Heapsort using Multiple Heaps

D. Levendeas[1], C. Zaroliagis[1, 2]

[1]Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece
{lebenteas, zaro}@ceid.upatras.gr
[2]Computer Technology Institute, N. Kazantzaki Str, Patras University Campus,
26500 Patras, Greece

Abstract

We present a modification of the in-place algorithm HEAPSORT using multiple heaps. Our algorithm reduces the number of swaps made between the elements of the heap structure after the removal of the root at the cost of some extra comparisons. In order to retain the property of sorting in-place, we perform a kind of multiplexing and store all heaps in a single array such that no extra space is required. We also present a theoretical analysis of our algorithm, providing a specific formula for computing the optimal number of heaps to be used, and back up our theoretical findings with an experimental study, which shows the superiority of our approach with respect to the classical HEAPSORT.

1. Introduction

Heapsort [Williams, (1964)] as described in the original paper is one of the most widely used sorting algorithms which runs in *O(n log n)* time and sorts in-place (*log* implies throughout the paper base 2). It is known that *O(n log n)* is the optimal asymptotic running time upper bound for any comparison-based sorting algorithm. Even though mergesort gets closely to this optimum, it requires an array of *O(n)* auxiliary space. Heapsort belongs to the family of algorithms that sort by selection and requires *O(1)* auxiliary space, falling into the category of in-place algorithms. An in-place sorting algorithm allows maximizing the usage of the main memory using input data instead of keeping a large amount of auxiliary data. Several efforts have been made to improve the performance of sorting, because it is one of the most crucial problems in computer science [Knuth, (1998)] (section 5.5). Consequently, improved versions of heapsort have been proposed in the literature. Bottom-up-heapsort [Wegener, (1993)] and ”shiftdown” process [Schaffer and Sedgewick, (1993)] modify the way the heap property is maintained by changing how it is constructed or reconstructed after the removal of its root, while weak-heapsort [Edelkamp and Wegener, (2000)] defines a new heap structure and outperforms the other implementations in some cases [Edelkamp and Wegener, (2000), Edelkamp and Stiegeler, (2001)]. Weak-heapsort uses one more bit for each element in the heap that is not in-place.

In this work, we propose the use of multiple heaps in the heapsort algorithm, in order to decrease the height of the required trees (and hence the time for their manipulation), thus achieving better performance. To retain the in-place property in our variant, we perform a kind of multiplexing so that different heaps are organized in the same array without using any auxiliary space. We also present a theoretical analysis of our algorithm, providing a specific formula for computing the optimal number of heaps to be used, and back up our theoretical findings with an experimental study, which shows the superiority of our approach with respect to the classical HEAPSORT.

2. HEAPSORT USING TWO HEAPS

2.1. What is a heap

Heap is an elementary data structure often used in applications concerned with priority queues and ordering [Hwang, (1997)]. It first appeared when the heapsort [Williams, (1964)] was described. Besides its original application to sorting, heap has wide applications to algorithmic design [Aho et al., (1974), Knuth, (1998), Mehlhorn and Tsakalidis, (1990), Cormen et al., (2001)].

Heapsort works similarly using either max heaps or min heaps. We work throughout this paper using min heaps, but the same results apply in the case when a max heap is used.

A min heap is an array of elements a_j, $1 \le j \le n$, satisfying the so-called path-monotonic property [Hwang and Steyaert, Knuth, (1998), Hwang, (1997)]: $a_j \ge a_{\lfloor \frac{j}{2} \rfloor}$ for $j = 2, 3,\ldots,n$ where $\lfloor x \rfloor$ denotes the integral part of the real number x. There are several ways in which a heap can be viewed. One of the most popular perspectives is as a nearly complete binary tree in which the heap property is preserved; that is, the key of each parent has value no smaller than that of its children. We assume that the tree is represented as a one-dimensional array.

Let i denote (the array position of) a particular node. On most implementations, the left child of i is located at the node $2i$ which can be computed in one instruction (by left shifting the binary representation of i by one position), while the right child is located at the node $2i + 1$ (which can be computed in one instruction too, if there is an FMA (= Fused Multiply-Add) instruction in the instruction set). The parent of a node can be found by computing the position $\left\lfloor \frac{i}{2} \right\rfloor$ [Cormen et al., (2001)].

