An In-Place Sorting with $O(n \log n)$ Comparisons and $O(n)$ Moves

GIANNI FRANCESCHINI

University of Pisa, Pisa, Italy

AND

VILIAM GEFFERT

P. J. Šafárik University, Košice, Slovakia

Abstract. We present the first in-place algorithm for sorting an array of size n that performs, in the worst case, at most $O(n \log n)$ element comparisons and $O(n)$ element transports.

This solves a long-standing open problem, stated explicitly, for example, in Munro and Raman [1992], of whether there exists a sorting algorithm that matches the asymptotic lower bounds on all computational resources simultaneously.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

General Terms: Algorithms

Additional Key Words and Phrases: Sorting in-place

1. *Introduction*

From the very beginnings of computer science, sorting has been one of the most fundamental problems, of great practical and theoretical importance. In virtually every field of computer science, there are problems that have the sorting of a set of objects as a primary step toward solution. (For early history of sorting, see Knuth [1973, Sect. 5.5].) It is well known that a comparison-based algorithm must perform, even in the average case, at least $\log n! \approx n \cdot \log n - n \cdot \log e \approx n \cdot \log n - 1.443n$ comparisons to sort an array consisting of n elements. (All logarithms throughout

The research of G. Franceschini was partially supported by the Italian MIUR.

The research of V. Geffert was supported by the Slovak Grant Agency of Science (VEGA) under contract "Combinatorial Structures and Complexity of Algorithms."

Authors' addresses: G. Franceschini, Department of Informatics, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy, e-mail: francesc@di.unipi.it; V. Geffert, Department of Computer Science, P. J. Šafárik University, Jesenná 5, 04001, Košice, Slovakia, e-mail: geffert@upjs.sk.

this paper are to the base 2, unless otherwise stated explicitly.) By Munro and Raman [1996b], the corresponding lower bound for element moves is $\lfloor 3/2 \cdot n \rfloor$.

Concerning upper bounds for the number of comparisons, already the plain version of mergesort gets closely to the optimum, with fewer than $n \cdot \log n$ comparisons. However, this algorithm needs also an auxiliary array for storing n elements, it is not an *in-place* algorithm. That is, it does not work with only a constant auxiliary storage, in addition to the data stored in the input array. In-place algorithms play an important role, because they maximize the size of data that can be processed in the main memory without an access, during the computation, to a secondary storage device.

The rich history of the comparisons-storage family of sorting algorithms, using $O(n \cdot \log n)$ comparisons and, at the same time, $O(1)$ auxiliary storage, begins with a binary-search version of insertsort. This algorithm uses fewer than $\log n! + n$ comparisons, only a single storage location for putting elements aside, and only $O(1)$ index variables, of $\log n$ bits each, for pointing to the input array. Unfortunately, the algorithm performs $\Omega(n^2)$ element moves, which makes it unacceptably slow, as n increases.

The heapsort [Floyd 1964; Williams 1964] was the first in-place sorting algorithm with a total running time bounded by $O(n \cdot \log n)$ in the worst case. More precisely, it uses less than $2n \cdot \log n$ comparisons with the same $O(1)$ storage requirements as insertsort, but only $n \cdot \log n + O(n)$ moves, if the moves are organized carefully. Since then, many versions of heapsort have been developed; the two most important ones are bottom-up-heapsort [Wegener 1993] and a $\log^*$-variant [Carlsson 1992]. Both these variants use not only the same number of moves as the standard heapsort, but even *exactly the same* sequence of element moves for each input. (See also the procedure "shiftdown" in Schaffer and Sedgewick [1993].) However, they differ in the number of comparisons. Though bottom-up variant uses only $3/2 \cdot n \cdot \log n + O(n)$ comparisons, its upper bound for the average case is even more important; with $n \cdot \log n + O(n)$ comparisons, it is one of the most efficient in-place sorting algorithms. The $\log^*$-variant is slightly less efficient in an average, but it guarantees less than $n \cdot \log n + n \cdot \log^* n$ comparisons in the worst case. For a more detailed analysis, see also Li and Vitányi [1993] and Schaffer and Sedgewick [1993].

Next in-place variants of a k-way mergesort came to the scene [Katajainen et al. 1996; Reinhardt 1992], with at most $n \cdot \log n + O(n)$ comparisons, $O(1)$ auxiliary storage, and $\varepsilon \cdot n \cdot \log n + O(n)$ moves. Instead of merging only 2 blocks, k sorted blocks are merged together at the same time. Here, k denotes an arbitrarily large, but fixed, integer constant, and $\varepsilon > 0$ an arbitrarily small, but fixed, real constant. Except for the first extracted element in each k-tuple of blocks, the smallest element is found with $\log k$ comparisons, if k is a power of two, since the k currently leftmost elements of the respective blocks are organized into a selection tree. Though $\log k$ is more than one comparison required in the standard 2-way merging, the number of merging sweeps across the array comes down to $\lceil \log n / \log k \rceil$, so the number of comparisons is almost unchanged. As an additional bonus, the number of element moves is reduced if, instead of elements, only pointers to elements are swapped in the selection tree. By the use of some other tricks, the algorithm is made in-place and the size of auxiliary storage is reduced to $O(1)$. The early implementation of this algorithm, having such promising upper bounds, turned out to be unacceptably slow. It was observed that operations with indices representing the current state of the selection tree became a bottleneck of the program. Fortunately, the state of a selection tree with a constant number of leaves can be represented implicitly, without swapping indices. This indicates that even by summing comparisons and moves we do not get the whole truth, the arithmetic operations with indices are also important.

The k-way variant has been generalized to a $(\log n/\log\log n)$-way in-place mergesort [Katajainen and Pasanen 1999]. This algorithm uses $n \cdot \log n + O(n \cdot \log\log n)$ comparisons, $O(1)$ auxiliary storage, and only $O(n \cdot \log n/\log\log n)$ element moves. Since k is no longer a constant here, the information about the selection tree is compressed, along with other information, into bits of $(\log n)$-bit index variables by complicated bitwise operations. This increases several cost metrics, including the number of arithmetic operations. Therefore, the algorithm is mainly of theoretical interest, as the first member of the comparisons-storage family breaking the bound $\Omega(n \cdot \log n)$ for the number of moves.

The family of algorithms sorting with $O(n)$ element moves and $O(1)$ auxiliary storage is not so numerous. The first algorithm of this type is selectsort, which is a natural counterpart of insertsort. Carefully implemented, it sorts with at most $2n - 1$ moves, a single location for putting one element aside, and $O(1)$ index variables. Unfortunately, it performs also $\Omega(n^2)$ comparisons.

As shown in Munro and Raman [1996b], $O(n^2)$ comparisons and $O(1)$ indices suffice for reduction of the number of moves to the lower bound $\lfloor 3/2 \cdot n \rfloor$.

Another improvement is a generalized heapsort [Munro and Raman 1992]: This method is based on a heap in which internal nodes have $\lfloor n^{1/k} \rfloor$ children, for a fixed integer k. The corresponding heap tree is thus of constant height, which results in an algorithm sorting with $O(n)$ moves, $O(1)$ storage, and $O(n^{1+\varepsilon})$ comparisons.

Finally, consider the family of algorithms sorting with $O(n \cdot \log n)$ comparisons and $O(n)$ element moves. The first member is a so-called tablesort [Knuth 1973; Munro and Raman 1992]. We use any algorithm with $O(n \cdot \log n)$ comparisons but, instead of elements, we move only indices pointing to the elements. When each element's final position has been determined, we transport all elements to their destinations in linear time. However, this algorithm requires $\Omega(n)$ auxiliary indices.

The storage requirements have been reduced to $O(n^{\varepsilon})$ by a variant of samplesort [Munro and Raman 1992]. The same result can also be obtained by the in-place variant of the k-way mergesort [Katajainen et al. 1996; Katajainen and Pasanen 1999], mentioned above, if $k = \lceil n^{\varepsilon} \rceil$. This reduces the number of merging sweeps to a constant, which results in $O(n \cdot \log n)$ comparisons and $O(n)$ element moves. Such modification is no longer in-place, as it uses $O(n^{\varepsilon})$ auxiliary indices to represent a selection tree. We leave the details to the reader.

No previously published sorting algorithm uses, in the worst case, $O(n \cdot \log n)$ comparisons, $O(n)$ moves, $O(1)$ auxiliary storage, and, at the same time, $O(n \cdot \log n)$ arithmetic operations.

This ultimate goal has only been achieved in the average case [Munro and Raman 1992]. In the worst case, the algorithm uses $\Omega(n^2)$ comparisons but, for a randomly chosen permutation of input elements, the probability of this worst case scenario is negligible.

It was generally conjectured, for many years, that an algorithm matching simultaneously the asymptotic lower bounds on all above computational resources does not exist. For example, in Raman [1991], it was proved that the algorithm with $O(n^{1+\varepsilon})$ comparisons using generalized heaps is optimal among a certain restricted family of in-place sorting algorithms performing $O(n)$ moves. It was hoped that, by generalizing from a restricted computational model to all comparison-based algorithms, we could get a higher trade-off among comparisons, moves, and storage.

1.1. OUR RESULT. The result we present in this article contradicts the above conjectures and closes a long-standing open problem. We shall exhibit the first sorting algorithm that reduces, simultaneously, the number of comparisons, moves, and storage. Our algorithm operates, in the worst case, with at most $2n \cdot \log n + o(n \cdot \log n)$ element comparisons, $(13 + \varepsilon) \cdot n$ element moves, and $O(1)$ auxiliary storage, for each $n \geq 1$. Here, $\varepsilon > 0$ denotes an arbitrarily small, but fixed, real constant. The number of auxiliary arithmetic operations with indices is $O(n \cdot \log n)$. We can slightly reduce the number of moves, to $(12+\varepsilon)\cdot n$, with a modified version that uses $6n \cdot \log n + o(n \cdot \log n)$ comparisons.

The algorithm was born as a union of the ideas contained in two independent technical reports [Geffert 2002; Franceschini 2003]. We believe that, besides the theoretical breakthrough achieved by its analysis, the algorithm can also be of practical interest, because of its simplicity.

