On the Performance of *WEAK-HEAPSORT*

Stefan Edelkamp[1] and Ingo Wegener[2]

[1] Institut für Informatik, Albert-Ludwigs-Universität,
Am Flughafen 17, D-79110 Freiburg;
eMail: edelkamp@informatik.uni-freiburg.de
[2] Lehrstuhl 2, Fachbereich Informatik
Universität Dortmund, D-44221 Dortmund
wegener@ls2.cs.uni-dortmund.de

Abstract. Dutton (1993) presents a further *HEAPSORT* variant called *WEAK-HEAPSORT*, which also contains a new data structure for priority queues. The sorting algorithm and the underlying data structure are analyzed showing that *WEAK-HEAPSORT* is the best *HEAPSORT* variant and that it has a lot of nice properties.
It is shown that the worst case number of comparisons is $n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + n - \lceil \log n \rceil \leq n \log n + 0.1n$ and weak heaps can be generated with $n-1$ comparisons. A double-ended priority queue based on weak-heaps can be generated in $n + \lceil n/2 \rceil - 2$ comparisons.
Moreover, examples for the worst and the best case of *WEAK-HEAPSORT* are presented, the number of *Weak-Heaps* on $\{1, \ldots, n\}$ is determined, and experiments on the average case are reported.

1 Introduction

General sequential sorting algorithms require at least $\lceil \log(n!) \rceil = n \log n - n \log e + \Theta(\log n) \approx n \log n - 1.4427n$ key comparisons in the worst case and $\lceil \log(n!) \rceil - \frac{1}{n!} 2^{\lceil \log(n!) \rceil} \leq \lceil \log(n!) \rceil - 1$ comparisons in the average case. We assume that the time for all other operations should be small compared to the time of a key comparison. Therefore, in order to compare different sorting algorithms the following six criteria are desirable:

1. The sorting algorithm should be general, i.e., objects of any totally ordered set should be sorted.
2. The implementation of the sorting algorithm should be easy.
3. The sorting algorithm should allow internal sorting, i.e., beside the space consumption for the input array only limited extra space is available.
4. For a small constant c the average case on key comparisons should be less than $n \log n + cn$.
5. For a small constant c' the worst case on key comparisons should be less than $n \log n + c'n$.
6. The number of all other operations such as exchanges, assignments and other comparisons should exceed the number of key comparisons by at most a constant factor.

BUCKETSORT and its *RADIXSORT* variants (Nilsson (1996)) are not general as required in Property 1.

Given $n = 2^k$ traditional *MERGESORT* performs at most $n \log n - n + 1$ key comparisons, but requires $O(n)$ extra space for the objects, which violates Property 3. Current work of J. Katajainen, T. A. Pasanen, and J. Teuhola (1996), J. Katajainen and T. A. Pasanen (1999), and K. Reinhardt (1992) shows that *MERGESORT* can be designed to be *in-place* and to achieve promising results, i.g. $n \log n - 1.3n + O(\log n)$ comparisons in the worst case. However, for practical purposes these algorithms tend to be too complicated and too slow.

INSERTIONSORT (Steinhaus (1958)) invokes less than $\sum_{i=1}^{n-1} \lceil \log(i+1) \rceil = \log(n!) + n - 1$ key comparisons, but even in the average case the number of exchanges is in $\Theta(n^2)$ violating Property 6.

SHELLSORT (Shell (1959)) requires as an additional input a decreasing integer sequence $h_1, \ldots, h_t = 1$. According to these distances of array indices traditional *INSERTIONSORT* is invoked. The proper choice of the distances is important for the running time of *SHELLSORT*. In the following we summarize the worst case number of operations (comparisons and exchanges). For $n = 2^k$ Shell's original sequence $(2^k, \ldots, 2, 1)$, leads to a quadratic running time. A suggested improvement of Hibbard (1963) achieves $O(n^{3/2})$ with the analysis given by Papernov and Stasevich (1965). Pratt (1979) provided a sequence of length $\Theta(\log^2 n)$ that led to $\Theta(n \log^2 n)$ operations. Sedgewick (1986) improves the $O(n^{3/2})$ bound for sequences of maximal length $O(\log n)$ to $O(n^{4/3})$ and in a joint work with Incerpi (1985) he further improves this to $O(n^{1+\epsilon/\sqrt{\log n}})$ for a given $\epsilon > 0$. Based on incompressibiliy results in Kolmogorov complexity, a very recent result of Jiang, Li and Vitányi (1999) states that the average number of operations in (so-called p pass) *SHELLSORT* for any incremental sequence is in $\Omega(pn^{1+1/p})$. Therefore, *SHELLSORT* violates Properties 4. and 5.

QUICKSORT (Hoare (1962)) consumes $\Theta(n^2)$ comparisons in the worst case. For the average case number of comparisons $V(n)$ we get the following recurrence equation $V(n) = n - 1 + \frac{1}{n} \sum_{k=1}^{n} (V(k-1) + V(n-k))$. This sum can be simplified to $V(n) = 2(n+1)H_n - 4n$, with $H_n = \sum_{i=1}^{n} 1/i$, and to the approximation $V(n) \approx 1.386 n \log n - 2.846n + O(\log n)$.

