

A New Data Structure for Heapsort with Improved Number of Comparisons

(Extended Abstract)

Md. Shahjalal¹ and M. Kaykobad²

¹ CSE Department, North South University
shahjalal@msn.com

² CSE Department, Bangladesh University of Engineering and Technology
kaykobad@cse.buet.ac.bd

Abstract. In this paper we present a new data structure for implementing heapsort algorithm for pairs of which can be simultaneously stored and processed in a single register. Since time complexity of Carlsson type variants of heapsort has already achieved a leading coefficient of 1, concretely $n \lg n + n \lg \lg n$, and lower bound theory asserts that no comparison based in-place sorting algorithm can sort n data in less than $\lceil \lg(n!) \rceil \approx n \lg n - 1.44n$ comparisons on the average, any improvement in the number of comparisons can only be achieved in lower terms. Our new data structure results in improvement in the linear term of the time complexity function irrespective of the variant of the heapsort algorithm used. This improvement is important in the context that some of the variants of heapsort algorithm, for example weak heapsort although not in-place, are near optimal and is away from the theoretical bound on number of comparisons by only $1.54n$.

Keywords: heap, heapsort, clock cycles, worst-case analysis

1 Introduction

Heapsort is one of the best sorting algorithms with both average and worst case complexity of $O(n \lg n)$. It is based on heap data structure implemented in a complete binary tree. While leading coefficient of $n \lg n$ in the worst case complexity of traditional heapsort [15] is 2, Carlsson's variant brought the complexity down to $n \lg n + n \lg \lg n$ [2]. Potential of heapsort algorithm attracted attention of many researchers and as a result many promising variants of heapsort like Weak-Heapsort [5], Ultimate Heapsort [10], Bottom-Up-Heapsort [14], MDR-Heapsort [11, 3], Generalized Heapsort [13, 9] as well as heaps like Weak heap [6], Min-max heap [1], Min-max pair heap [12, 4] have been introduced. Although weak heap-sort is not an in-place sorting algorithm, in terms of number of comparisons it is by far the best variant requiring $n \lg n + 0.1n$ comparisons and is only $1.54n$

M. Kaykobad and Md. Saidur Rahman (Eds.): WALCOM 2007, pp. 88–96, 2007.

more than the theoretical bound for number of comparisons [2]. Heapsort algorithm works in two phases. In the first phase, a heap is built that requires about $1.5n$ comparisons for all variants excepting for Weak-Heapsort in which case about n comparisons suffice although it needs additional n bits. In the second phase sorting is performed. Williams' [15] or Floyd's version [7] requires $2n \lg n$ comparisons, whereas Carlsson's and similar variants require $n \lg n + n \lg \lg n$. We introduce a new data structure in which creation of heap will require lesser number of comparisons excepting for Weak-Heaps. When heapsort algorithm is implemented on such data structures all variants of heapsort algorithm require lesser number of comparisons. Simulation runs also support the results of theoretical analysis. In this paper we restrict our discussion and analysis to in-place variants of heapsort. In Section 2, we elaborately describe our data structure, whereas section 3 describes improvement in the heapsort algorithm implemented on the new data structure. Section 4 contains analysis of the number of movements required by Carlsson's [2] variant of heapsort algorithm implemented on this new data structure.

2 New Data Structure

Assuming that any of the $n!$ permutations are equally likely probability that any permutation is sorted is $\frac{1}{n!}$. Independent pairwise comparisons result in reducing uncertainty by the most and probability of a sequence, complying with the comparison, being sorted is doubled after each such independent comparison. This fact encourages creation of heaps with two element ordered in each node

For simplicity of discussion let us sort pair of elements in ascending order. By $n/2$ comparisons (assuming sequence starts from 0) all larger elements of each pair $2i, 2i + 1$ can be brought to even positions. Let us call even elements active and the other ones dormant.