1.2. ALGORITHM IN A NUTSHELL. Using an evenly distributed sample $a_1, \ldots, a_f$ of size $\Theta(n/(\log n)^4)$, split the remaining elements into some segments $\sigma_0, \sigma_1, \ldots, \sigma_f$, of length $\Theta((\log n)^4)$ each, so that all elements in the segment σ_k satisfy $a_k \leq a \leq a_{k+1}$. The sorted array is obtained by forming the sequence $\sigma'_0, a_1, \sigma'_1, \ldots, a_f, \sigma'_f$, where σ'_k denotes the segment σ_k in sorted order. To sort σ_k, use a modified heapsort, based on a heap structure in which the internal nodes have $\Theta((\log n)^{4/5})$ sons. This results in a constant number of moves per each element extracted from the heap.

Since an evenly distributed sample is hard to find, the sample grows dynamically, as the computation demands. Initially, the sample is empty, with $f = 0$. That is, all elements are transported, one after another, into the segment σ_0. Whenever, in the course of the computation, some segment σ_k becomes "too large," halve this segment into two segments of equal length. The median element of the original segment is inserted in the proper position of the sample, so that the sample remains sorted. To minimize the number of moves required for insertions in the sample, the sample is sparsely distributed in a block of size $\Theta(n/(\log n)^3)$, in such a way that we do not lose the advantage of a quick binary search over the sample elements. Whenever required, a local density of the sample elements in the block is eliminated by redistributing the sample more evenly, which does not happen "too often." To avoid the corresponding rearrangement of segments, we use also a pointer structure, connecting each sample element a_k with the corresponding segment σ_k. That is, only pointers are moved, the segments stay motionless in a separate workspace.

However, an in-place algorithm does not have an additional buffer array of size $3n$, required for the sample and the segments, nor $P \approx \Theta(n/(\log n)^2)$ bits, required for pointers. The bits are "created" at the very beginning by a modified heapsort. This initial routine collects the smallest and the largest P elements into blocks Π_{L} and Π_{R} placed, respectively, at the beginning and at the end of the array, which leaves an unsorted block $\mathcal{A}'$ in between. Then, the jth bit can be encoded by swapping the jth element in Π_{L} with the jth element in Π_{R}.

To "create" a buffer for sorting the block $\mathcal{A}'$ of length n', select the element $b^{\preceq}$ of rank $\lfloor n'/4 \rfloor$ and partition $\mathcal{A}'$ into blocks $A_{<}$ and $B_{\geq}$, using $b^{\preceq}$ as a pivot. Then sort the block $A_<$, using $B_\geq$ as an empty buffer array. (We can test if a given location contains a buffer element, by a single comparison with $b^{\preceq}$. Before an "active" element is moved to a new location, one buffer element escapes from this location to the current location of the "hole.") After sorting $A_<$ we iterate, focusing on $B_\geq$ as a new block $\mathcal{A}'$. After $O(\log n)$ iterations, we are done.

2. *Sorting with Additional Memory*

Before presenting our in-place algorithm, we concentrate on a simpler task. We are going to sort a given contiguous block A, consisting of m elements, using only $O(m \cdot \log m)$ comparisons and $O(m)$ element moves. As some additional resources, we are given a *buffer memory*, of size at least $3m-1$, that can be used as a temporary workspace, and a *pointer memory*, capable of containing at least $\lfloor 4m/(\log m)^2 \rfloor$ bits.

To let the elements move, we also have a *hole*, that is, one location, the content of which can be modified without destroying any element. An assignment $a_j := a_i$ transports not only one element from the location i to j, but also the hole from j to i. At the very beginning, the hole is in a single extra location.

2.1. BUFFER MEMORY. The buffer memory forms a separate contiguous block B, initially consisting of at least $3m-1$ *buffer elements*. All buffer elements are greater than or equal to a given *buffer separator* $b^{\preceq}$, placed in an extra location, while all elements in A are strictly smaller than $b^{\preceq}$. During the computation, the elements of A and B are mixed up. However, by a single comparison with $b^{\preceq}$, we can test whether any given location contains a buffer element, or an *active element*, a subject of sorting, placed originally in A.

The buffer memory B consists of two parts. First, there is a low level *segment memory*, a sequence of *segments* allocated dynamically from the right end of B and growing to the left, as the computation demands. All allocated segments are of the same fixed length. Second, there is a fixed high level *frame memory*, placed at the left end of B.

2.2. STRUCTURE OF THE SEGMENT MEMORY. All segments are of a fixed length s, where

$$s = \begin{cases} \lceil (\log m)^4 \rceil \\ \lceil (\log m)^4 \rceil + 1 \end{cases} \quad \text{so that } s \text{ is odd.} \tag{1}$$

During the computation, the number of active segments never exceeds $s_\#$, defined by

$$s_\# = \lfloor 2m/s \rfloor \leq 2m/(\log m)^4, \tag{2}$$

and hence the size of workspace reserved for the segment memory is bounded by

$$S = s_\# \cdot s \leq 2m\,. \tag{3}$$

Here we assume that m is "sufficiently large," such that $s \leq m$, and hence $s_\# \geq 2$. We shall later discuss how to handle a block A that is "short."

Initially, all segments are *free*, containing buffer elements only. The algorithm keeps the starting position of the last segment that has been allocated in a global index variable $\vec{s}$. Initially, $\vec{s}$ points to the right end of the buffer memory B. To allocate a new segment, the procedure simply performs the operation $\vec{s} := \vec{s}-s$, and returns the new value of $\vec{s}$ as the starting position of the new segment. Immediately after allocation, some $\lfloor s/2 \rfloor$ active elements (smaller than $b^{\preceq}$) are transported to the first $\lfloor s/2 \rfloor$ positions of the new segment. The corresponding buffer elements are saved in the locations released by the active elements. From this point forward, the segment becomes *active*.

In general, the structure of an active segment is $c_1 \cdots c_h b_{h+1} \cdots b_s$, where $c_1 \cdots c_h$ are active elements stored in the segment, while $b_{h+1} \cdots b_s$ are some buffer elements. The value of h is kept between $\lfloor s/2 \rfloor$ and $s-1$, so that at least one half (roughly) of elements in each active segment is active, and still there is a room for storing one more active element. Neither $c_1 \cdots c_h$ nor $b_{h+1} \cdots b_s$ are sorted. In addition, the algorithm does not keep any information about the boundary h separating active and buffer elements, if the segment is not being manipulated at the present moment. However, since all active elements are strictly smaller than $b^{\preceq}$ and all buffer elements are greater than or equal to $b^{\preceq}$, we can quickly determine the number of active elements in any given segment, using a binary search with $b^{\preceq}$ over the s locations of the segment, which costs only $1 + \lfloor \log s \rfloor \leq O(\log\log m)$ comparisons, by (1).

2.3. STRUCTURE OF THE FRAME MEMORY. The frame memory, placed at the left end of B, consists of $r_{\#}$ so-called *frame blocks*, each of length r, where

$$\begin{aligned} r \; &= \; 1 + \lceil \log(2m/s) \rceil \quad \leq 2 + \log(2m/8) = \log m\,, \\ r_{\#} \; &= \; 2^{r-1} = 2^{\lceil \log(2m/s) \rceil} \; \leq 2 \cdot 2m/s \qquad \leq 4m/(\log m)^4, \end{aligned} \tag{4}$$

using (1) and $m \geq 4$. That is, the frame memory is of total length

$$R = r_{\#} \cdot r \, \leq 4m/(\log m)^3. \tag{5}$$

Using (3) and $m \geq 4$, we get that the total space requirements for the segment and frame memories do not exceed the size of the buffer B, since $R + S \leq 4m/(\log m)^3 + 2m \leq 3m - 1$.

A frame block is either *free*, containing buffer elements only, or it is *active*, containing some active elements followed by some buffer elements. Initially, all frame blocks are free. During the computation, active frame blocks are concentrated in a contiguous left part of the frame, followed by some free frame blocks in the right part. However, there are some important differences from the segment memory structure:

First, the active elements, forming a left part of a frame block, are in sorted order. So are the active frame blocks, forming a left part of the frame memory. More precisely, let $a_1, a_2, \ldots, a_f$ denote the sequence of all active elements stored in the frame memory, obtained by reading active elements from left to right, ignoring buffer elements and frame block boundaries. Then, $a_1, a_2, \ldots, a_f$ is a sorted sequence of elements. Consequently, a subsequence of these, stored in the first (leftmost) positions of active frame blocks, denoted here by $a_{i_1}, a_{i_2}, \ldots, a_{i_g}$, must also be sorted. Here f denotes the total number of active elements in the frame, while g the number of active frame blocks, at the given moment. Similarly, $a_{i_j} a_{i_j+1} a_{i_j+2} \cdots a_{i_{j+1}-1}$, the sequence of active elements stored in the jth frame block, is also sorted.

Second, the number of active elements in an active frame block can range between 1 and $r-1$. That is, we keep room for potential storing of one more active element in each active frame block, but we do not care about a sparse distribution of active elements in the frame. The only restriction follows from the fact that there are no free blocks in between some active blocks.

2.4. RELATIONSHIP BETWEEN THE FRAME AND SEGMENTS. Each active element in the frame memory, that is, each of the elements $a_1, a_2, \ldots, a_f$, has an associated segment $\sigma_1, \sigma_2, \ldots, \sigma_f$ in the segment memory. The segment σ_k, for k ranging between 1 and f, contains some active elements satisfying $a_k \leq a \leq a_{k+1}$, taken from A and stored in the structure so far. The active elements satisfying $a_f \leq a$ are stored in σ_f, similarly, those satisfying $a \leq a_1$ are stored in a special segment σ_0. Note that the segment σ_0 has no "parent" in the sequence $a_1, a_2, \ldots, a_f$, that is, no frame element to be associated with. Chronologically, σ_0 is the first active segment that has been allocated. If $f = 0$, that is, no active elements have been stored in the frame yet, all active elements are transported from A to σ_0.

Note also that (in order to keep the number of active elements in active segments balanced) we do allow some elements equal to a_k be stored both in σ_{k-1} and in σ_k. In general, we may even have $a_k = a_{k+1} = \cdots = a_{k'}$, for some $k < k'$. Then elements equal to a_k may be found in any of the segments $\sigma_{k-1}, \sigma_k, \ldots, \sigma_{k'}$. However, the algorithm tries to store each "new" active element a, coming from A, in the leftmost segment that can be used at the moment, that is, it searches for k satisfying $a_k < a \leq a_{k+1}$.