Hoare also proposed *CLEVER-QUICKSORT*, the median-of-tree variant of *QUICKSORT*. In the worst case we still have $\Theta(n^2)$ key comparisons but the average case number can be significantly reduced. A case study reveals that the median of three objects can be found in $8/3$ comparisons on the average. Therefore, we have $n - 3 + 8/3 = n - 1/3$ comparisons in the divide step leading to the following recurrence for the average case $V(n) = n - 1/3 + \binom{n}{3}^{-1} \sum_{k=1}^{n} (k-1)(n-k)(V(k-1) + V(n-k))$. This sum simplifies to $V(n) = \frac{12}{7}(n+1)H_{n-1} - \frac{477}{147}n + \frac{223}{147} + \frac{252}{147n} \approx 1.188 n \log n - 2.255n + O(\log n)$ (Sedgewick (1977)). No variant of *QUICKSORT* is known with $n \log n + o(n \log n)$ comparisons on average (cf. van Emden (1970)). Hence, it violates Properties 4. and 5.

The worst case case number on key comparisons in *HEAPSORT* independently invented by Floyd (1964) and Williams (1964) is bounded by $2n \log n + O(n)$. For the generating phase less than $2n - 2$ comparisons are required.

BOT-TOM-UP-HEAPSORT (Wegener (1993)) is a variant of *HEAPSORT* with $1.5n \log n + O(n)$ key comparisons in the worst case. The idea is to search the

path to the leaf independently to the place for the root element to sink. Since the expected depth is high this path is traversed bottom-up. Fleischer (1991) as well as Schaffer and Sedgewick (1993) give worst case examples for which *BOT-TOM-UP-HEAPSORT* requires at least $1.5n \log n - o(n \log n)$ comparisons. Based on the idea of Ian Munro (cf. Li and Vitányi (1992)) one can infer that the average number of comparisons in this variant is bounded by $n \log n + O(n)$.

MDR-HEAPSORT proposed by McDiarmid and Reed (1989) performs less than $n \log n + cn$ comparisons in the worst case and extends *BOT-TOM-UP-HEAPSORT* by using one bit to encode on which branch the smaller element can be found and another one to mark if this information is unknown. The analysis that bounds c in *MDR-HEAPSORT* to 1.1 is given by Wegener (1993). *WEAK-HEAPSORT* is more elegant and faster. Instead of two bits per element *WEAK-HEAPSORT* uses only one and the constant c is less than 0.1.

In order to ensure the upper bound $n \log n + O(n)$ on the number of comparisons, *ULTIMATE-HEAPSORT* proposed by Katajainen (1998) avoids the worst case examples for *BOT-TOM-UP-HEAPSORT* by restricting the set of heaps to two layer heaps. It is more difficult to guarantee this restricted form. Katajainen obtains an improved bound for the worst case number of comparisons but the average case number of comparisons is larger as for *BOT-TOM-UP-HEAPSORT*. Here we allow a larger class of heaps that are easier to handle and lead to an improvement for the worst case *and* the average case.

There is one remaining question: How expensive is the extra space consumption of one bit per element? Since we assume objects with time-costly key comparisons we can conclude that their structure is more complete than an integer, which on current machines consumes 64 bits to encode the interval $[-2^{63} - 1, 2^{63} - 1]$. Investing one bit per element only halves this interval.

The paper is structured as follows. Firstly, we concisely present the design, implementation and correctness of the *WEAK-HEAPSORT* algorithm with side remarks extending the work of Dutton (1993). Secondly, we determine the number of *Weak-Heaps* according to different representation schemas. Afterwards we prove that there are both worst and best case examples that exactly meet the given bounds. Finally, we turn to the use of *Weak-Heaps* as a priority queue.

2 The *WEAK-HEAPSORT* Algorithm

2.1 Definition *Weak-Heap* and Array-Representation

A (Max-) *Weak-Heap* is established by relaxing the heap condition as follows:

1. Every key in the right subtree of each node is smaller than or equal to the key at the node itself.
2. The root has no left child.
3. Leaves are found on the last two levels of the tree only.

An example of a *Weak-Heap* is given in Figure 1. The underlying structure to describe the tree structure is a combination of two arrays. First, we have the array a in which the objects are found and second, the array of so-called *reverse* bits represents whether the tree associated with a node is rotated or not.

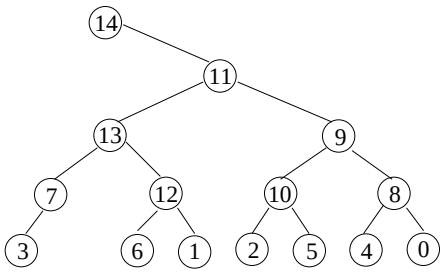

Fig. 1. Example of a *Weak-Heap*.

For the array representation we define the left child of index i as $2i + r_i$ and the right child as $2i + 1 - r_i$, $0 \leq i \leq n - 1$. Thus by flipping r_i we exchange the indices of the right and left children. The subtree of i is therefore rotated. For example, one array representation according to Fig. 1 is $a = [14, 11, 9, 13, 8, 10, 12, 7, 0, 4, 5, 2, 1, 6, 3]$ and $r = [0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]$.

The successors of a node on the left branch of the right subtree are called grandchildren. For the example of Figure 1 the grandchildren of the root are labeled by 11, 13, 7 and 3. The inverse function `Gparent(x)` (for *grandparent*) is defined as `Gparent(Parent(x))` in case x is a left child and `Parent(x)` if x is a right one. `Gparent(x)` can be calculated by the following pseudo-code `while(odd(`x`) = `$r_{x/2}$`)` $x \leftarrow x/2$ followed by `return` $x/2$ with an obvious interpretation of `odd`.

