 81–98.
26. I. Wegener, A simple modification of Xunrang and Yuzhang's HEAPSORT variant improving its complexity significantly, *The Computer Journal* **36** (1993) 286–288.
27. L. M. Wegner, Quicksort for equal keys, *IEEE Transactions on Computers* **C-34** (1985) 362–367.
28. J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7** (1964) 347–348.
29. G. Xunrang and Z. Yuzhang, A new HEAPSORT algorithm and the analysis of its complexity, *The Computer Journal* **33** (1990) 281–282.
30. G. Xunrang and Z. Yuzhang, Asymptotic optimal HEAPSORT algorithm, *Theoretical Computer Science* **134** (1994) 559–565.