Recall that we also maintain the invariant that each active segment contains at least $\lfloor s/2 \rfloor$ active elements. Thus, if the frame contains f active elements at the given moment, namely, $a_1, a_2, \ldots, a_f$, for some $f \geq 1$, the total number of active elements, stored both in the frame and the segments $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_f$, is at least $f + (f+1) \cdot \lfloor s/2 \rfloor$. Now, using the fact that s is odd, by (1), we get that this number is at least $f + (f+1) \cdot (s/2 - 1/2) = (f+1) \cdot s/2 + (f/2 - 1/2) \geq (f+1) \cdot s/2$. However, the total number of all active elements is exactly equal to m, which gives $m \geq (f+1) \cdot s/2$, and hence also $f + 1 \leq 2m/s$. But $f+1$, the number of active segments, is an integer number, which gives that $f + 1 \leq \lfloor 2m/s \rfloor$. Therefore, using (2) and (4),

$$\begin{aligned} f + 1 &\leq \lfloor 2m/s \rfloor = s_{\#}, \\ f &\leq \lfloor 2m/s \rfloor \leq 2^{\lceil \log(2m/s) \rceil} = r_{\#}. \end{aligned} \tag{6}$$

(The argument has used the assumption that $f \geq 1$. However, (6) is trivial for $f = 0$, since $r_{\#} \geq s_{\#} \geq 2$, if m is sufficiently large.)

As a consequence, we get that $f + 1$, the number of active segments, does not exceed $s_{\#}$, the capacity of the segment memory. Second, f, the number of active elements in the frame, will never exceed $r_{\#}$, the total number of blocks in the frame, and hence there is enough room to store all active frame elements, even if each active frame block contained only a single element of the sequence $a_1, a_2, \ldots, a_f$.

2.5. STRUCTURE OF THE POINTER MEMORY. The relative order of active frame elements in the sequence $a_1, a_2, \ldots, a_f$ does not correspond to the chronological order, in which the segments $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_f$ are allocated in the segment memory. Therefore, with each element position in the frame, we associate a *pointer* to the starting position of corresponding segment. More precisely, if the frame is viewed as a single contiguous zone of elements $x_1 \ldots x_R$ (ignoring boundaries between the frame blocks), then the corresponding zone of pointers is $\pi_1 \ldots \pi_R$. If, for some ℓ, the element x_ℓ is a buffer element, then $\pi_\ell = 0$, which represents a *NIL* pointer. Conversely, if x_ℓ is an active element belonging to the sequence $a_1, a_2, \ldots, a_f$, then the value of π_ℓ represents the starting position of the segment associated with x_ℓ. (The pointer π_0 to the segment σ_0, having no "parent" in the frame, is stored separately, in a global index variable.)

Since there are at most $s_\#$ segments, all of equal length, a pointer to a segment can be represented by an integer value ranging between 0 and $s_\# = \lfloor 2m/s \rfloor \leq m/2$, using (2). Thus, a single pointer can be represented by a block of p bits, where

$$p = 1 + \lfloor \log s_\# \rfloor \leq \log m \,. \tag{7}$$

The number of pointers is clearly equal to R, the total size of the frame. Therefore,

$$p_\# = R \,.$$

Thus, the pointer memory can be viewed as a contiguous array consisting of $p_\#$ bit blocks, of p bits each, and hence, by (5), its total length is at most

$$P = p_\# \cdot p = R \cdot p \leq \lfloor 4m/(\log m)^2 \rfloor \,, \tag{8}$$

using also the fact that P must be an integer number.

Since an in-place algorithm can store only a limited amount of information in index variables, the pointer memory is actually simulated by two separate contiguous blocks Π_L and Π_R, each containing at least $\lfloor 4m/(\log m)^2 \rfloor$ elements. Initially, Π_L and Π_R are sorted, and the largest (rightmost) element in Π_L is strictly smaller than the smallest (leftmost) element in Π_R. This allows us to encode the value of the jth bit, for any j ranging between 1 and $\lfloor 4m/(\log m)^2 \rfloor$, by swapping the jth element of Π_L with the jth element of Π_R. Testing the value of the jth bit is thus equivalent to comparing the relative order of the corresponding elements in Π_L and Π_R, which costs only a single comparison. Setting a single bit value requires a single comparison and, optionally, a single swap of two elements, that is, 3 element moves. The initial distribution of elements in Π_L and Π_R represents all $\lfloor 4m/(\log m)^2 \rfloor$ bits cleared to zero.

2.6. INSERTING ELEMENTS IN THE STRUCTURE. The procedure sorting the block A works in two phases. In the first phase, the procedure takes, one after another, all m active elements from A and inserts them in the structure described above. The procedure also saves some buffer elements from B, and keeps the structure "balanced." In the second phase, all active elements are transported back to A, this time in sorted order.

For each active element a in A, we find a segment, among $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_f$, where this element should go.

First, by the use of a binary search with the given element a over $a_{i_2}, \ldots, a_{i_g}$, that is, over the leftmost locations in the active frame blocks, find the "proper" frame block for the element a, that is, the index j satisfying $a_{i_j} < a \leq a_{i_{j+1}}$. Note that the element a_{i_1} is excluded from the range of the binary search. If $a \leq a_{i_2}$, the binary search will return $j = 1$, that is, the first frame block. Similarly, for $a_{i_g} < a$, the binary search returns $j = g$, that is, the last frame block. If $g < 2$, we can go directly to the first (and only) active frame block without using any binary search, that is, $j := 1$.

Second, by the use of a binary search with the given element a over the r locations in the jth active frame block, find the "proper" active frame element for the element a, that is, the index k satisfying $a_k < a \leq a_{k+1}$. Note that, since $a_{i_j} < a \leq a_{i_{j+1}}$, the elements a_k and a_{k+1} are between a_{i_j} and $a_{i_{j+1}}$ in the sequence $a_1, a_2, \ldots, a_f$ of all frame elements, not excluding the possibility that $a_{i_j} = a_k$, and/or $a_{k+1} = a_{i_{j+1}}$. Recall that the jth active frame block begins with the active elements $a_{i_j} a_{i_j+1} a_{i_j+2} \ldots a_{i_{j+1}-1}$, followed by some buffer elements, to fill up the room, so that the length of the block is exactly equal to r. These buffer elements are not sorted, however, they are all greater than or equal to $b^{\preceq}$, the smallest buffer element. On the other hand, the element a, being active, is strictly smaller than $b^{\preceq}$. This allows us to use the binary search with the given a in the standard way, which returns the index k satisfying $a_k < a \leq a_{k+1}$. For $a_{i_{j+1}-1} < a$, the binary search returns correctly $k = i_{j+1} - 1$. If $j = 1$, that is, if we are in the first frame block, the binary search may end up with $k = 0$, indicating that $a \leq a_1 = a_{i_1}$.

Third, let the active frame element a_k, satisfying $a_k < a \leq a_{k+1}$, be placed in a position ℓ of the frame memory, that is, $a_k = x_\ell$. (For $k = 0$, we take $\ell := 0$.) Then read the information from π_ℓ in the pointer memory and compute the starting position of the segment σ_k. This segment contains elements ranging between a_k and a_{k+1}. If $k = 0$, that is, the element a should go to σ_0, the starting position of the segment is obtained from a separate global index variable.

Fourth, by the use of a binary search with the buffer separator $b^{\preceq}$ over the s locations in the current segment, find the boundary h dividing the segment into two parts, namely, $c_1 \cdots c_h$, the active elements stored in the segment, and $b_{h+1} \cdots b_s$, some buffer elements, filling up the room.

Fifth, save the buffer element b_{h+1} aside, to the current location of the hole, and, after that, store the given element a in the segment. If $h + 1 < s$, we are ready to insert the next element from A. However, if $h + 1 = s$, the current segment cannot absorb any more elements. Therefore, if the segment has become full, we call a procedure "rebalancing" the structure before trying to store the next element. This procedure will be described later, in Section 2.9.

The above process is repeated until all m active elements have been inserted in the structure.

Initially, the procedure allocates the segment σ_0, and stores the first $s-1$ active elements directly in σ_0, without travelling via the frame. The number of moves for these elements is the same as in the standard case, that is, two moves per each inserted element.

Let us now determine the standard cost of inserting a single element. The binary search looking for a proper frame block inspects a range consisting of $g - 1 < r_\#$ elements, and hence it performs at most $1 + \lfloor \log r_\# \rfloor \leq \log m$ comparisons, by (4). The second binary search, looking for a proper active element within the given frame block, inspects a range of r elements, performing at most $1 + \lfloor \log r \rfloor \leq O(\log \log m)$ comparisons, using (4). Reading the value encoded in the pointer π_ℓ requires $p \leq \log m$ element comparisons, by (7). The binary search with $b^{\preceq}$ over the s locations in the current segment uses $1 + \lfloor \log s \rfloor \leq O(\log \log m)$ comparisons, by (1). Finally, saving one buffer element and transporting the element a to the current segment can be performed with 2 element moves. However, these costs do not include rebalancing. Since m elements are inserted this way, we get:

LEMMA 2.1. *If we exclude the costs of rebalancing, inserting m elements in the structure requires* $2m \cdot \log m + O(m \cdot \log\log m)$ *comparisons and* $2m$ *moves.*

2.7. EXTRACTING IN SORTED ORDER—FRAME LEVEL. In the second phase, the active elements are transported back to A, in sorted order. Let f_{m} denote the maximal value of f, corresponding to the number of active elements in the frame at the moment when the last active element has been stored in the structure. Thus, the frame memory contains the sorted sequence of active elements $a_1, a_2, \ldots, a_{f_{\mathrm{m}}}$, intertwined with some buffer elements, so the total size of the frame is R, consisting of elements $x_1 \cdots x_R$. Then, we have active elements in the segments $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{f_{\mathrm{m}}}$, with σ_k containing active elements that satisfy $a_k \leq a \leq a_{k+1}$. Thus, to produce the sorted order of all active elements, it is sufficient to move, back to A, the sequence $\sigma'_0, a_1, \sigma'_1, a_2, \sigma'_2, \ldots, a_{f_{\mathrm{m}}}, \sigma'_{f_{\mathrm{m}}}$, where σ'_k denotes the block of sorted active elements contained in σ_k.

The procedure begins with moving the block σ'_0 to A. (We shall return to the problem of sorting a given segment σ_k below, in Section 2.8.)