2.2 Generating Phase

Let y be the index of the root of a tree T and x be the index of a node with $a_x \geq a_z$ for all z in the left subtree of T and let the right subtree of T and y itself be a *Weak-Heap*. *Merging* x and y gives a new *Weak-Heap* according to the following case study. If $a_x \geq a_y$ then the tree with root x and right child T is a *Weak-Heap*. If, however, $a_y > a_x$ we swap a_y with a_x and rotate the subtrees in T. By the definition of a *Weak-Heap* it is easy to see that if the leaves in T are located only on the last two levels merging x with y results in a *Weak-Heap*. The pseudo-code according to `merge` is given by `if` $(a_x < a_y)$ `swap(`a_x`,`a_y`)`; $r_y \leftarrow 1 - r_y$.

In the generating phase all nodes at index i for decreasing $i = n-1, \ldots, 1$ are merged to their grandparents. The pseudo-code for the so-called `WeakHeapify` procedure can be specified as: `for` $i \in \{n - 1, \ldots, 1\}$ `Merge(Gparent(i),i)`.

Theorem 1. `WeakHeapify` *generates a* Weak-Heap *according to its definition.*

Proof. Assume that there is an index y, such that `Merge(Gparent(y),y)` does not return a *Weak-Heap* at $x =$ `Gparent(y)`. Then choose y maximal in this sense. Since all nodes $w > y$ with `Gparent(w)` $= x$ have led to a correct *Weak-Heap*, we have $a_x \geq a_z$ for all z in the left subtree of root y. On the other hand

y and its right subtree already form a *Weak-Heap*. Therefore, all preconditions of merging x with y are fulfilled yielding a contradicting *Weak-Heap* at root x.

One reason why the *WEAK-HEAPSORT* algorithm is fast is that the generating phase requires the minimal number of $n-1$ comparisons.

Note that the `Gparent` calculations in `WeakHeapify` lead to several shift operations. This number is linear with respect to the accumulated path length $L(n)$, which can recursively be fixed as $L(2)=1$ and $L(2^k)=2\cdot L(2^{k-1})+k$. For $n=2^k$ this simplifies to $2n-\log n-2$. Therefore, the additional computations in the generating phase are in $O(n)$.

2.3 Sorting Phase

Similar to *HEAPSORT* we successively swap the top element a_0 with the last element a_m in the array, $n-1 \geq m \geq 2$, and restore the defining *Weak-Heap* conditions in the interval $[0...m-1]$ by calling an operation `MergeForest(m)`:

First of all we traverse the grandchildren of the root. More precisely, we set an index variable x to the value 1 and execute the following loop: `while` $(2x+r_x < m)$ $x \leftarrow 2x+r_x$. Then, in a bottom-up traversal, the *Weak-Heap* conditions are regained by a series of merge operations. This results in a second loop: `while`$(x>0)$ `Merge`$(0,x)$; $x \leftarrow x/2$ with at most $\lceil \log(m+1) \rceil$ key comparisons.

Theorem 2. `MergeForest` *generates a* Weak-Heap *according to its definition.*

Proof. After traversing the grandchildren set of the root, x is the leftmost leaf in the *Weak-Heap*. Therefore, the preconditions to the first `Merge` operation are trivially fulfilled. Hence the root and the subtree at x form a *Weak-Heap*. Since the *Weak-Heap* definition is reflected in all substructures for all grandchildren y of the root we have that y and its right subtree form a *Weak-Heap*. Therefore, we correctly combine the *Weak-Heaps* at position 0 and y and continue in a bottom-up fashion.

2.4 The *WEAK-HEAPSORT* Algorithm

WEAK-HEAPSORT combines the generating and the sorting phase. It invokes `WeakHeapify` and loops on the two operations `swap(0,m)` and `MergeForest(m)`. Since the correctness has already been shown above, we now turn to the time complexity of the algorithm measured in the number of key comparisons.

Theorem 3. *Let* $k=\lceil \log n \rceil$. *The worst case number of key comparisons of* WEAK-HEAPSORT *is bounded by* $nk-2^k+n-1 \leq n\log n+0.086013n$.

Proof. The calls `MergeForest`(i) perform at most $\sum_{i=2}^{n-1}\lceil \log(i+1) \rceil = nk-2^k$ comparisons (and at least $\sum_{i=2}^{n-1}\lceil \log(i+1) \rceil - 1 = nk-2^k-n+2$ comparisons). Together with the $n-1$ comparisons to build the *Weak-Heap* we have $nk-2^k+n-1$ comparisons altogether. Utilizing basic calculus we deduce that for all n there is an x in $[0,1]$ with $nk-2^k+n-1 = n\log n+nx-n2^x+n-1 = n\log n+n(x-2^x+1)-1$ and that the function $f(x)=x-2^x+1$ takes it maximum at $x_0=-\ln\ln 2/\ln 2$ and $f(x_0)=0.086013$. Therefore, the number of key comparisons in *WEAK-HEAPSORT* is less than $n\log n+0.086013n$.