A more recursive approach could be that each node is the root of two subtrees where the heap property is valid and has the no smaller value of all the elements of those

two trees. This perspective enables us to view a heap as the union of more heaps and that is the first step towards our algorithm. If we omit the nodes above a specific level k, the remainder can be viewed as 2^k independent heaps of depth $d - k$, where d stands for the depth of the initial heap.

2.2. How heapsort works

Heapsort initially creates a heap using the input elements. The construction of a heap of n elements takes $O(n)$ time using a function called Build-Heap which calls the heapify function [Cormen et al., 2001].

Let $A[1..n]$, where $n = length[A]$, represent the heap array, i.e., an almost complete binary tree. Given a tree that is a heap except for node i, the heapify function arranges node i and its subtrees to satisfy the heap property. It can be thought of as letting the value at $A[i]$ to successively shift down in the heap, assuming that its children are already heaps, so that the subtree rooted at index i satisfies the heap property.

The Build-Heap function exploits the fact that the elements in the subarray $A[(\left\lfloor \frac{n}{2} \right\rfloor + 1)..n]$ are all leaves of the tree, and so each such element is a 1-element heap to begin with. Hence, it goes through the remaining nodes of the tree and runs the heapify function on each one, creating a binary heap.

Heapsort works as follows. A heap is constructed by calling the Build-Heap function. The smallest value is extracted repeatedly until the heap is empty. The values having been extracted are placed in sorted order. After each extraction, the heap property is preserved at a cost of $O(\log n)$ by calling the heapify function. Thus, heapsort runs in $O(n \log n)$ time because it creates a heap in $O(n)$ time and then extracts n elements, spending for each one $O(\log n)$ time concluding to $O(n \log n) + O(n) = O(n \log n)$ running time. During extraction, the only space required is that for storing the heap. In order to achieve constant space overhead, the heap is stored in the part of the input array that has not been sorted yet.

2.3. Using two heaps

The basic idea is to use the fact that after the removal of the root, instead of recreating the heap, the remainder of the structure can be viewed as two different heaps in which their roots are the nodes which were children of the node that has just been removed. One of these two roots is the next element that is going to be placed to the sorted elements after the next iteration of the algorithm. At the cost of one comparison we can determine which of them will be the appropriate one. So, there is no need of reconstructing the heap yet.

After the transfer of the root to the sorted part of the heap, its place is given to the last element of the unsorted heap (the element at the last position of the initial). Its size has been decreased by one element (the element that has just been swapped). Now, the heap property has been violated because a descendant has been placed in the root position. In order to restore the heap, we have to perform *O(log n)* necessary operations (swaps and comparisons) until the property is restored by calling heapify. However, we do not do it immediately. We "forget", for the time being, the existence of the root. As a result, we have two independent heaps now, its subtrees. We select the minimum root of these two at the cost of one comparison and we swap it with the last element of the heap. So far, two elements have been transferred in the sorted part of the heap. Now, we have to rebuild the initial heap in order to repeat the same process for the rest of the elements.

We call the heapify function in order to restore the heap property to the heap from which the second element was extracted. This heap has depth $d' = d - 1$, where d' is its depth and $d = \log n$ is the depth of the initial heap. Thus, the heapify function runs in $O(d') = O(d - 1)$ time. The merging of the two heaps into the initial one can be performed by calling the heapify function to the "forgotten" root of the initial heap. We repeat the same process until two or fewer elements have been remained. The sorting of two elements is trivial.

These two steps of the algorithm are illustrated in Figures 1, 2 and 3.

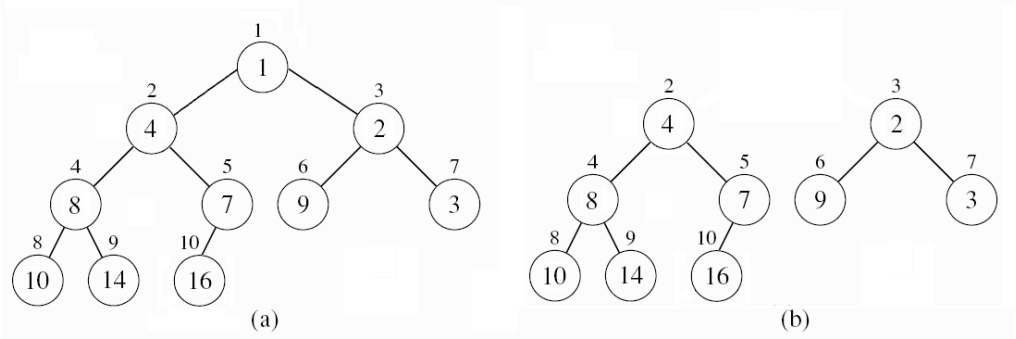

Figure 1: *(a) The initial heap and (b) the two heaps that are created after the removal of the root.*