Then, in a loop iterated for $\ell = 1, \ldots, R$, check whether x_ℓ is an active element. This requires only a single comparison, comparing x_ℓ with $b^{\preceq}$. If x_ℓ is a buffer element, it is skipped, we can go to the next element in the frame.

If x_ℓ is an active element, that is, $x_\ell = a_k$, for some k, the procedure saves the leftmost buffer element, not moved yet from the output block A, in the current location of the hole and, after that, moves $x_\ell = a_k$ to A. (The first free position in A, that is, the position of the leftmost buffer element, is kept in a separate global index variable, and incremented each time a new active element is transported back to A.) Then, we read the value encoded in the pointer π_ℓ and compute the starting position of the segment σ_k. After that, we move all active elements contained in σ_k to A, in sorted order, by the procedure presented in Section 2.8.

Before showing how the segment σ_k can be sorted, let us derive computational costs of the above procedure, not including the cost of sorting σ_k. Testing whether x_ℓ is an active element, for $\ell = 1, \ldots, R$, requires $R \leq O(m/(\log m)^3)$ comparisons, by (5). Transporting $x_\ell = a_k$ to A requires only $2f_{\mathrm{m}}$ element moves in total, since only active elements are moved. This gives $2f_{\mathrm{m}} \leq 2r_{\#} \leq O(m/(\log m)^4)$ element moves, by (6) and (4). Reading the values of f_{m} pointers, of length p bits each, can be done with $f_{\mathrm{m}} \cdot p \leq r_{\#} \cdot p \leq O(m/(\log m)^3)$ comparisons, using (6), (4), and (7). Summing up, we have:

LEMMA 2.2. *If we exclude the costs of sorting the segments, extracting in sorted order requires* $O(m/(\log m)^3)$ *comparisons and* $O(m/(\log m)^4)$ *moves.*

2.8. EXTRACTING IN SORTED ORDER—SEGMENT LEVEL. Now we can describe the routine extracting, in sorted order, all active elements contained in the given segment σ_k. Let h_k denote the number of active elements in σ_k. Clearly, $h_k \leq s \leq \lceil(\log m)^4\rceil + 1$, using (1). Initially, the routine determines the value of h_k by the use of a binary search with $b^{\preceq}$ over the s locations of the segment. This costs $1 + \lfloor\log s\rfloor \leq O(\log\log m)$ comparisons.

After that, the routine uses a generalized version of heapsort, which in turn uses a modified heap-like structure, with

$$t = \lceil(\log m)^{4/5}\rceil$$

root nodes (instead of a single root node), and with internal nodes having t sons (instead of two sons). More precisely, we organize $c_1 \cdots c_{h_k}$, the active elements contained in the segment, into the implicit structure with the following properties:

First, the father of the node c_e is the node $c_{e'}$, where $e' = \lfloor (e-1)/t \rfloor$, provided that $e' \geq 1$. If $e' < 1$, then c_e is one of the root nodes. This implies that the heap has t roots, and that the sons of c_e are the nodes $c_{t\cdot e+1}, c_{t\cdot e+2}, \ldots, c_{t\cdot e+t}$. If, for some e and $d < t$, we have $t \cdot e + d = h_k$, the corresponding node c_e has only d sons, instead of t. A leaf is a node c_e without any sons, that is, with $t \cdot e \geq h_k$.

Before passing further, note that the heap does not have more than five levels, since, by travelling to a root from c_{h_k}, we get

$$\begin{aligned} h^{(1)} &= \lfloor (h_k - 1)/t \rfloor \quad < h_k/t\,, \\ h^{(2)} &= \left\lfloor \left(h^{(1)} - 1\right)/t \right\rfloor < h_k/t^2, \\ h^{(3)} &= \left\lfloor \left(h^{(2)} - 1\right)/t \right\rfloor < h_k/t^3, \\ h^{(4)} &= \left\lfloor \left(h^{(3)} - 1\right)/t \right\rfloor < h_k/t^4, \\ h^{(5)} &= \left\lfloor \left(h^{(4)} - 1\right)/t \right\rfloor \leq h^{(4)}/t - 1/t < h_k/t^5 - 1/t \leq s/t^5 - 1/t^5. \end{aligned}$$

If we had $1 \leq h^{(5)}$, then $1 < s/t^5 - 1/t^5$, and hence also $t^5 < s - 1$. Now, using $t = \lceil (\log m)^{4/5} \rceil$ and $s \leq \lceil (\log m)^4 \rceil + 1$, by (1), we would obtain $\lceil (\log m)^{4/5} \rceil^5 < \lceil (\log m)^4 \rceil$, which is a contradiction. To see this, note that, for each real $x > 0$, $\lceil x^{4/5} \rceil^5 \geq x^4$. But $\lceil x^{4/5} \rceil^5$ is an integer number, and hence $\lceil x^{4/5} \rceil^5 \geq \lceil x^4 \rceil$.

The second property of our heap is that, if a node contains an active element, then this element is not greater than any of its sons. Note that we do not care about sons of a node containing a buffer element. (Initially, there are no buffer elements in the heap. However, when some active elements have been extracted, buffer elements will fill up the holes.)

This heap property is established in the standard way: For $e = \lfloor (h_k - 1)/t \rfloor, \ldots, 1$, establish this property in the positions $e, \ldots, h_k$. This only requires to determine whether c_e is not greater than the smallest of its sons and, if necessary, swap the smallest son with c_e. Processing a single node this way costs t comparisons and 3 element moves. After that, the heap property is re-established for the son just swapped in the same way. This may activate a further walk, up to some leaf.

Taking into account that there are $h^{(1)}$ nodes with paths of lengths 1, 2, 3, or 4 (starting from the given node and ending in a leaf), $h^{(2)}$ nodes with paths of lengths 2, 3, or 4, $h^{(3)}$ nodes with paths of lengths 3 or 4, and $h^{(4)}$ nodes with paths of length 4, we get that building the heap costs $t \cdot \sum_{i=1}^{4} h^{(i)} < 2h_k$ comparisons and $3 \cdot \sum_{i=1}^{4} h^{(i)} < 6h_k/(\log m)^{4/5}$ moves.

After building the heap, the routine transports, h_k times, the smallest element from the heap to the output block A. Here the moves are organized as follows. First, save the leftmost buffer element, not moved yet from A, in the current location of the hole. Then, find the smallest element, placed in one of the t roots, and move this element to A. After that, find the smallest element among the t sons of this root, and move this element to the node corresponding to its father. Iterating this process at most five times, we end up with a hole in some leaf. Now, we are done. The hole in the leaf will be filled up by a buffer element in the future, as a side effect. (Usually, in the next iteration, extracting the next smallest element from the heap.)

Thus, unlike in the standard version of heapsort, the size of the heap does not shrink but, rather, some new buffer elements are inserted into the heap structure, filling up the leaf holes. These buffer elements are then handled by the extracting routine in the standard way, as ordinary active elements. Since these elements may travel down, from the leaf level closer to the root level, a node containing a buffer element may have a son containing a smaller buffer element. This will do no harm, however, since each buffer element is strictly greater than any active element, because of the buffer separator $b^{\preceq}$. Thus, no buffer element can be extracted from the heap as the smallest element in the first h_k iterations, when the routine terminates.

Deriving the costs of the above routine is straightforward. The routine repeats h_k iterations, performing each time at most $5(t-1) \leq 5(\log m)^{4/5}$ comparisons and 6 moves, since the heap has at most five levels. This gives $h_k \cdot 5(\log m)^{4/5}$ comparisons and $h_k \cdot 6$ moves.

Now we can sum the costs of sorting the segment σ_k. Determining the value of h_k costs $O(\log\log m)$ comparisons. Building the heap costs at most $2h_k$ comparisons and $6h_k/(\log m)^{4/5}$ moves. Extracting active elements in sorted order costs $h_k \cdot 5(\log m)^{4/5}$ comparisons and $h_k \cdot 6$ moves. Summing up, we get $h_k \cdot O((\log m)^{4/5})$ comparisons and $h_k \cdot (6/(\log m)^{4/5} + 6)$ moves.

To obtain the total cost of sorting all segments $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{f_{\mathrm{m}}}$, we use the fact that $\sum_{k=0}^{f_{\mathrm{m}}} h_k \leq m$, since the number of active elements stored in the segments is bounded by the total number of active elements. Therefore, the sum over all segments results in the following upper bounds:

LEMMA 2.3. *Sorting all segments does not require more than* $O(m \cdot (\log m)^{4/5})$ *comparisons or* $6m + O(m/(\log m)^{4/5})$ *moves.*

Alternatively, we could use the heap structure with parameter $t = \lceil \log m \rceil$. This results in a heap with four levels, instead of five (since $\lceil x \rceil^4 \geq \lceil x^4 \rceil$, for each real $x > 0$). This reduces the leading factor for the number of moves from $6m$ to $5m$. The price we pay is increasing the number of comparisons, from $o(m \cdot \log m)$ to $4m \cdot \log m + O(m)$. The detailed argument is very similar to the proof for $t = \lceil (\log m)^{4/5} \rceil$.

2.9. REBALANCING AT THE SEGMENT LEVEL. This procedure is activated by the routine of Section 2.6, inserting a new active element in the structure, when, for some k, the segment σ_k has become full, having absorbed s active elements.

At the moment of activation, some global index variable is pointing to the starting position of σ_k. The procedure also remembers ℓ, the position of the associated active element $a_k = x_\ell$ in the frame memory, as well as j, the position of the frame block containing the element a_k. We shall call this block the current frame block. (If $\sigma_k = \sigma_0$, that is, $k = 0$, there is no associated element in the frame. Then, $\ell = 0$, but we still have the current frame block, namely, $j = 1$.) The above indices were computed when the latest active element was inserted in the structure.

First, by the use of a binary search with the buffer separator $b^{\preceq}$ over the r locations in the current frame block, find ℓ', the position of the leftmost buffer element in this block. We shall denote this element by $b^{\diamond}$. Recall that we maintain the invariant that each active frame block has a room for one more active element, and therefore it does contain at least one buffer element.

Second, find a median in the segment σ_k, that is, an element $a^{\diamond}$ of rank $\lfloor s/2 \rfloor + 1$. Without loss of efficiency, the selection procedure will position $a^{\diamond}$ at the end of σ_k.