Now a heap is constructed using $n/2$ active elements. For any heap creation algorithm requiring $(1 + \alpha)n$ comparisons with $\alpha > 0$ creation phase will require lesser number of comparisons since $\frac{n}{2} + (1 + \alpha)\frac{n}{2} = (1 + \frac{\alpha}{2})n$ comparisons. In the case of constructing weak heaps which takes as low as $n - 1$ comparisons, the new data structure takes the same number of comparisons. Because for any positive integer n , $\lfloor (n/2) \rfloor + \lfloor ((n + 1)/2) \rfloor - 1 = n - 1$. Weak heap requires n extra bits for sorting but if we deploy the new data structure, it only takes $n/2$ extra bits [5].

But in this data structure during swapping we must swap nodes, containing pair of elements, with similar nodes. Now-a-days 64-bit processors are very common and most of the data for sorting can be accommodated in 32 bits. We use this advantage of element swapping equivalent to swapping of two-element nodes with another two-element node at no extra cost. In fact, this advantage can be taken for all processors and data for which data size is half the size of the processor's register.

3 Improvement in Sorting Phase

In the sorting phase we can apply any algorithm in the following way and as depicted in Figure 1-4 and see that in terms of number of comparisons the new data structure is better. After constructing the heap, in the sorting phase the active element in the root will be placed in the last unsorted position (the 1st or the 2nd key position of the unsorted pair) and perform heapify or adjustment of heap. The new data structure always need an extra comparison before the heapify. This is to find out the larger or smaller key between the 2nd key of the 1st pair and the 1st key of the last pair.

While for traditional data structure, if a heap with n elements has height h , in our data structure it will have height $\approx h - 1$ (actual height is $\lceil \lg(n + 2) \rceil - 2$). The complexity of any heapsort algorithm rely on the height of the tree or heap. Every sorting algorithm searches along the height for finding the larger or smaller element.

For Floyd's [7] and similar type variants which look for larger key in every level, requires 2 comparisons per level. The worst number of comparisons for sorting each element is $2h$, whereas in the proposed data structure it will be around $2(h - 1) + 1 = 2h - 1$ comparisons.

In Carlsson type variants [2], the path of the elder sons is constructed and then binary search is performed to ascertain the position of the last unsorted element. In these variants the number of comparisons for adjusting an element at position $i - 1$ is $\lfloor \lg(i) \rfloor + \lceil \lg(\lfloor \lg(i) \rfloor) \rceil$, the new data structure will need $(\lceil \lg(i + 2) \rceil - 2) + \lceil \lg(\lceil \lg(i + 2) \rceil - 2) \rceil$. If the maximum number of comparisons needed for the Carlsson's heapsort is denoted by $C(n)$, then

$$C(n) = \sum_{i=1}^2 0 + \sum_{i=3}^n (\lfloor \lg(i - 1) \rfloor + \lceil \lg(\lfloor \lg(i - 1) \rfloor) \rceil) = \sum_{i=2}^{n-1} (\lfloor \lg(i) \rfloor + \lceil \lg(\lfloor \lg(i) \rfloor) \rceil)$$

Number of comparisons required by the proposed data structure, denoted by $\tilde{C}(n)$ is then

$$\begin{aligned} \tilde{C}(n) &= \sum_{i=1}^4 0 + \sum_{i=5}^n ((\lceil \lg(i + 1) \rceil - 2) + \lceil \lg(\lceil \lg(i + 1) \rceil - 2) \rceil) + (n - 2) \\ &= \sum_{i=6}^{n+1} ((\lceil \lg(i) \rceil - 2) + \lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) + (n - 2) \end{aligned}$$

Their difference is $C(n) - \tilde{C}(n)$

$$= \sum_{i=2}^{n-1} (\lfloor \lg(i) \rfloor + \lceil \lg(\lfloor \lg(i) \rfloor) \rceil) - \sum_{i=6}^{n+1} ((\lceil \lg(i) \rceil - 2) + \lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) - (n - 2)$$