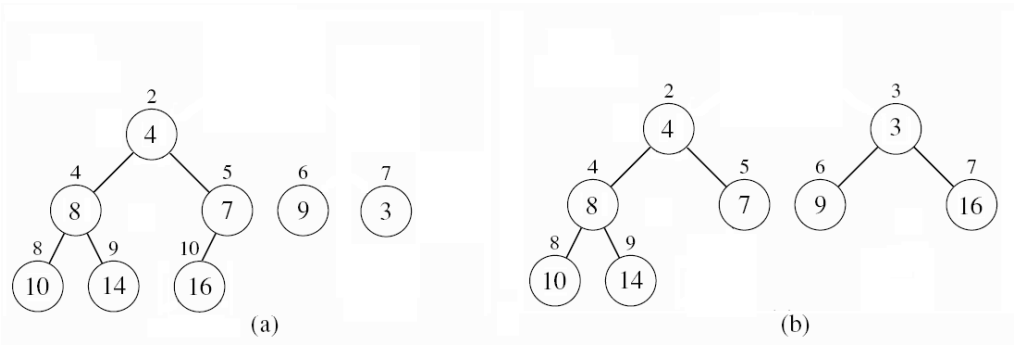

Figure 2: *(a) The root of the second heap is the extracted element because it's the smallest element of the two roots. (b) Afterwards, the second heap is restored, transferring the last element of the array into the root position and calling heapify.*

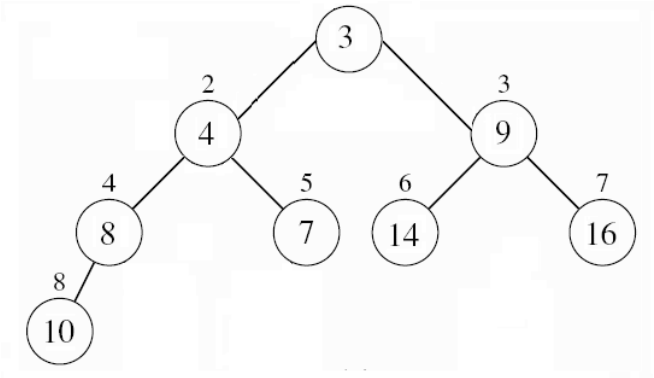

Figure 3: *The initial heap is restored transferring the last element into the root position by calling heapify.*

In this way, when we have one heap which contains all the elements, the algorithm runs as it was originally described in [Williams, 1964]. On the other hand, when the two heaps are used, at the cost of one comparison (the two elements in the root position of the two heaps) we gain two comparisons. These comparisons are performed when we place the last element of the (initial) heap in the position of the root and then restore the heap property. This requires two comparisons among the three elements in order to determine the one with the smaller value.

The above discussion implies that in two consecutive extract-min operations our algorithm performs one extra comparison and two swaps while it calls the heapify function for trees of depth $d' = d - 1$ and d. On the other hand, the original algorithm calls the heapify process for a tree of depth d twice after two swaps, in order to transfer the smallest element each time to the ordered part of the heap. Recalling that the heapify function has to perform two comparisons and one swap for each level, we

save one swap and one comparison for every two steps of the algorithm, because the second time the heapify function is called in a tree of smaller depth.

This gain occurs for $\frac{n-2}{2}$ of the iterations of our algorithm, because in the rest of them we are executing the original algorithm without modifications, while the sorting of the two remaining elements is trivial at the end. We have already observed that the use of two heaps reduces the depth of the heap trees and thus it reduces the cost of heapify. This leads us to a refinement of our algorithm trying to make this happen for more iterations. The ideal would be for *n – 2* iterations and in order to achieve this, ot would require the use of at least two heaps constantly without merging them into one in any step. This is the main idea of our approach.

3. ORGANISING MULTIPLE HEAPS IN ONE ARRAY

3.1. Description of the problem

To be able to use more than one heap during the heapsort and keep the algorithm in-place, we need a way of having more than one heaps in the same space and the functions computing the parent and the children of a particular node.