Third, the median $a^{\diamond}$ is inserted in the current frame block, one position to the right of a_k. The active elements lying in between a_k and $b^{\diamond}$, that is, occupying locations $x_{\ell+1} \cdots x_{\ell'-1}$ in the frame memory, are shifted one position to the right.

At the same time, $b^\diamond$ is saved from $x_{\ell'}$ to the location released by $a^\diamond$ at the end of the segment σ_k. (As a special case, if a_k is the rightmost active element in the current frame block, only $b^\diamond$ and $a^\diamond$ are swapped. The same holds when σ_0 is rebalanced for the first time, with $\ell = 0$ and $\ell' = 1$.) Since $a^\diamond$ has been picked from σ_k, it satisfies $a_k \leq a^\diamond \leq a_{k+1}$, and hence the sequence of active elements stored in the frame memory remains sorted.

Fourth, after shifting the active elements in the locations $x_{\ell+1} \cdots x_{\ell'-1}$ one position to the right, we have to shift the corresponding pointers $\pi_{\ell+1} \cdots \pi_{\ell'-1}$ as well, so the active elements remain connected with their segments. To move an integer pointer value from π_e to π_{e+1}, we only have to read the value encoded in π_e and, at the same time, clear π_e, and then to encode this value in π_{e+1}. Such transport of a pointer costs $O(p)$ comparisons and moves.

Fifth, we need to connect a new active element in the frame with a new segment. This concerns the element $a^\diamond$, now placed in $x_{\ell+1}$. Thus, we allocate a new segment $\sigma^\diamond$ and encode its starting position in the pointer $\pi_{\ell+1}$.

Sixth, the full segment σ_k is halved, that is, we place some $\lfloor s/2 \rfloor$ active elements greater than or equal to $a^\diamond$ into the left part of $\sigma^\diamond$ and collect the remaining $\lfloor s/2 \rfloor$ active elements, smaller than or equal to $a^\diamond$, in the left part of the original segment σ_k. Since many elements may be equal to $a^\diamond$, we distribute such elements both to σ_k and $\sigma^\diamond$, so that their active parts are of equal lengths. This also requires to save $\lfloor s/2 \rfloor$ buffer elements, placed originally in $\sigma^\diamond$, to the locations released in σ_k. (We shall give more details below, in Section 2.10.) The outcome of halving is that the active elements in σ_k are split into two segments σ_k and $\sigma^\diamond$, satisfying $a_k \leq a \leq a^\diamond$ and $a^\diamond \leq a \leq a_{k+1}$, respectively.

Seventh, if there is still a room for storing one more active element in the current frame block, the structure has been rebalanced. We are done, ready to take the next element from A. However, if this block has become full, because of $a^\diamond$, the program control jumps to a routine rebalancing the frame level, described later, in Section 2.11.

Let us now derive the computational costs. The binary search, determining the position of the leftmost buffer element in the current frame block, inspects a range of r elements, performing $1 + \lfloor \log r \rfloor \leq O(\log \log m)$ comparisons, by (4). Finding a median, in a segment of length s, requires only $O(s) \leq O((\log m)^4)$ comparisons and $\varepsilon \cdot s \leq \varepsilon \cdot (2 + (\log m)^4)$ element moves, where $\varepsilon > 0$ is an arbitrarily small, but fixed, real constant, by Geffert and Kollár [2001] and (1). Rearranging the elements $a^\diamond$, $b^\diamond$, and $x_{\ell+1} \cdots x_{\ell'-1}$ in their locations can be done with at most $r+2 \leq O(\log m)$ moves, by (4). Shifting the pointers $\pi_{\ell+1} \cdots \pi_{\ell'-1}$ one position to the right costs $O(r \cdot p) \leq O((\log m)^2)$ comparisons, by (4) and (7), together with the same number of moves. Encoding the starting position of a new segment in the pointer $\pi_{\ell+1}$ requires $O(p) \leq O(\log m)$ element moves, by (7). Halving the active elements in σ_k into two segments σ_k and $\sigma^\diamond$ requires only $O(s) \leq O((\log m)^4)$ comparisons and $3/2 \cdot s \leq 3/2 \cdot (2 + (\log m)^4)$ moves, using Lemma 2.5, displayed in Section 2.10 below, and (1).

By summing the bounds above, we get that a single activation of the procedure rebalancing a segment performs $O((\log m)^4)$ comparisons and $(3/2 + \varepsilon) \cdot (\log m)^4$ moves. Taking into account that each activation increases the number of active segments, that we start with one segment, namely, σ_0, and that we end up with $f_{\mathrm{m}} + 1$ segments, we see that the number of activations is bounded by f_{m}. This value is bounded by $f_{\mathrm{m}} \leq s_{\#} \leq 2m/(\log m)^4$, using (6) and (2). This gives:

LEMMA 2.4. *The total cost of keeping the segment level balanced is $O(m)$ comparisons and $(3+\varepsilon)\cdot m$ moves, where $\varepsilon > 0$ is an arbitrarily small, but fixed, real constant.*

2.10. HALVING A SEGMENT. Here, we describe a simple procedure for halving, needed in Section 2.9 above. We are given a segment σ_k of size s, and a median $a^\diamond$, that is, an element of rank $\lfloor s/2 \rfloor + 1$, put aside. We want to place some $\lfloor s/2 \rfloor$ active elements greater than or equal to $a^\diamond$ into the left part of another given segment $\sigma^\diamond$, of size s again, and collect the remaining $\lfloor s/2 \rfloor$ elements smaller than or equal to $a^\diamond$ in the left part of σ_k. The first $\lfloor s/2 \rfloor$ buffer elements of $\sigma^\diamond$ must be saved.

In the first phase, with $s-1$ comparisons and no moves, we count c', the number of elements strictly smaller than $a^\diamond$, in σ_k. This gives us $c = \lfloor s/2 \rfloor - c'$, the number of elements equal to $a^\diamond$ that should remain in σ_k. This number will be required in the second phase, when each element a of σ_k is compared with $a^\diamond$ twice, using "$a < a^\diamond$" and "$a > a^\diamond$." The elements strictly smaller than $a^\diamond$ and the first c elements detected to be equal to $a^\diamond$ will be considered "small," while the remaining equal elements and those strictly greater than $a^\diamond$ will be "large." Each time an element $a = a^\diamond$ will be detected, the counter c will be decreased by one, until it gets to zero. From then on, any "new" element a will be considered "small" if and only if $a < a^\diamond$, and "large" otherwise.

In the second phase, the configurations of the segments are $\sigma_k = A_1\,U\,B_1 b^\diamond$ and $\sigma^\diamond = A_2 B_2$, where A_1 and A_2 denote, respectively, the active elements of σ_k found to be "small" or "large," collected so far, B_1 the buffer elements moved from $\sigma^\diamond$ to σ_k, B_2 the elements of $\sigma^\diamond$ not moved yet, U the elements of σ_k not examined yet, and $b^\diamond$ a single buffer element, filling up the room. A_2 and B_1 are of equal length, not exceeding $\lfloor s/2 \rfloor$. Initially, $\sigma_k = U b^\diamond$, $\sigma^\diamond = B_2$, with A_1, A_2, and B_1 empty. The procedure also remembers the current position of the hole. (After the first iteration, the hole is always in the leftmost location of B_1.)

The second phase proceeds in a loop, as follows. Using at most two comparisons, the rightmost element a of U is determined to be "small" or "large." If a is large, we save the leftmost element from B_2 in the current location of the hole and fill up the new hole in B_2 by a. Thus, A_2 and B_1 have been extended, while U and B_2 have been reduced. If a is small, we scan U from left to right until we find the first element a' that is large. All elements on the left of a' become a part of A_1, without being moved. Since a is a small element placed on the right of the position $\lfloor s/2 \rfloor$, a' must be found before we reach the position $\lfloor s/2 \rfloor + 1$, or else we would have more than $\lfloor s/2 \rfloor$ small elements, which is a contradiction. Now we save the leftmost element from B_2 to the hole, fill up the hole in B_2 by a', and move a to the place released by a'. Then, all necessary boundaries are updated.

This is repeated until we have transported exactly $\lfloor s/2 \rfloor$ active large elements from σ_k to $\sigma^\diamond$. As a consequence, the remaining $\lfloor s/2 \rfloor$ active elements of σ_k, placed on the left of B_1, must be all small, since the rank of $a^\diamond$ is $\lfloor s/2 \rfloor + 1$ and s is odd, by (1).

Clearly, we have used at most $3s$ comparisons in total, and at most three moves per each large element moved from σ_k to $\sigma^\diamond$. This gives:

LEMMA 2.5. *Given a median $a^\diamond$, a segment of size s can be halved with at most $O(s)$ comparisons and $3/2 \cdot s$ moves.*

2.11. REBALANCING AT THE FRAME LEVEL. This routine is activated by the procedure of Section 2.9, rebalancing a segment, when it finds out that, for some j, the jth frame block has become full, having absorbed r active elements. As a side effect, the routine may increase the number of active blocks in the frame. The routine is based on a new variant of the well-known data structure (see Itai et al. [1981] and Willard [1982]), used to maintain a set of elements in sorted order in a contiguous zone of memory.

For the purpose of keeping the frame memory balanced, the frame consisting of $r_{\#}$ frame blocks is viewed, implicitly, as a complete binary tree with $r_{\#} = 2^{r-1}$ leaves, and hence of (edge) height $r-1$. We introduce the following numbering of levels: $i = 0$ for the leaves, 1 for their fathers, and so on, ending by $i = r-1$ for the root. Each node of the tree is associated with a contiguous subarray of the frame blocks, and with a path leading to this node from the root, as follows:

The jth leaf, for any j ranging between 1 and 2^{r-1}, is associated with the jth frame block, that is, with a subarray consisting of $1 = 2^0$ frame blocks, starting from the block position j. The corresponding path from the root to this leaf is represented by the number $\vec{j} = j-1$. It is easy to see that by reading the binary representation of $\vec{j}$ from left to right (with leading zeros so that its length is $r-1$) we get the branching sequence along this path; 0 is interpreted as branching to the left, while 1 as branching to the right.