$$\begin{aligned}
&= \sum_{i=2}^{n-1} (\lfloor \lg(i) \rfloor) - \sum_{i=6}^{n+1} (\lceil \lg(i) \rceil - 2) + \sum_{i=2}^{n-1} \lceil \lg(\lfloor \lg(i) \rfloor) \rceil - \sum_{i=6}^{n+1} (\lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) - (n-2) \\
&= \sum_{i=6}^{n-1} (\lfloor \lg(i) \rfloor) + 6 - \sum_{i=6}^{n-1} (\lceil \lg(i) \rceil) - \lceil \lg(n) \rceil - \lceil \lg(n+1) \rceil + 2(n-4) + \sum_{i=6}^{n-1} \lceil \lg(\lfloor \lg(i) \rfloor) \rceil + 2 \\
&\quad - \sum_{i=6}^{n-1} (\lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) - \lceil \lg(\lceil \lg(n) \rceil - 2) \rceil - \lceil \lg(\lceil \lg(n+1) \rceil - 2) \rceil - (n-2) \\
&= \left[\sum_{i=6}^{n-1} (\lfloor \lg(i) \rfloor) - \sum_{i=6}^{n-1} (\lceil \lg(i) \rceil) \right] + \left[\sum_{i=6}^{n-1} \lceil \lg(\lfloor \lg(i) \rfloor) \rceil - \sum_{i=6}^{n-1} (\lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) \right] \\
&\quad - \lceil \lg(n) \rceil - \lceil \lg(n+1) \rceil - \lceil \lg(\lceil \lg(n) \rceil - 2) \rceil - \lceil \lg(\lceil \lg(n+1) \rceil - 2) \rceil + n + 2
\end{aligned}$$

We know the difference between ceil and floor values of $\lg i$ is always 1, for all values of i expecting for powers of 2. So, from $i = 6$ to $i = n - 1$ there are $(n - 1 - 6 + 1) = (n - 6)$ elements and for $i = 2$ and $i = 4$, we miss 2 elements because we start from $i = 6$. So for the expression inside the first pair of third brackets

$$\sum_{i=6}^{n-1} (\lfloor \lg(i) \rfloor) - \sum_{i=6}^{n-1} (\lceil \lg(i) \rceil) = -(n-6+2) + \lfloor \lg(n-1) \rfloor = -(n-4) + \lfloor \lg(n-1) \rfloor$$

The value of the expression $\lceil \lg(\lfloor \lg(i) \rfloor) \rceil - \lceil \lg(\lceil \lg(i) \rceil - 2) \rceil$ inside the second pair of third brackets is 0 or 1 in alternate intervals as shown in the table below, excepting for $i = 8$ where the value becomes 2. In the second row the value after plus sign indicates total savings at the end of the interval. For example, in interval $[2^5, 2^6]$ there are 33 elements where we save one comparison per element. So total savings is 33 comparisons, whereas in the preceding interval we do not have any savings per element as reflected in 0 after plus sign. In Table 1 values of this expression have been shown for different important intervals of i . So,

$$\sum_{i=6}^{n-1} \lceil \lg(\lfloor \lg(i) \rfloor) \rceil - \sum_{i=6}^{n-1} (\lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) = \sum_{i=17}^{n-1} (\lceil \lg(\lfloor \lg(i) \rfloor) \rceil - \lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) + 12$$

Let $x = \lceil \lg(\lfloor \lg(n-1) \rfloor - 1) \rceil - 2$ and $R = \begin{cases} n - 2^{2^{x+2}+1} + 1, & \text{if } n - 1 \geq 2^{2^{x+2}+1} \\ 0, & \text{otherwise.} \end{cases}$

Hence, $\sum_{i=6}^{n-1} (\lceil \lg(\lfloor \lg(i) \rfloor) \rceil - \lceil \lg(\lceil \lg(i) \rceil - 2) \rceil) = \sum_{k=1}^x (2^{4 \times 2^{k-1}+1} + 1) + R + 12$

Table 1. Cumulative Savings in comparison in different intervals

Value of i	$[6, 2^4]$	$[2^4 + 1, 2^5 - 1]$	$[2^5, 2^6]$	$[2^6 + 1, 2^9 - 1]$	$[2^9, 2^{10}]$
Savings in comparison	12	$12 + 0 = 12$	$12 + 33 = 45$	$45 + 0 = 45$	$45 + 513 = 558$