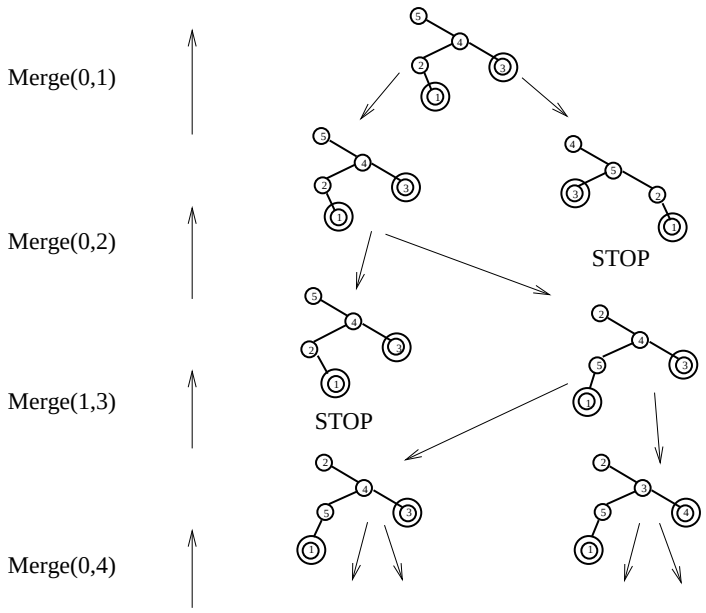

Fig. 2. Backward analysis of *WEAK-HEAPSORT*.

3 The Number of *Weak-Heaps*

Let $W(n)$ be the set of roots of complete subtrees in the *Weak-Heap* of size n.

Theorem 4. *If the input of* WEAK-HEAPSORT *is a random permutation of the elements* $\{1,\ldots,n\}$*, then every possible and feasible* Weak-Heap *occurs with the same probability. Moreover, there are* $n!/2^{|W(n)|}$ *different* Weak-Heaps *represented as a binary tree.*

Instead of a formal proof (cf. [3]) in Fig. 2 we give a simple example illustrating the idea of the backward analysis: For $n = 5$ we find two roots of complete subtrees (these nodes are double-encircled). In the first step swapping the top two elements leads to a dead end, since the generated *Weak-Heap* becomes infeasible. No further operation can move the (deepest) leaf to the leftmost branch as required for a correct input. Swapping 2 and 5 in the second step analogously leads to an infeasible *Weak-Heap*. Only in the following steps (according to roots of complete subtrees) both successor *Weak-Heaps* are feasible.

On the other hand, the assignments to reserve bits uniquely determines which cases in `Merge` have been chosen in the generating phase.

Theorem 5. *There are* $n!$ *different array-embedded* Weak-Heaps.

4 The Best case of *WEAK-HEAPSORT*

This section proves Dutton's conjecture (1992) that an increasing sequence of input elements leads to the minimal number of key comparisons.

For the best case it will be sufficient that in every invocation of *MergeForest* the traversed path P to the leaf node, *special path* for short, terminates at the last position of the array. Subsequently, by exchanging the current root with the element at this position the path is pruned by one element. Fig. 1 depicts an example for this situation.

Therefore, by successively traversing the $2i + r_i$ successors from index 1 onwards we end up at index $n-1$. Hence, r_i has to coincide with the binary encoding $(b_k \dots b_0)_2$ of $n-1$. More precisely, if $r_{\lfloor \frac{n-1}{2^i} \rfloor} = b_{i-1}$ for $i \in \{1, \dots, k\}$, then $n-1$ is the last element of the special path. In case of the input $a_i = i$ for $i \in \{0, \dots, n-1\}$, `WeakHeapify` leads to $r_0 = 0$ and $r_j = 1$ for $j \notin P$. Moreover, for r_j with $j \in P$ we get the binary representation of $n-1$ as required. In other words, $Heap(n)$ defined as

$$
\begin{array}{l}
\tau^1_{Gparent(n-1),n-1} \circ \tau^1_{Gparent(n-2),n-2} \circ \dots \circ \\
\tau^{b_0}_{Gparent(\lfloor \frac{n-1}{2^1} \rfloor),\lfloor \frac{n-1}{2^1} \rfloor} \circ \tau^1_{Gparent(\lfloor \frac{n-1}{2^1} \rfloor - 1),\lfloor \frac{n-1}{2^1} \rfloor - 1} \circ \dots \circ \\
\tau^{b_1}_{Gparent(\lfloor \frac{n-1}{2^2} \rfloor),\lfloor \frac{n-1}{2^2} \rfloor} \circ \tau^1_{Gparent(\lfloor \frac{n-1}{2^2} \rfloor - 1),\lfloor \frac{n-1}{2^2} \rfloor - 1} \circ \dots \circ \\
\vdots \\
\tau^{b_{k-2}}_{Gparent(\lfloor \frac{n-1}{2^{k-1}} \rfloor),\lfloor \frac{n-1}{2^{k-1}} \rfloor} \circ \tau^{b_{k-1}}_{0,1}
\end{array}
$$

correctly determines the transpositions according to the generating phase, where $\tau^a_{i,j}$ is the transposition of i and j if a is odd and the identity, otherwise. As an example consider $Heap(15) = (14\ \ 3)^1(13\ \ 6)^1(12\ \ 1)^1(11\ \ 5)^1$ $(10\ \ 2)^1(9\ \ 4)^1(8\ \ 0)^1(7\ \ 3)^0(6\ \ 1)^1(5\ \ 2)^1(4\ \ 0)^1(3\ \ 1)^1(2\ \ 0)^1(1\ \ 0)^1$.

We now consider the second largest element $n-2$ and assume that $n-2$ has the binary encoding $(c_l \dots c_0)_2$. Further, let $\oplus$ denote the exclusive or operation.