There are some restrictions on how the heaps may be organised in the array. The most obvious way of placing them one after another fails because after the decrement of the size of a heap, the free space that is left is not continuous, so there is no obvious way to keep the sorted elements in this space. As a consequence, we are forced to choose an alternative way of storing the multiple heaps in the same array and this is by multiplexing them and changing the functions that compute the parent and the children of a node.

3.2. Multiplexing the heaps in the same array

We make use of zero indexed arrays. The proposed solution consists of placing the root of each heap in the first positions of the array, one next to each other, starting from the beginning of the array. Let *N* be the number of the heaps we are using.

The left child of each node *i* is located at the position

$$2i + N \quad \textbf{(1)}$$

the right child of the node *i* is located at the position

$$2i + N + 1 \quad \textbf{(2)}$$

while the parent of a node is in the position

$$\left\lfloor \frac{i-N}{2} \right\rfloor \quad \textbf{(3)}$$

It is worth to mention that the position of a child (or of the parent) can be found with only one operation if the FMA operation is in the instruction set. It is easy to see that functions (1) and (2) that determine the position of the children of node *i* guarantee that the elements of one heap do not overwrite the elements of another heap.

4. Cost Analysis

4.1. General Idea

Instead of one heap, we use *N* heaps. After the arrangement of the heaps using the idea above, the sorting can be done with only a few changes compared to the original algorithm. It is sufficient to use a function that finds which of the *N* roots is the minimum one, and swaps it with the last unordered element. Such a function requires *N – 1* comparisons. After placing that element in the root, the heap property is restored using heapify (the function which maintains the heap property) on the heap that has been altered during this step. The rest of the heaps remain as they were because they are not affected by the removal of the root of another heap. Our search is limited to the *N* roots, because the root of a min-heap always contains the element with the minimum key.

For the sake of simplicity, we can assume that *N* is a power of *2* even though this does not change anything in the implementation of the algorithm. As we mentioned earlier in the text, a heap can be viewed recursively as the union of many heaps. By having *N* heaps, we have omitted *log N* levels having totally *N - 1* elements. These elements will cause *N* other elements to appear in the last level of the heaps as leaves. However, the depth of the heaps is now decreased due to the move of those *N -1* elements to the leaves of the heaps. The new depth is *log (n + N) – log N*. An additional cost is that of finding the minimum of the *N* roots. Due to the breaking of the heap into *N* heaps, we have to examine *log N* less levels than before. A node can be placed in each one of these levels requiring two comparisons and one swap. Because of these levels being near to the root, there is a high probability that a node may have to travel through all these levels during the heapify phase.

We examine only the cost of extracting elements from the heaps because the aggregate cost of creating *N* heaps of *n* total size is similar to the cost of creating one heap of size *n*. This holds because the changes in the heapify function are minor and do not affect the total cost. Recall that we have changed only the way that the children are computed.

4.2. Cost Analysis using N heaps

Let us examine briefly the cost of sorting the *n - N* elements using *N* heaps in comparison with the classical implementation of the heapsort. The remainder of the elements, the last *N* elements which consist of the roots of the *N* heaps, can be sorted using any sorting algorithm.

Let *c* denote the cost of a comparison and *s* denote the cost of a swap operation.

The complexity of using *N* heaps is:

- The number of the levels of each heap is *d = (log (n+N) – log N)* because the structure could be viewed as one heap where the elements of its first *log N* levels have been transferred to the last one. Their addition to the last level increases the size of the heap which is now given by *log (n+N)* but by omitting these elements from the first levels, the depth of the heap is decreased by *log N*.
- Each one of the *n - N* elements requires *2* comparisons and *1* swap for each level in which is ”shifted down”
- *N - 1* comparisons to select the element among the roots of the heaps that satisfies the condition (has the minimum or the maximum key, according to the type of sorting) in order to be transferred in the list of the ordered elements
- Because of the transfer of *(N – 1) (log (n + N) – log N)* elements to the last level *d* of the heaps, there is an additional cost due to the fact that these extra elements have to travel all the way to one of the roots. This cost is two comparisons and one swap for each level as stated above.

Hence, the total cost is:

$$(n-N)\cdot d\cdot(2c+s)+(n-N)(N-1)\cdot c+(N-1)\cdot d\cdot(2c+s)+O(N\log N) \quad \textbf{(4)}$$

where *O(N log N)* stands for the cost of sorting the *N* elements that have been remained at the roots when all the *N* heaps have exactly one element.