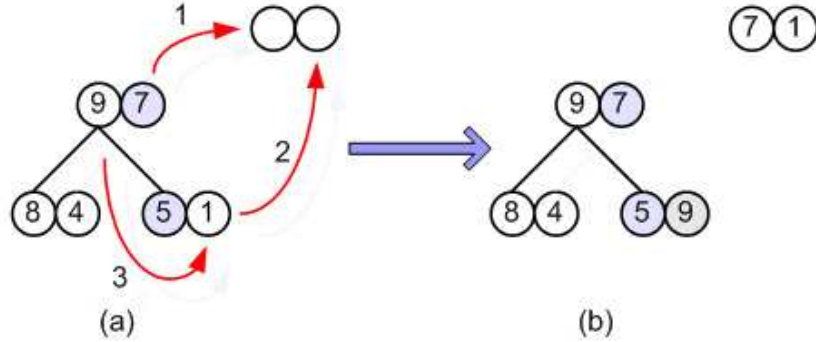
$$= \sum_{k=1}^x \left(2^{2^{k+1}+1} \right) + R + 12 + x. \text{ Now For simplicity, let}$$

$$p = \lfloor \lg(n-1) \rfloor - \lfloor \lg(n) \rfloor - \lfloor \lg(n+1) \rfloor - \lfloor \lg(\lfloor \lg(n) \rfloor - 2) \rfloor - \lfloor \lg(\lfloor \lg(n+1) \rfloor - 2) \rfloor$$

$$\begin{aligned} \text{Then we have, } C(n) - \tilde{C}(n) &= -(n-4) + \sum_{k=1}^x \left(2^{2^{k+1}+1} \right) + R + x + 12 + n + 2 + p \\ &= \sum_{k=1}^x \left(2^{2^{k+1}+1} \right) + 18 + R + x + p \end{aligned}$$

4 Analysis on the Number of Movements

Till now we have had discussion on comparisons. Now in case of movement our new data structure requires three movements for each key. It may be noted here that 'temp' is similar to a node capable of holding 2 data elements together. There are four different scenarios for sorting keys (two for 1st key of the pair and two for 2nd key of the pair).

**Fig. 1.** Scenario 1: (a)before copy (b)after copy

Scenario 1: If the 2nd key of the 1st pair is greater than the 1st key of the last pair (Figure 1(a), 1(b)), we move the 2nd key of the 1st pair to the 1st key of temp pair, move the 2nd key of the last pair to 2nd key of the temp pair and finally move the 1st key of the 1st pair to the 2nd key of the last pair.

Scenario 2: If the 2nd key of the 1st pair is less than the 1st key of the last pair (Figure 2(a), 2(b)), we move the last pair to the temp pair, the 1st key of

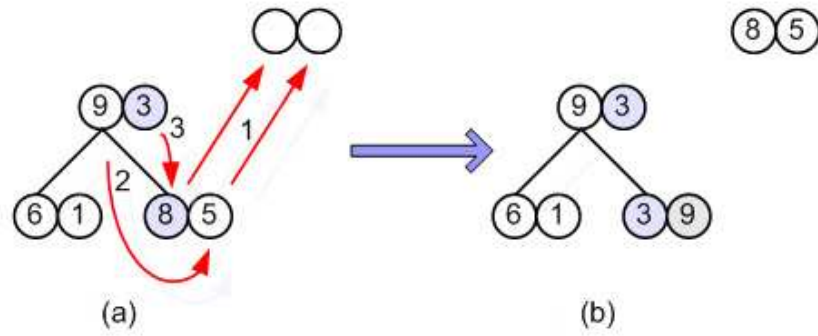


Fig. 2. Scenario 2: (a)before copy (b)after copy

the 1st pair to the 2nd key of the last pair and finally the 2nd key of the 1st pair to the 1st key of the last pair.