Lemma 6. *Let $a_i = i$ for $i \in \{0, \dots, n-1\}$ be the input for* WEAK-HEAPSORT. *After the generating phase the element $n-2$ will be placed at position $\lfloor \frac{n-1}{2^{i^*}} \rfloor$ with $i^* = max\{i \mid b_{i-1} \oplus c_{i-1} = 1\}$. Moreover, for $j < i \leq i^*$ we have $a[\lfloor \frac{n-1}{2^j} \rfloor] < a[\lfloor \frac{n-1}{2^i} \rfloor]$.*

Proof. If $b_0 = 1$ then $n-1$ is a right child and $n-2$ is a left child. Hence, $Parent(n-2) = Gparent(n-1)$ and $Gparent(n-2) = Gparent(Gparent(n-1))$. Therefore, $n-2$ will be finally located at position $Gparent(n-1) = \lfloor \frac{n-1}{2^1} \rfloor$. Moreover, the key $n-2$ at $\lfloor \frac{n-1}{2^1} \rfloor$ is larger than $\lfloor \frac{n-1}{2^1} \rfloor + 1$ located at $n-1$. Therefore, for $j < i \leq i^*$ we have $a_{\lfloor \frac{n-1}{2^j} \rfloor} < a_{\lfloor \frac{n-1}{2^i} \rfloor}$ as required.

For the other case $b_0 = 0$ we first consider $n-1 \neq 2^k$. The leaf $n-1$ is as long a left child as $n-2$ is a right child. Therefore, $n-2$ will be a left child at $\lfloor \frac{n-2}{2^{i^*-1}} \rfloor$ with $i^* = max\{i \mid b_{i-1} \oplus c_{i-1} = 1\}$. Since $Gparent(Gparent(n-1)) = Gparent(\lfloor \frac{n-1}{2^{i^*}} \rfloor) = Gparent(\lfloor \frac{n-2}{2^{i^*-1}} \rfloor)$, $n-2$ will finally be located at $\lfloor \frac{n-1}{2^{i^*}} \rfloor$.

Now let $n-1=2^k$. In this case $Gparent(n-1)$ is the root. Since for all i we have $Parent(\lfloor\frac{n-2}{2^i}\rfloor) = Gparent(\lfloor\frac{n-2}{2^i}\rfloor)$, the element $n-2$ will eventually reach position $1 = \lfloor\frac{n-1}{2^{i^*}}\rfloor$.

To complete the proof the monotonicity criterion remains to be shown. An element can *escape* from a *Weak-Heap* subtree structure only via the associated root. The position $Gparent(n-1)$ will be occupied by $n-1$ and for all elements on P we know that the key at position $\lfloor\frac{n-1}{2^i}\rfloor$ is equal to $max\{\{\lfloor\frac{n-1}{2^i}\rfloor\} \cup \{k \mid k \in rT(\lfloor\frac{n-1}{2^i}\rfloor)\}\} = \lfloor\frac{n-1}{2}\rfloor + 2^{i-1}$, with $rT(x)$ denoting the right subtree of x. Therefore, for all $j < i < i^*$ we conclude that the key at $\lfloor\frac{n-1}{2^j}\rfloor$ is larger than the key at $\lfloor\frac{n-1}{2^i}\rfloor$. Since $a_{\lfloor\frac{n-1}{2^{i^*}}\rfloor} = n-2$, this condition also holds at $i = i^*$.

Lemma 7. *After the initial swap of position 0 and n-1* `MergeForest` *invokes the following set of transpositions* $CaseB(n) := \tau_{0,\lfloor\frac{n-1}{2^1}\rfloor}^{b_0 \oplus c_0} \circ \ldots \circ \tau_{0,1}^{b_{k-1} \oplus c_{k-1}}$.

Proof. Lemma 6 proves that all swaps of position $\lfloor\frac{n-1}{2^j}\rfloor$ with position 0 with $j < i^*$ are executed, since at the root we always find a smaller element of the currently considered one. We also showed that the maximum element $n-2$ is located at $\lfloor\frac{n-1}{2^{i^*}}\rfloor$. Therefore, no further element can reach the root. This corresponds to the observation that only for $j > i^*$ we have $b_{j-1} \oplus c_{j-1} = 0$.

For the example given above we have $CaseB(15) = (0\ \ 7)^1(0\ \ 3)^1(0\ \ 1)^0$.

The proof of the following two results

Lemma 8. $Heap(n) \circ (0\ \ n-1) = (Gparent(n-1)\ \ n-1) \circ Heap(n)$.

Lemma 9. $Heap(n-1)^{-1} \circ (Gparent(n-1)\ \ n-1) \circ Heap(n) = CaseB(n)^{-1}$.

is technically involved and can be found [3].

Lemma 10. $Heap(n) \circ Swap(0, n-1) \circ CaseB(n) \circ Heap(n-1)^{-1} = id_{\{0,\ldots,n-1\}}$.

Proof. By right and left multiplication with $Heap(n-1)$ and $Heap(n-1)^{-1}$ the equation $Heap(n) \circ Swap(0, n-1) \circ CaseB(n) \circ Heap(n-1)^{-1} = id_{\{0,\ldots,n-1\}}$ can be transformed into $Heap(n-1)^{-1} \circ Heap(n) \circ Swap(0, n-1) \circ CaseB(n) = id_{\{0,\ldots,n-1\}}$. By Lemma 8 this is equivalent to $Heap(n-1)^{-1} \circ (Gparent(n-1)\ \ n-1) \circ Heap(n) \circ CaseB(n) = id_{\{0,\ldots,n-1\}}$. Lemma 9 completes the proof.