Omitting the last term, expression (4) can be written in simplified form as:

$$[(2n-2)d+N(n-N+1)-n]c+d(n-1)s \quad \textbf{(5)}$$

To simplify the comparison of the classical implementation, we omitted the sorting time of the last *N* elements, and do the same in the analysis of the classical implementation. Hence, the cost of the classical implementation for *n – N* elements, after the initial phase when all the heaps have been created, is:

$$(n-N)\cdot\log n\cdot(2\cdot c+s) \quad \textbf{(6)}$$

To find the optimal value of *N*, we subtract (5) from (6), take the derivative of this difference, equal it to zero, and solve the resulting equation. This will give us the point where the gain takes its highest value and use that *N* as the number of heaps.

5. EXPERIMENTAL ANALYSIS

In order to confirm our theoretical analysis, we ran an experimental study measuring the speed-up of the performance of our algorithm with *N=2* heaps, which turned out to be the best value for N in our testing environment. All of our experiments have been performed on a Pentium 4, 3.2 GHz, 1 GB RAM, under Ubuntu Linux 8.10. The code was written in C, compiled by gcc 4.2.3, using the optimization flag –O2. Our tests showed a boost-up in performance of about 10% on the average.

On the first experiment, we used two implementations of heapsort in order to emphasize the fact that the number of operations reduced is strictly related to the size of input and not to the implementation. The first implementation that we used is a very efficient one, based on the book "Numerical Recipes" and referred by wikibooks, while the other one is more modular and easily implemented, but inefficient. As it is shown in Figure 4, the faster implementation presents greater acceleration. The input we used used is a uniform distribution of random elements. We output the speed-up achieved in comparison to the classical algorithm.

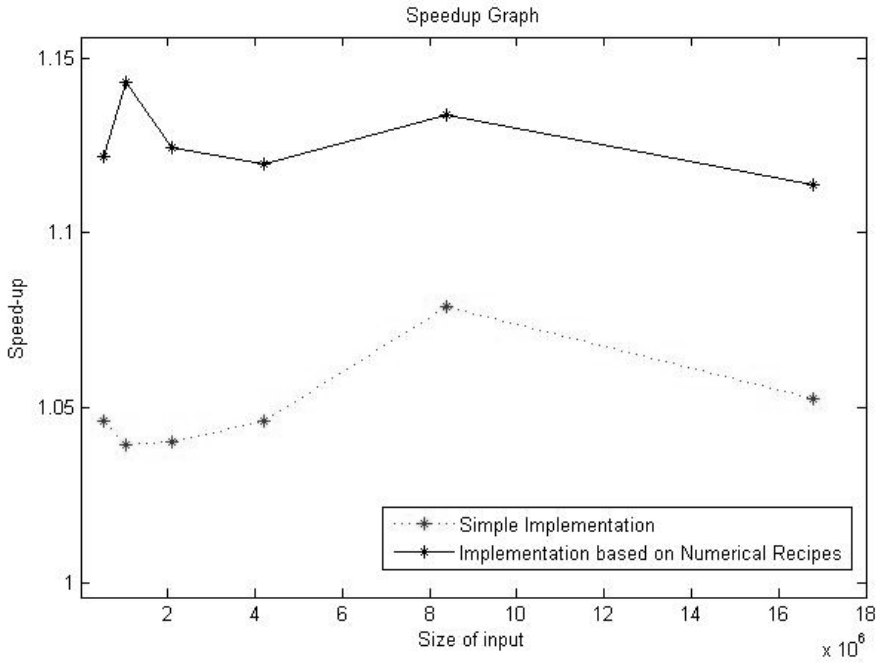

Figure 4. *Speed-up of two implementations of our algorithm in comparison to the classical algorithm.*

Except for data drawn according to the uniform distribution, we also tested the behaviour of our algorithm on different data sets. We tested the case where all elements are equal. This is the best case for heapsort. We observed that there was a decrement of the speed-up, but the overall process was executed very fast in this special case (in linear time because no actual sorting was taking place), so this delay is negligible compared to the time to accomplish the sorting. Furthermore, there is no practical need to use a sorting algorithm on equal elements.