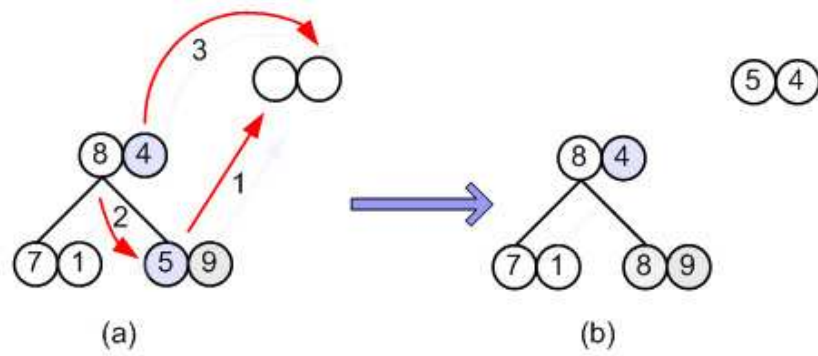


Fig. 3. Scenario 3: (a)before copy (b)after copy

Scenario 3: After sorting the 1st key for the 2nd one, if the 2nd key of the 1st pair is less than the 1st key of the last pair (Figure 3(a), 3(b)), we move the 1st key of the last pair to the 1st key of the temp pair, then the 1st key of the 1st pair moves to the 1st key of the last pair and finally we move the 2nd key of the 1st pair to 2nd key of the temp pair.

Scenario 4: If the 2nd key of the 1st pair is greater than the 1st key of the last pair (Figure 4(a), 4(b)), we move the 2nd key of the 1st pair to the 1st key of the temp pair, then move the 1st key of the last pair to the 2nd key of the temp pair and finally the 1st key of the 1st pair to the 1st key of the last pair.

In Carlsson's version of heapsort, it takes 2 movements- one for taking the last element to temp and the other one for moving root to the last unsorted

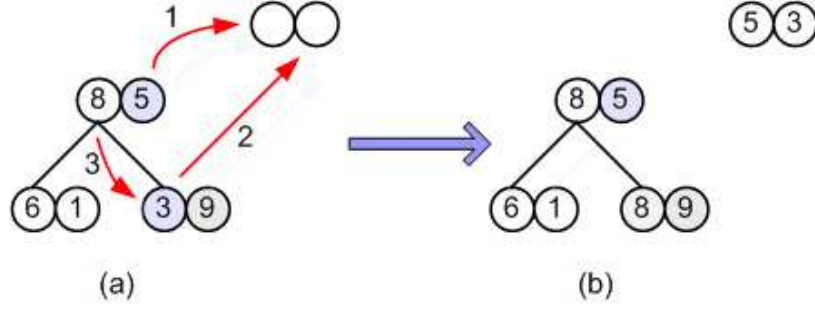


Fig. 4. Scenario 4: (a) before copy (b) after copy

location (Figure 5(a), 5(b)). So for movement calculation we take 1 extra movement than Carlsson's version. Dominating operations in sorting algorithms are combination of comparisons and movements. For different computer architectures, compilers and a few other things, time for comparison and movement varies [8]. Here we are presenting some latency of AMD64 processors (64-bit) for movement and comparisons [?]. We only consider here static movement latency and comparison latency. All other calculations, like effective address calculation, jumping decisions, are not considered here.

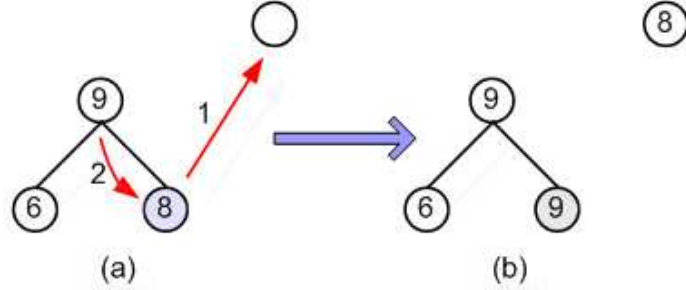


Fig. 5. Copy in Carlsson variant (a) before copy (b) after copy

In scenarios 1 and 2 (Figure 1(a), 1(b), 2(a), 2(b)), if we move the 1st and the last pair to register (from memory) we need three clock cycles for each and one clock cycle per movement for those three movements, as shown in figures mentioned above (Because those are from register to register). So, in total we take $(3+3+1+1+1=)$ 9 cycles for adjusting the first key and in scenario 3 and 4

we move the 1st pair to register (from memory) and it takes three clock cycles and those three movements take (as in Figure 3(a), 3(b), 4(a), 4(b)) three clock cycles (one for each movement). We are not saving anything in registers and moving back every thing to memory. So after sorting a pair of keys corresponding to a node we move it back to memory which takes three clock cycles. So, all this takes $(3+1+1+1+3=)$ 9 clock cycles. Now we can see that each key of a pair takes 9 clock cycles.