Given a node v at a level i, associated with a path number $\vec{j}$ and with a subarray of length 2^i blocks, starting from a block position j, the father v' of this node is associated with the path number $\vec{j}\,' = \lfloor \vec{j}/2 \rfloor$, and with the subarray of length 2^{i+1}, starting from the block position $j' = j$, if $\vec{j}$ is even (v is a left son of v'), but from $j' = j-2^i$, if $\vec{j}$ is odd (right son). Thus, the subarray for the father is obtained by concatenation of the two subarrays for its sons, while its path number by cutting off the last bit in the path number for any of its sons.

During the computation, the number of active elements in some local area of the frame may become too large. The purpose of rebalancing a subarray, associated with a node v at a level i, for $i > 0$, is to eliminate such local densities and redistribute active elements more evenly. More precisely, after rebalancing the subarray, the following two conditions will hold:

(i) The number of active elements, in any frame block belonging to the subarray associated with the given node v at the level i, will not exceed the threshold $\tau_i = r-i$.
(ii) The frame memory will not contain any free blocks (without active elements) in between some active blocks.

Note that, if a node v at a level $i > 0$ is an ancestor of the jth leaf, the condition (i) ensures that the jth frame block is not full any longer. Neither is any other block within the subarray. Such redistribution of active elements is possible only if $\alpha(v)$, the total number of active elements in the subarray associated with v, is bounded by $\alpha(v) \leq \tau_i \cdot 2^i$. We say that the node v *overflows*, if $\alpha(v) > \tau_i \cdot 2^i$.

The condition (ii) is required only because of the procedure presented in Section 2.6, transporting active elements from the block A to the structure. Recall that this procedure uses a binary search over the leftmost locations in the active frame blocks, and hence these blocks must form a contiguous zone.

Now we can describe the routine rebalancing the frame.

First, starting from the father of the frame block that is full, climb up and find the lowest ancestor v that does not overflow, with $\alpha(v) \le \tau_i \cdot 2^i$. The formulas for j and $\vec{j}$, presented above, give us a simple tool for computing the boundaries of the associated subarrays, along the path climbing towards the root. To compute the value of $\alpha(v)$, for the given ancestor v at the given level i, simply scan all 2^i frame blocks forming the associated subarray and sum up the numbers of active elements in these blocks, using a binary search with the buffer separator $b^{\preceq}$ over the r locations in each block.

Second, move the $\alpha(v)$ active elements in the associated subarray of v to the last $\alpha(v)$ locations. That is, processing all $2^i \cdot r$ locations in the subarray from the right to left, collect all elements smaller than $b^{\preceq}$ to the right end. Before moving an active element from x_e to $x_{e'}$, for some $e < e'$, the buffer element in the target position $x_{e'}$ is saved to the current location of the hole. Then move the associated pointer in the corresponding positions of the pointer memory, from π_e to $\pi_{e'}$, by reading and clearing the bit value encoded in π_e and encoding this value in $\pi_{e'}$.

Third, redistribute the $\alpha(v)$ active elements back, this time more evenly in the 2^i frame blocks of the subarray, moving also the pointers in the corresponding positions, as follows. Let $\alpha_{\text{D}} = \lfloor \alpha(v)/2^i \rfloor$ and $\alpha_{\text{M}} = \alpha(v) \bmod 2^i$. Then, put $\alpha_{\text{D}}+1$ active elements in each of the first α_{M} blocks, and α_{D} active elements in each of the remaining $2^i - \alpha_{\text{M}}$ blocks. In each block, the active elements are concentrated in its left part.

Fourth, as a side effect of redistribution, the size of the active part in the frame memory may have been increased. This requires to update the value of g, the number of active frame blocks, kept in a separate global index variable. Let g' be the block position of the rightmost frame block in the subarray of v. Then, let $g := \max\{g, g'\}$.

It should be pointed out that, for each leaf, the desired ancestor v without overflow does exist. Using (6), (4), and $\tau_i = r - i$, for the level $i = r-1$, that is, for v being the root node, we get $\alpha(v) = f \le r_{\#} = 1 \cdot 2^{r-1} = \tau_{r-1} \cdot 2^{r-1}$, and hence at least the root node does not overflow. Therefore, in the first step, the loop climbing up towards the root must halt correctly.

Further, the redistribution of active elements, presented in the third step, is correct, since $(\alpha_{\text{D}} + 1) \cdot \alpha_{\text{M}} + \alpha_{\text{D}} \cdot (2^i - \alpha_{\text{M}}) = \alpha(v)$. It is easy to see that the redistribution satisfies the condition (i) above, using the fact that the node v does not overflow, and hence $\alpha(v) \le \tau_i \cdot 2^i$. There are two cases to consider: For $\alpha(v) \le \tau_i \cdot 2^i - 1$, we have $\alpha_{\text{D}} + 1 \le \lfloor (\tau_i \cdot 2^i - 1)/2^i \rfloor + 1 \le (\tau_i - 1) + 1 = \tau_i$, since τ_i is an integer. If $\alpha(v) = \tau_i \cdot 2^i$, we get $\alpha_{\text{M}} = 0$, and hence all 2^i blocks are "remaining," with only α_{D} active elements in each. But here $\alpha_{\text{D}} = \lfloor (\tau_i \cdot 2^i)/2^i \rfloor = \lfloor \tau_i \rfloor = \tau_i$.

It is also easy to see that the redistribution satisfies the condition (ii). Since v is the *lowest* ancestor that does not overflow, along the path from the full frame block towards the root, it must have a son that does overflow, with at least $\tau_{i-1} \cdot 2^{i-1}$ active elements in its subarray. (As a special case, for $i = 1$, we get $\tau_0 \cdot 2^0 = (r-0) \cdot 1 = r$ active elements in the jth frame block that is full.) The subarray of the son is a part of the subarray associated with v, and hence $\alpha(v) \ge \tau_{i-1} \cdot 2^{i-1} \ge 2 \cdot 2^{i-1}$, using the fact that $i - 1 \le r - 2$. But then $\alpha_{\text{D}} = \lfloor \alpha(v)/2^i \rfloor \ge 1$. This implies that each frame block in the subarray associated with v contains at least one active element after redistribution, and hence the zone of active frame blocks will remain contiguous.

Consider now the cost of a single activation of the above routine, rebalancing a subarray for a node v at a level i. Looking for the lowest ancestor without overflow requires to count the numbers of active elements in the associated subarrays along a path climbing up from a father of a leaf, for levels $e = 1, \dots, i$. In the eth level, 2^e blocks are examined, by a binary search over the r locations of the block. By (4), this gives $\sum_{e=1}^{i} 2^e \cdot (1 + \lfloor \log r \rfloor) \leq 2^i \cdot O(\log \log m)$ comparisons. The cost of the second step, collecting $\alpha(v)$ active elements to the right end, is $2^i \cdot r$ comparisons (one comparison with $b^{\preceq}$ for each location in the subarray), plus $\alpha(v) \cdot 2 + 1$ moves (two moves per each collected element). However, with each collected element, the corresponding pointer must also be transported, which gives additional $\alpha(v) \cdot O(p)$ comparisons and moves. Using $\alpha(v) \leq \tau_i \cdot 2^i \leq r \cdot 2^i$, together with (4) and (7), the cost of the second step can be bounded by $2^i \cdot O(r \cdot p) \leq 2^i \cdot O((\log m)^2)$ comparisons and moves. The same computational resources are sufficient in the third step, redistributing the same number of active elements back, but more evenly, together with their pointers. Again, this gives $\alpha(v) \cdot O(p)$ comparisons and moves, which can be bounded by $2^i \cdot O((\log m)^2)$. Finally, the fourth step does not require any element comparisons or moves, it just updates one index variable, in $O(1)$ time.

Summing up, the cost of a single activation is $2^i \cdot O((\log m)^2)$ comparisons and moves, for each node v at the fixed level $i > 0$. To get the total cost, we must take into account how frequently such rebalancing is activated.

When a rebalancing is activated, v must have a son with at least $\tau_{i-1} \cdot 2^{i-1}$ active elements, since v is the lowest ancestor that does not overflow, along some path climbing up. Now, trace back the history of computation, to the moment when the entire subarray associated with v was a subject of redistribution for the last time. This way we get a node v', either an ancestor of v or v itself, at a level $i' \geq i$, with the associated subarray containing the entire subarray for v. After the redistribution for v', both sons of v contained at most $\tau_{i'} \cdot 2^{i-1} \leq \tau_i \cdot 2^{i-1}$ active elements. Thus, in the meantime, the number of active elements in one of the sons of v has been increased by at least $\tau_{i-1} \cdot 2^{i-1} - \tau_i \cdot 2^{i-1} = 2^{i-1}$. Since other redistributions, taking place between the moments of rebalancing v' and v, could not "import" any active elements to the subarray of v from any other parts of the frame, the 2^{i-1} additional active elements must have been *inserted* here. (See the procedure of Section 2.9, third step). Thus, there have to be at least 2^{i-1} insertions in the associated subarray between any two redistributions for v. Note that, for the fixed level i, subarrays associated with different nodes v do not overlap. Thus, we can charge the cost of each activation, for the given node v, to the 2^{i-1} insertions preceding this activation in the given subarray, without charging the same insertion more than once. This gives $2^i \cdot O((\log m)^2)/2^{i-1} \leq O((\log m)^2)$ comparisons and moves, per a single insertion of an active element in the frame memory. Since, in the whole computation, there were only $f_{\mathrm{m}} \leq r_{\#} \leq O(m/(\log m)^4)$ insertions, by (6) and (4), we get the cost $O(m/(\log m)^2)$ comparisons and moves, for rebalancing of all nodes at the fixed level i. By summing over all levels, using $i \leq r - 1 \leq \log m$, by (4), we get the total cost:

LEMMA 2.6. *The total cost of keeping the frame memory balanced is $O(m/\log m)$ comparisons, together with the same number of moves.*

2.12. SUMMARY. By summing the bounds presented in Lemmas. 2.1–2.4 and 2.6 above, we get:

THEOREM 2.7. *The cost of sorting the given block A of size m is $2m \cdot \log m + O(m \cdot (\log m)^{4/5})$ comparisons and $(11+\varepsilon) \cdot m$ moves, where $\varepsilon > 0$ is an arbitrarily small, but fixed, real constant, provided we can use additional buffer and pointer memories, of respective sizes $3m-1$ and $\lfloor 4m/(\log m)^2 \rfloor$.*

The algorithm presented above assumes that m is "sufficiently large," so that s, defined by (1), satisfies $s \leq m$. This presupposition holds for each $m > 2^{16} = 65536$. Shorter blocks are handled in a different way, by the procedure described later, in Section 3.3. The bounds presented by Theorem 2.7 for the number of comparisons and moves will remain valid.