On the other hand, we tested our algorithm using already sorted arrays, one in ascending and one in descending order. That makes the build-heap process trivial in both cases (original algorithm and our modification). The results that we obtained showed a speed-up in performance somewhat above 10% and are similar to those of the uniform distribution. Figure 5 shows the speed-up compared to the classical implementation.

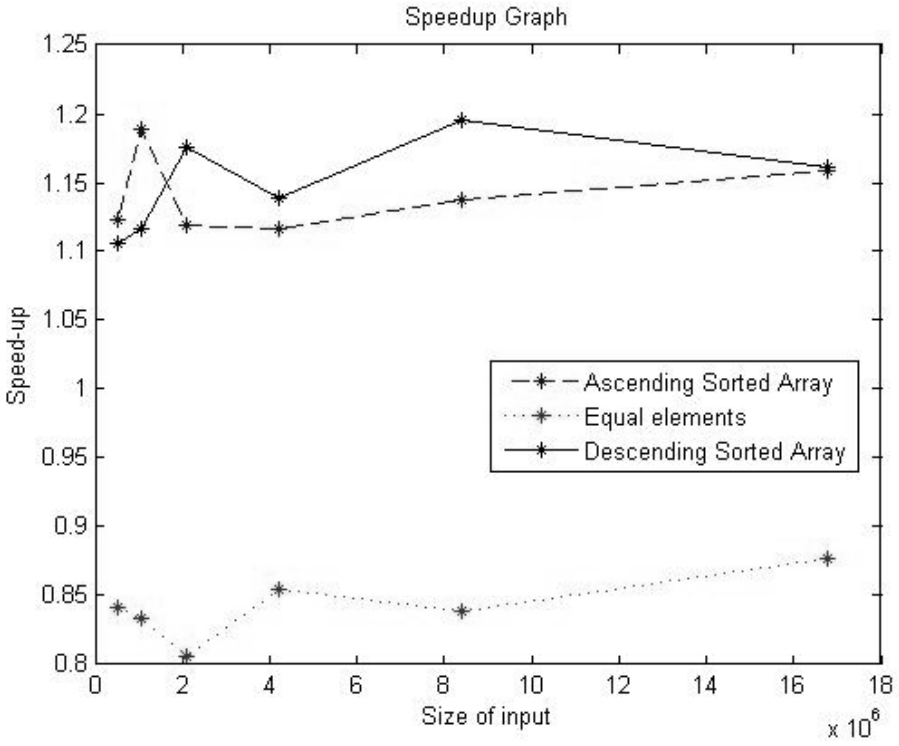

Figure 5. *Different Data Sets.*

6. Research Perspectives

Some interesting perspectives that could be the ground for further research are improvements focused in the way that the roots of the heaps are manipulated in order to extract the minimum of them. We could study the case of keeping a sorted array with the roots of the heaps. After the removal of a root from this array, the insertion of the new root would take place in a sorted array, giving us the possibility to insert it in logarithmic time. Another interesting perspective would be to investigate the parallelization of our approach, which is inherent to our algorithm (each heap can be treated separately).

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

2. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to algorithms.* MIT Press, Cambridge, MA, USA, 2001. ISBN 0-262-03293-7.

3. Stefan Edelkamp and Patrick Stiegeler. *Pushing the limits in sequential sorting.* Lecture Notes in Computer Science, 1982:39+, 2001.

4. Stefan Edelkamp and Ingo Wegener. *On the performance of WEAK–HEAPSORT.* Lecture Notes in Computer Science, pages 254–??, 2000.

5. Hwang. *Optimal algorithms for inserting a random element into a random heap.* IEEETIT: IEEE Transactions on Information Theory, 43, 1997.

6. H. Hwang and J. Steyaert. *The number of heaps and the cost of heap construction.*

7. Donald E. Knuth. *Sorting and Searching,* volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, USA, second edition, 1998. ISBN 0-201-89685-0.

8 . K. Mehlhorn and A. Tsakalidis. *Data structures.* pages 302–341, 1990.

9. Russel Schaffer and Robert Sedgewick. *The analysis of heapsort.* J. Algorithms, 15(1):76–100, 1993. ISSN 0196-6774.

10. Ingo Wegener. *Bottom-up-heapsort, a new variant of heapsort beating, on an average, quicksort (if n is not very small).* Theor. Comput. Sci., 118(1):81–98, 1993. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(93)90364-Y.

11. J. W. J. Williams. *Algorithm 232: Heapsort.* Communications of the ACM, 7:347–348, 1964.