Continuing our example with $n = 15$ we infer $n-1 = 14 = (b_3\ \ b_2\ \ b_1\ \ b_0)_2 = (1\ \ 1\ \ 1\ \ 0)_2$ and $n-2 = 13 = (c_3\ \ c_2\ \ c_1\ \ c_0)_2 = (1\ \ 1\ \ 0\ \ 1)_2$. Furthermore, $Heap(14)^{-1} = (1\ \ 0)^1(2\ \ 0)^1(3\ \ 1)^0(4\ \ 0)^1(5\ \ 2)^1(6\ \ 1)^1(7\ \ 3)^1(8\ \ 0)^1(9\ \ 4)^1(10\ \ 2)^1$ $(11\ \ 5)^1(12\ \ 1)^1(13\ \ 6)^1$. Therefore,

$(1\ \ 0)(2\ \ 0)(4\ \ 0)(5\ \ 2)(6\ \ 1)(7\ \ 3)(8\ \ 0)(9\ \ 4)(10\ \ 2)(11\ \ 5)(12\ \ 1)(13\ \ 6)$
$(14\ \ 3)(13\ \ 6)(12\ \ 1)(11\ \ 5)(10\ \ 2)(9\ \ 4)(8\ \ 0)(6\ \ 1)(5\ \ 2)(4\ \ 0)(3\ \ 1)(2\ \ 0)(1\ \ 0)$
$(0\ \ 14)(0\ \ 7)(0\ \ 3) = id_{\{0,\ldots,14\}}$.

Inductively, we get the following result

Theorem 11. *The best case of* WEAK-HEAPSORT *is met given an increasing ordering of the input elements.*

5 The Worst case of *WEAK-HEAPSORT*

The worst case analysis, is based on the best case analysis. The main idea is that the special path misses the best case by one element. Therefore, the `Merge` calls in `MergeForest(m)` will contain the index $m-1$. This determines the assignment of the reverse bits on P: If $n-2 = (b_k \dots b_0)_2$ and if $r_{\lfloor \frac{n-2}{2^i} \rfloor} = b_{i-1}$ for all $i \in \{1, \dots, k\}$ then $n-2$ is the last element of P.

An appropriate example fulfilling this property, is the input $a_i = i+1$ with $i \in \{0, \dots, n-2\}$ and $a_{n-1} = 0$. After termination of `WeakHeapify` we have $r_0 = 0$, $r_j = 1$ for $j \notin P$, $r_{n-1} = 0$, $r_{n-2} = 1$, and $r_{\lfloor \frac{n-2}{2^i} \rfloor} = b_{i-1}$ for $i \in \{1, \dots, k\}$. The transpositions $Heap(n)$ of the *Weak-Heap* generation phase are:

$$
\begin{array}{l}
\tau^{1}_{Gparent(n-2),n-2} \circ \tau^{1}_{Gparent(n-2),n-2} \circ \tau^{1}_{Gparent(n-2),n-2} \circ \dots \circ \\
\tau^{b_0}_{Gparent(\lfloor \frac{n-2}{2^1} \rfloor),\lfloor \frac{n-2}{2^1} \rfloor} \circ \tau^{1}_{Gparent(\lfloor \frac{n-2}{2^1} \rfloor -1),\lfloor \frac{n-2}{2^1} \rfloor -1} \circ \dots \circ \\
\tau^{b_1}_{Gparent(\lfloor \frac{n-2}{2^2} \rfloor),\lfloor \frac{n-2}{2^2} \rfloor} \circ \tau^{1}_{Gparent(\lfloor \frac{n-2}{2^2} \rfloor -1),\lfloor \frac{n-2}{2^2} \rfloor -1} \circ \dots \circ \\
\vdots \\
\tau^{b_{k-2}}_{Gparent(\lfloor \frac{n-2}{2^{k-1}} \rfloor),\lfloor \frac{n-2}{2^{k-1}} \rfloor} \circ \tau^{b_{k-1}}_{0,1}
\end{array}
$$

Unless once per level (when the binary tree rooted at position 1 is complete) we have $\lceil \log(n+1) \rceil$ instead of $\lceil \log(n+1) \rceil - 1$ comparisons. If $n-1$ is set to $n-2$ Lemma 6 remains valid according to the new input. Therefore, we conclude

Lemma 12. *The first invocation of* `MergeForest` *(with the above input) leads to following set of transpositions* $\mathrm{CaseW}(n) = \tau^{1}_{0,n-2} \circ \tau^{b_0 \oplus c_0}_{0,\lfloor \frac{n-2}{2^1} \rfloor} \circ \dots \circ \tau^{b_{k-1} \oplus c_{k-1}}_{0,\lfloor \frac{n-2}{2^k} \rfloor = 1}$.

The following two results are obtained by consulting the best case analysis.

Lemma 13. $Heap(n) \circ (0 \;\; n-1) = (n-2 \;\; n-1) \circ Heap(n)$.

Lemma 14. $CaseW(n) = Swap(0, n-2) \circ CaseB(n-1)$.

Since the definitions of $Heap(n)$ are different in the worst case and best case analysis we invent labels $Heap_b(n)$ for the best case and $Heap_w(n)$ for the worst case, respectively.