3. *In-Place Sorting*

Now we can present an in-place algorithm sorting the given array $\mathcal{A}$ consisting of n elements. If $n \leq 2^{16}$, the array is sorted directly, by the procedure of Section 3.3, described later. In the general case, for $n > 2^{16}$, the task of the main program is to provide sufficiently large pointer and buffer memories for the procedure presented in Section 2.

3.1. BUILDING A POINTER MEMORY. The size of the largest block ever sorted by the procedure of Section 2 will not exceed $m = n/4$. Using (8) and the fact that the function $4x/(\log x)^2$ is monotone increasing for $x \geq 8$, we see that the size of the pointer memory can be bounded by $P = \lfloor 4(n/4)/(\log(n/4))^2 \rfloor = \lfloor n/(\log(n/4))^2 \rfloor$. This will suffice for all sorted blocks.

The pointer memory is built by collecting two contiguous blocks Π_{L} and Π_{R}. The block Π_{L}, placed at the left end of $\mathcal{A}$, will contain the smallest P elements of the array $\mathcal{A}$, while Π_{R}, placed at the right end, the largest P elements.

The block Π_{R} is created first, by the use of the heapsort with t root nodes and internal nodes having t sons. The detailed topology of edges connecting nodes in this kind of heap has been presented in Section 2.8, devoted to extracting sorted elements at the segment level.

However, there are some substantial differences from the generalized heapsort of Section 2.8. This time the branching degree is $t = \lceil \log n \rceil$. Therefore, the heap has $q \leq 1 + \lfloor \log_t n \rfloor \leq O(\log n / \log\log n)$ levels. Here we keep large elements at the root level, instead of small elements. That is, no node contains an element smaller than any of its sons. Unlike in Section 2.8, no buffer elements are used here to fill up the holes, the heap structure shrinks in the standard way, when the largest element is extracted.

The initial building of the heap structure is standard, and agrees with the heap building in Section 2.8. It is easy to see that, for a heap with n elements, branching degree equal to t, and q levels, the cost of the heap initialization can be bounded by $t \cdot \sum_{i=1}^{q-1} n/t^i < n \cdot t/(t-1) \leq O(n)$ comparisons and $3 \cdot \sum_{i=1}^{q-1} n/t^i < n \cdot 3/(t-1) \leq O(n/\log n)$ moves, using $t \geq \log n$.

After building the heap, the routine extracts, P times, the largest element from the heap in the standard way. That is, when the largest element is extracted, it replaces the element in the rightmost leaf, which in turn is inserted into the "proper" position along the so-called special path, starting from the position of the largest root (just being extracted) and branching always to the largest son.

The costs of the above routine are straightforward. The trajectory of the special path can be localized with $q \cdot (t-1)$ comparisons, and the new position for the element in the rightmost leaf can be found by a binary search along this trajectory with $1 + \lfloor \log q \rfloor$ comparisons. Summing up, an extraction of the largest element can be done with $q \cdot (t-1) + (1 + \lfloor \log q \rfloor)$ comparisons, together with $q + 2$ moves. Using $t \leq O(\log n)$ and $q \leq O(\log n / \log \log n)$, we get, per a single extraction, at most $O((\log n)^2 / \log \log n)$ comparisons, together with $O(\log n / \log \log n)$ moves.

If we let the above procedure run till the end, it would sort the entire array $\mathcal{A}$ in time $O(n \cdot (\log n)^2 / \log \log n)$. However, the execution is aborted as soon as the largest P elements are collected. Since $P \leq O(n/(\log n)^2)$, the cost of building the heap becomes dominant, and hence the block Π_{R} is created with $O(n)$ comparisons and $O(n/\log n)$ moves.

After Π_{R}, the block Π_{L} is created in the same way, with the same computational needs of comparisons and moves. Instead of large elements, here we collect the smallest P elements. In addition, since Π_{L} should be created at the left end of $\mathcal{A}$, all indices are manipulated in a mirrorlike way, seeing the first position to the left of Π_{R} as the beginning of the array.

LEMMA 3.1. *Building the pointer memory requires $O(n/\log n)$ moves and $O(n)$ comparisons.*

Now the configuration of the array $\mathcal{A}$ has changed to $\Pi_{\mathrm{L}}\mathcal{A}'\Pi_{\mathrm{R}}$, where $\mathcal{A}'$ denotes the remaining elements, to be sorted. Before proceeding further, the algorithm verifies, with a single comparison, whether the largest (rightmost) element in Π_{L} is strictly smaller than the smallest (leftmost) element in Π_{R}.

If this is not the case, all elements in $\mathcal{A}'$ must be equal to these two elements. Therefore, the algorithm terminates, the entire array $\mathcal{A}$ has already been sorted.

Conversely, if Π_{L} and Π_{R} pass the test above, they can be used to imitate a pointer memory consisting of P bits.

3.2. PARTITION-BASED SORTING. When the blocks Π_{L} and Π_{R} have been created, the zone $\mathcal{A}'$ is kept in the form $\mathcal{A}_{\mathrm{S}}\mathcal{A}_{\mathrm{U}}$, where $\mathcal{A}_{\mathrm{S}}$ and $\mathcal{A}_{\mathrm{U}}$ represent the sorted and unsorted parts of $\mathcal{A}'$, respectively. Each element in $\mathcal{A}_{\mathrm{S}}$ is strictly smaller than the smallest element of $\mathcal{A}_{\mathrm{U}}$. The routine described here is a partition-based loop. In the course of the ith iteration, the length of $\mathcal{A}_{\mathrm{U}}$ is n_i, with $n_i < n_{i-1}$. Initially, for $i = 0$, $\mathcal{A}_{\mathrm{S}}$ is empty, $\mathcal{A}_{\mathrm{U}} = \mathcal{A}'$, and $n_0 = n - 2P < n$. The loop proceeds as follows:

First, find $b^{\preceq}$, an element of rank $\lceil n_i/4 \rceil$ in $\mathcal{A}_{\mathrm{U}}$. The selection procedure places this element at the right end of $\mathcal{A}_{\mathrm{U}}$, so the configuration of $\mathcal{A}'$ changes to $\mathcal{A}_{\mathrm{S}}\mathcal{A}'_{\mathrm{U}}b^{\preceq}$. Here $\mathcal{A}'_{\mathrm{U}}$ denotes a mix of elements in $\mathcal{A}_{\mathrm{U}}$, of length $n_i - 1$.

Second, $\mathcal{A}'_{\mathrm{U}}$ is partitioned into two blocks $A_{<}$ and $B_{\geq}$ consisting, respectively, of elements strictly smaller than $b^{\preceq}$ and of those greater than or equal to $b^{\preceq}$. The configuration of the array thus changes to $\mathcal{A}_{\mathrm{S}}A_{<}B_{\geq}b^{\preceq}$. The respective lengths of $A_{<}$ and $B_{\geq}$ will be denoted here by $n_{i,<}$ and $n_{i,\geq}$. Note that, even for a large block $\mathcal{A}_{\mathrm{U}}$, we may obtain a very short block $A_{<}$, since many elements may be equal to $b^{\preceq}$. In fact, the block $A_{<}$ may even be empty, of length $n_{i,<} = 0$.

Third, sort the block $A_{<}$ by the procedure described in Section 2, using some initial segments of Π_{L} and Π_{R} as a pointer memory and of $B_{\geq}$ as a buffer memory, with $b^{\preceq}$ as a buffer separator. This is possible, since $b^{\preceq}$ has been selected as an element of rank $\lceil n_i/4 \rceil$, and hence $n_{i,<} \leq \lceil n_i/4 \rceil - 1 \leq n_i/4$, with $n_{i,<} + n_{i,\geq} + 1 = n_i$. But the required size of buffer is only $3n_{i,<} - 1 \leq 3/4 \cdot n_i - 1 = n_i - 1 - n_i/4 \leq n_i - 1 - n_{i,<} = n_{i,\geq}$. Therefore, the block $B_{\geq}$ of length $n_{i,\geq}$ is sufficiently long. Similarly, the required number of bits for pointers is $\lfloor 4n_{i,<}/(\log n_{i,<})^2\rfloor \le \lfloor 4(n/4)/(\log(n/4))^2\rfloor = P$, and hence the pointer memory is also sufficiently large. (If $n_{i,<} \le 2^{16}$, $A_<$ is sorted as a short block.)

Fourth, restore the sorted order in Π_L and Π_R, by clearing all bits of the pointer memory to zero. Among others, this is required because the procedure of Section 2 will also be used in subsequent iterations, when it assumes that all bits are initially cleared.

Fifth, after sorting $A_<$, the configuration of $\mathcal{A}'$ is $\mathcal{A}_S A_{<,S} B'_{\ge} b^{\preceq}$, where $A_{<,S}$ denotes the sorted version of the block $A_<$ and $B'_{\ge}$ a mixed up version of $B_{\ge}$. Now put the first element in $B'_{\ge}$ aside and move $b^{\preceq}$ to the first position after $A_{<,S}$. After that, collect all elements smaller than or equal to $b^{\preceq}$ to the left part of $B'_{\ge}$, processing also the element put aside. Since $B_{\ge}$ did not contain elements strictly smaller than $b^{\preceq}$, this actually partitions $B'_{\ge}$ into two blocks $A_=$ and $B_>$ consisting, respectively, of elements equal to $b^{\preceq}$ and of those strictly greater than $b^{\preceq}$, of respective lengths $n_{i,=}$ and $n_{i,>}$. Clearly, $n_{i,=}+n_{i,>} = n_{i,\ge}$. The configuration has changed to $\mathcal{A}_S A_{<,S} b^{\preceq} A_= B_>$.

Sixth, observe that $\mathcal{A}_S A_{<,S} b^{\preceq} A_=$ and $B_>$ can be viewed as "new" variants of blocks $\mathcal{A}_S$ and $\mathcal{A}_U$. Thus, we can start a new iteration, with $B_>$ as a new block $\mathcal{A}_U$, of length $n_{i+1} = n_{i,>}$. The above process is iterated until the length of unsorted part drops to 2^{16}, or below. This residue is then sorted as a short block, without using a buffer or pointers, which will be described later, in Section 3.3.