Considering Carlsson's version to move the last key from memory to register it takes three clock cycles and to move the 1st key to the last unsorted position it takes six clock cycles (3 for moving memory to register and 3 for moving register to memory). In all it takes 9 clock cycles to adjust a key. Though we make more movements than Carlsson's variant, application of the technique shown above ensures that in terms of clock cycle requirement we are at par even in case of time requirements for movements. Moreover, we perform sorting with lesser number of comparisons than Carlsson's version and for both the proposed data structures and the existing ones it costs 4 clock cycles to compare register with memory or memory with register.

5 Conclusion

Our analysis and experimental results indicate that the new data structure for heaps result in better performance of the heapsort algorithm. It appears to be superior in terms of numbers of comparisons. Although it requires more movements, the technique shown does indicate that in terms of clock cycles required for movement the proposed data structure is evenly comparable with existing data structures. Expressions for decrease in the number of comparisons remained as an explicit sum and could not be expressed in closed form although it is evident from experimental results that in the best case this savings is around $n/2$ and for the worst case it is as low as 0, and these happen in alternate intervals as shown in Table 1.

References

1. M. D. Atkinson, J. R. Sack, N. Santoro and T. Strothotte, Min-max heaps and generalized priority queues, programming techniques and data structures, *Communications of the ACM*, 29(10), 996-1000, October 1986.
2. S. Carlsson, A variant of HEAPSORT with almost optimal number of comparisons, *Information Processing Letters*, 24, 247-250, 1987.
3. R. A. Chowdhury, S. K. Nath and M Kaykobad, A Simplified Complexity Analysis of McDiarmid and Reed's Variant of Bottom-up Heapsort Algorithm, *International Journal of Computer Mathematics*, 73, 293-297, 2000.
4. R. A. Chowdhury, M. Z. Rahman and M Kaykobad, On the bounds of min-max pair heap construction, *Computers and Mathematics with Applications*, 911-916, 43, 2002.
5. R. D. Dutton, Weak-heap sort, *BIT*, 33, 372-381, 1993.

6. R. D. Dutton, The weak-heap data structure. *Technical report*, University of Central Florida, Orlando, FL 32816, 1992.
7. R. W. Floyd, ACM algorithm 245, Treesort 3, *Communications of the ACM*, 7(12), 701, 1964.
8. J. L. Hennessy and D. A. Patterson, *Computer Architecture, A quantitative approach*, 3rd edition, Morgan Kaufmann Publishers, 2003.
9. T. M. Islam and M. Kaykobad, Worst-case Analysis of Generalized Heapsort Algorithm Revisited, *International Journal of Computer Mathematics*, 83(1), 59-67, January 2006.
10. J. Katajainen, The ultimate heapsort, *In Proceedings of the Computing: the 4th Australasian Theory Symposium, Australian Computer Science Communications*, 20(3), 87-95, 1995.
11. C. J. H. McDiarmid and B. A. Reed, Building heaps fast, *Journal of Algorithms*, 10, 352-365, 1989.
12. S. Olariu, C. M. Overstreet and Z. Wen, A mergeable double-ended priority queue, *Computer Journal*, 34, 423-427, 1991.
13. A. Paulik, Worst-case analysis of a generalized heapsort algorithm, *Information Processing Letters*, 36, 159-165, 1990.
14. 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, 81-98 1993.
15. J. W. J. Williams, ACM algorithm 232, Heapsort, *Communications of the ACM*, 7(6), 347-348, 1964.
16. Advanced Micro Devices, *Software Optimization Guide for AMD64 Processors*, Publication # 25112, Revision: 3.06, September, 2005.