Lemma 15.

$$Heap_w(n) \circ Swap(0, n-1) \circ CaseW(n) \circ Heap_w(n-1)^{-1} = (n-2 \;\; n-1).$$

Proof. According to Lemma 13 the stated equation is equivalent to $(n-2 \;\; n-1) \circ Heap_w(n) \circ CaseW(n) \circ Heap_w(n-1)^{-1} = (n-2 \;\; n-1)$.

The observation $Heap_w(n) = Heap_b(n-1)$ and Lemma 14 results in $(n-2 \;\; n-1) \circ Heap_b(n-1) \circ Swap(0, n-2) \circ CaseB(n-1) \circ Heap_b(n-2)^{-1} = (n-2 \;\; n-1)$, which is equivalent to Lemma 10 of the best case.

Inductively, we get

Theorem 16. *The worst case of* WEAK-HEAPSORT *is met with an input of the form* $a_{n-1} < a_i < a_{i+1}$, $i \in \{0, \ldots, n-3\}$.

As an example let $n = 16$, $n-2 = 14 = (b_3\ \ b_2\ \ b_1\ \ b_0)_2 = (1\ \ 1\ \ 1\ \ 0)_2$ and $n-3 = 13 = (c_3\ \ c_2\ \ c_1\ \ c_0)_2 = (1\ \ 1\ \ 0\ \ 1)_2$. Further let $Heap_w(16) = (14\ \ 3)^1(13\ \ 6)^1(12\ \ 1)^1(11\ \ 5)^1(10\ \ 2)^1(9\ \ 4)^1(8\ \ 0)^1(7\ \ 3)^0(6\ \ 1)^1(5\ \ 2)^1(4\ \ 0)^1$ $(3\ \ 1)^1(2\ \ 0)^1(1\ \ 0)^1$, $Swap(0, 16) = (0\ \ 15)^1$, $CaseW(16) = (0\ \ 14)^1(0\ \ 7)^1(0\ \ 3)^1$, and $Heap(15)^{-1} = (1\ \ 0)^1(2\ \ 0)^1(3\ \ 1)^0(4\ \ 0)^1(5\ \ 2)^1(6\ \ 1)^1(7\ \ 3)^1(8\ \ 0)^1(9\ \ 4)^1$ $(10\ \ 2)^1(11\ \ 5)^1(12\ \ 1)^1(13\ \ 6)^1$. Then $Heap_w(16) \circ Swap(0, 16) \circ CaseW(16) \circ Heap(15)^{-1} = (14\ \ 15)$.

6 The Average case of *WEAK-HEAPSORT*

Let $d(n)$ be given such that $n \log n + d(n)n$ is the expected number of comparisons of *WEAK-HEAPSORT*. Then the following experimental data show that $d(n) \in [-0.47, -0.42]$. Moreover $d(n)$ is small for $n \approx 2^k$ and big for $n \approx 1.4 \cdot 2^k$.

n	1000	2000	3000	4000	5000	6000	7000	8000
$d(n)$	-0.462	-0.456	-0.437	-0.456	-0.445	-0.429	-0.436	-0.458
n	9000	10000	11000	12000	13000	14000	15000	16000
$d(n)$	-0.448	-0.437	-0.432	-0.430	-0.436	-0.443	-0.449	-0.458
n	17000	18000	19000	20000	21000	22000	23000	24000
$d(n)$	-0.458	-0.449	-0.443	-0.437	-0.433	-0.431	-0.436	-0.427
n	25000	26000	27000	28000	29000	30000		
$d(n)$	-0.431	-0.437	-0.436	-0.440	-0.440	-0.447		

There was no significant difference between the execution of one trial and the average of 20 trials. The reason is that the variance of the number of comparisons in *WEAK-HEAPSORT* is very small: At $n = 30000$ and 20 trials we achieved a best case 432657 and a worst case of 432816 comparisons.

According to published results of Wegener (1992) and own experiments *WEAK-HEAPSORT* requires approx. $0.81n$ less comparisons than *BOT-TOM-UP-HEAP-SORT* and approx. $0.45n$ less comparisons than *MDR-HEAPSORT*.

7 The *Weak-Heap* Priority-Queue Data Structure

A priority queue provides the following operations on the set of items: *Insert* to enqueue an item and *DeleteMax* to extract the element with the largest key value. To insert an item v in a *Weak-Heap* we start with the last index x of the array a and put v in a_x. Then we climb up the grandparent relation until the *Weak-Heap* definition is fulfilled. Thus, we have the following pseudo-code: `while` $(x \neq 0)$ `and` $(a_{\texttt{Gparent}(x)} < a_x)$ `Swap(Gparent(`x`),`x`)`; $r_x = 1 - r_x$; $x \leftarrow$`Gparent(`x`)`. Since the expected path length of grandparents from a leaf node to a root is approximately half the depth of the tree, we expect at about $\log n/4$ comparisons

in the average case. The argumentation is as follows. The sum of the length of the grandparent relation from all nodes to the root in a weak-heap of size $n = 2^k$ satisfy the following recurrence formula: $S(2^1) = 1$ and $S(2^k) = 2S(2^{k-1}) + 2^{k-1}$ with closed form of $nk/2 + n$ such that the average length is at about $k/2 + 1$.

A double-ended priority queue, *deque* for short, extends the priority queue operation by *DeleteMin* to extract the smallest key values. The transformation of a *Weak-Heap* into its dual in $\lfloor (n-1)/2 \rfloor$ comparisons is performed by the following pseudo-code:

`for` $i = \{size - 1, \ldots, \lfloor (size-1)/2 \rfloor + 1\}$ `Swap(Gparent(`i`),`i`)` followed by
`for` $i = \{\lfloor (size-1)/2 \rfloor, \ldots, 1\}$ `Merge(Gparent(`i`),`i`)`

By successively building the two heaps we have solved the well-known min-max-problem in the optimal number of $n + \lceil n/2 \rceil - 2$ comparisons.

Each operation in a general priority queue can be divided into several compare and exchange steps where only the second one changes the structure. We briefly sketch the implementation. Let M be a Max-*Weak-Heap* and M' be a Min-*Weak-Heap* on a set of n items a and a', respectively. We implicitly define the bijection ϕ by $a_i = a'_{\phi(i)}$. In analogy we might determine ϕ' for M'. The conditions $a_i = a'_{\phi(i)}$ and $a'_i = a_{\phi'(i)}$ are kept as an invariance. Swapping j and k leads to the following operations: Swap a_j and a_k, exchange $\phi(j)$ and $\phi(k)$, set $\phi'(\phi(j))$ to j and set $\phi'(\phi(k))$ to k. We see that the invariance is preserved. A similar result is obtained if a swap-operation on M' is considered.

8 Conclusion

Weak-Heaps are a very fast data structure for sorting in theory and practice. The worst case number of comparisons for sorting an array of size n is bounded by $n \log n + 0.1n$ and empirical studies show that the average case is at $n \log n + d(n)n$ with $d(n) \in [-0.47, -0.42]$. Let $k = \lceil \log n \rceil$. The exact worst case bound for *WEAK-HEAPSORT* is $nk - 2^k + n - k$ and appears if all but the last two elements are ordered whereas the exact best case bound of $nk - 2^k + 1$ is found if all elements are in ascending order. On the other hand the challenging algorithm *BOT-TOM-UP-HEAPSORT* is bounded by $1.5n \log n + O(n)$ in the worst case. Its *MDR-HEAPSORT* variant consumes at most $n \log n + 1.1n$ comparisons. Therefore, the sorting algorithm based on *Weak-Heaps* can be judged to be the fastest *HEAPSORT* variant and to compete fairly well with other algorithms.

References

1. R. D. Dutton. The weak-heap data structure. Technical report, University of Central Florida, Orlando, FL 32816, 1992.
2. R. D. Dutton. Weak-heap sort. *BIT*, 33:372–381, 1993.
3. S. Edelkamp and I. Wegener. On the performance of *WEAK-HEAPSORT*. Technical Report TR99-028, Electronic Colloquium on Computational Complexity, 1999. ISSN 1433-8092, 6th Year.
4. R. Fleischer. A tight lower bound for the worst case of Bottom-Up-Heapsort. *Algorithmica*, 11(2):104–115, 1994.

5. R. W. Floyd. ACM algorithm 245: Treesort 3. *Communications of the ACM*, 7(12):701, 1964.
6. T. N. Hibbard. A empirical study of minimal storage sorting. *Communications of the ACM*, 6(5):206–213, 1963.
7. C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.
8. J. Incerpi and R. Sedgewick. Improved upper bounds on shellsort. *Journal of Computer and System Sciences*, 31:210–224, 1985.
9. T. Jiang, M. Li, and P. Vitányi. Average complexity of shellsort. In *ICALP'99*, volume 1644 of *LNCS*, pages 453–462, 1999.
10. J. Katajainen. The ultimate heapsort. In *Proceedings of the Computing: the 4th Australasian Theory Symposium, Australian Computer Science Communications 20(3)*, pages 87–95, 1998.
11. J. Katajainen, T. Pasanen, and J. Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3(1):27–40, 1996.
12. J. Katajainen and T. A. Pasanen. In-place sorting with fewer moves. *Information Processing Letters*, 70(1):31–37, 1999.
13. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Text and Monographs in Computer Science. Springer-Verlag, 1993.
14. C. J. H. McDiarmid and B. A. Reed. Building heaps fast. *Journal of Algorithms*, 10:352–365, 1989.
15. S. Nilsson. *Radix Sorting & Searching*. PhD thesis, Lund University, 1996.
16. A. Papernov and G. Stasevich. The worst case in shellsort and related algorithms. *Problems Inform. Transmission*, 1(3):63–75, 1965.
17. V. Pratt. *Shellsort and Sorting Networks*. PhD thesis, Stanford University, 1979.
18. K. Reinhardt. Sorting in-place with a worst case complexity of $n \log n - 1.3n + O(\log n)$ comparisons and $\epsilon n \log n + O(1)$ transports. *Lecture Notes in Computer Science*, 650:489–499, 1992.
19. W. Rudin. *Real and Complex Analysis*. McGraw–Hill, 1974.
20. R. Schaffer and R. Sedgewick. The analysis of heapsort. *Journal of Algorithms*, 15(1):76–100, 1993.
21. R. Sedgewick. The analysis of quicksort programs. *Acta Inform.*, 7:327–355, 1977.
22. R. Sedgewick. A new upper bound for shellsort. *Journal of Algorithms*, 2:159–173, 1986.
23. D. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.
24. H. Steinhaus. *One hundred problems in elemantary mathematics (Problems 52,85)*. Pergamon Press, London, 1958.
25. M. H. van Emden. Increasing the efficiency of QUICKSORT. *Communications of the ACM*, 13:563–567, 1970.
26. 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*, 97(1):86–96, 1992.
27. 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.
28. J. W. J. Williams. ACM algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.