Now we can derive computational costs. First, recall that $b^{\preceq}$ has been selected as an element of rank $\lceil n_i/4\rceil$, and hence $n_{i+1} = n_{i,>} \le n_i - \lceil n_i/4\rceil \le 3/4\cdot n_i$. Taking into account that $n_0 \le n$, we get $n_i \le (3/4)^i \cdot n$, for each $i \ge 0$. This gives that

$$\begin{aligned}\sum_{i=0}^{\mathcal{I}-1} n_i &\le 4n\,,\\ \mathcal{I} &\le O(\log n)\,,\end{aligned} \tag{9}$$

where $\mathcal{I}$ denotes the number of iterations. Second, it is easy to see that

$$\sum_{i=0}^{\mathcal{I}-1}(n_{i,<} + 1 + n_{i,=}) + n_{\mathcal{I}} \le n\,, \tag{10}$$

since, in different iterations, the final locations occupied by $A_{<,S}$, $b^{\preceq}$, and $A_=$, do not overlap. Here $n_{\mathcal{I}}$ denotes the length of the residual short block.

Let us now present the costs for the ith iteration. Selection of $b^{\preceq}$, an element of the given rank in a block of length n_i, costs $O(n_i)$ comparisons and $\varepsilon \cdot n_i$ moves, by Geffert and Kollár [2001]. Partitioning of $\mathcal{A}'_U$ into blocks $A_<$ and $B_{\ge}$ can be done with n_i comparisons and $2n_{i,<} + 1$ moves, since the length of $\mathcal{A}'_U$ is $n_i - 1$, and the number of collected elements, strictly smaller than $b^{\preceq}$, is $n_{i,<}$. The cost of sorting the block $A_<$ is bounded by $2n_{i,<} \cdot \log n_{i,<} + O(n_{i,<} \cdot (\log n_{i,<})^{4/5}) \le 2n_{i,<} \cdot \log n + O(n_{i,<} \cdot (\log n)^{4/5})$ comparisons and $(11+\varepsilon)\cdot n_{i,<}$ moves, by Theorem 2.7. Sorting of the block $A_<$ is followed by restoring the sorted order in Π_L and Π_R, by clearing all bits, which costs $O(P) \le O(n/(\log n)^2)$ comparisons, together with the same number of moves. Positioning $b^{\preceq}$ to the right of $A_{<,S}$ requires only 2 element moves. Finally, the ith iteration is concluded by partitioning $B'_{\ge}$ into blocks $A_=$ and $B_>$, with at most $n_{i,\ge} \le n_i$ comparisons and $2n_{i,=} + 1$ moves, since the length of $B'_{\succeq}$ is $n_{i,\geq}$, and the number of collected elements, equal to $b^{\preceq}$, is $n_{i,=}$. The cost of sorting the residual short block does not exceed the bounds for the standard case; $2n_{\mathcal{I}} \cdot \log n_{\mathcal{I}} + 6.25n_{\mathcal{I}} \leq 2n_{\mathcal{I}} \cdot \log n + O(n_{\mathcal{I}} \cdot (\log n)^{4/5})$ comparisons and $9.75n_{\mathcal{I}} \leq (11+\varepsilon) \cdot n_{\mathcal{I}}$ moves. (See Section 3.3 below.)

Now we can sum the above costs over all iterations, using (9) and (10). For the number of comparisons, this gives

$$\begin{aligned} C(n) \;\leq\; & \sum_{i=0}^{\mathcal{I}-1} n_i \cdot O(1) + \sum_{i=0}^{\mathcal{I}-1} n_{i,<} \cdot \big(2\log n + O\big((\log n)^{4/5}\big)\big) \\ & + \sum_{i=0}^{\mathcal{I}-1} O\big(n/(\log n)^2\big) + n_{\mathcal{I}} \cdot \big(2\log n + O\big((\log n)^{4/5}\big)\big) \\ \leq\; & O(n) + \left(\sum_{i=0}^{\mathcal{I}-1} (n_{i,<} + 1 + n_{i,=}) + n_{\mathcal{I}}\right) \cdot \big(2\log n + O\big((\log n)^{4/5}\big)\big) \\ & + O(n/\log n) \leq O(n) + n \cdot \big(2\log n + O\big((\log n)^{4/5}\big)\big) + O(n/\log n) \\ \leq\; & 2n \cdot \log n + O\big(n \cdot (\log n)^{4/5}\big)\,. \end{aligned}$$

For the number of moves, we get

$$\begin{aligned} M(n) \;\leq\; & \sum_{i=0}^{\mathcal{I}-1} \varepsilon \cdot n_i + \sum_{i=0}^{\mathcal{I}-1} (13+\varepsilon) \cdot n_{i,<} + \sum_{i=0}^{\mathcal{I}-1} 2n_{i,=} + \sum_{i=0}^{\mathcal{I}-1} O\big(n/(\log n)^2\big) \\ & + (11+\varepsilon) \cdot n_{\mathcal{I}} \\ \leq\; & \varepsilon \cdot n + \left(\sum_{i=0}^{\mathcal{I}-1} (n_{i,<} + 1 + n_{i,=}) + n_{\mathcal{I}}\right) \cdot (13+\varepsilon) + O(n/\log n) \\ \leq\; & \varepsilon \cdot n + n \cdot (13+\varepsilon) + O(n/\log n) \\ \leq\; & (13+\varepsilon) \cdot n\,, \end{aligned}$$

where $\varepsilon > 0$ is an arbitrarily small, but fixed, real constant. The above analysis did not include the costs of the initial building of pointer memory. However, by Lemma 3.1, this can be done with only $O(n)$ comparisons and $O(n/\log n)$ moves, and hence the bounds displayed above represent the total computational costs of the algorithm.

THEOREM 3.2. *The given array, consisting of n elements, can be sorted in-place by performing at most* $2n \cdot \log n + o(n \cdot \log n)$ *comparisons and* $(13 + \varepsilon) \cdot n$ *element moves, where* $\varepsilon > 0$ *denotes an arbitrarily small, but fixed, real constant. The number of auxiliary arithmetic operations with indices is bounded by* $O(n \cdot \log n)$.

3.3. HANDLING SHORT BLOCKS. The algorithm presented above needs a procedure capable of sorting blocks of small lengths, namely, with $m \leq 2^{16} = 65536$. This is required, among others, to sort blocks $A_{<}$ that are short. We could sweep the problem under the rug by saying that "short" blocks can, "somehow," be sorted with $O(1)$ comparisons and moves, since they are of constant lengths. However, the upper bounds presented by Theorem 2.7 in Section 2.12 require some more details, especially for $(11 + \varepsilon) \cdot m$, the number of moves. Last but not least, these lengths are important in practice.

One of the possible simple solutions is to use our version of heapsort, with 5 roots and internal nodes having 5 sons. Using the analysis presented in Section 3.1, devoted to building a pointer memory, for $t = 5$, $m \leq 2^{16}$, and hence for at most $q \leq 1 + \lfloor \log_t m \rfloor \leq 7$ levels, one can easily verify that we shall never use more than $2m \cdot \log m + 6.25m$ comparisons or $9.75m$ moves. (These bounds are not tight, we leave further improvement to the reader.)

3.4. AN ALTERNATIVE SOLUTION. As pointed out at the end of Section 2.8, devoted to extracting sorted elements from segments, we could use a heap structure with four levels, instead of five, in a segment. This slightly reduces the number of moves, but increases the number of comparisons. The detailed argument parallels the proof of Theorem 3.2, and hence it is left to the reader.

COROLLARY 3.3. *The given array, consisting of n elements, can be sorted in-place by performing at most* $6n \cdot \log n + o(n \cdot \log n)$ *comparisons and* $(12 + \varepsilon) \cdot n$ *element moves, where* $\varepsilon > 0$ *denotes an arbitrarily small, but fixed, real constant.*

4. *Concluding Remarks*

We have described the first in-place sorting algorithm performing $O(n \cdot \log n)$ comparisons and $O(n)$ element moves in the worst case, which closes a long-standing open problem.

However, the algorithms presented in Theorem 3.2 and Corollary 3.3 do not sort stably, since the order of buffer elements may change. If some elements used in buffers are equal, their original order cannot be recovered. This leaves us with a fascinating question:

> *Does there exist an algorithm operating in-place and performing, in the worst case, at most* $O(n \cdot \log n)$ *comparisons,* $O(n)$ *moves,* $O(n \cdot \log n)$ *arithmetic operations, and, at the same time, sorting elements stably, so that the relative order of equal elements is preserved?*

At the present time, we dare not formulate any conjectures about this problem. The best known algorithm for stable in-place sorting with $O(n)$ moves is still the one presented in Munro and Raman [1996a], performing $O(n^{1+\varepsilon})$ comparisons in the worst case.

We are also firmly convinced that the upper bounds of Theorem 3.2 and Corollary 3.3 are not optimal and can be improved, which is left as another open problem.

REFERENCES

CARLSSON, S. 1992. A note on Heapsort. *Comput. J. 35*, 410–411.

FLOYD, R. 1964. Treesort 3 (Algorithm 245). *Commun. ACM 7*, 701.

FRANCESCHINI, G. 2003. An in-place sorting algorithm performing $O(n \log n)$ comparisons and $O(n)$ data moves. Tech. rep., Dipartimento di Informatica, Università di Pisa. March. (Available from ftp://ftp.di.unipi.it/pub/techreports/TR-03-06.ps.Z.)

GEFFERT, V. 2002. Sorting with $O(n \log n)$ comparisons and $O(n)$ transports, in-place, in the worst case, simultaneously. Tech. rep., P. J. Šafárik University. July. (Available from http://ics.upjs.sk/techreports/2002/ultim.ps.)

GEFFERT, V., AND KOLLÁR, J. 2001. Linear-time in-place selection in $\varepsilon \cdot n$ element moves. Tech. rep., P. J. Šafárik University. April. (Available from http://ics.upjs.sk/techreports/2002/select.ps.)

ITAI, A., KONHEIM, A., AND RODEH, M. 1981. A sparse table implementation of priority queues. In *Proceedings of the International Colloquium on Automata, Languages, & Programming*. Lecture Notes in Computer Science, vol. 115. Springer-Verlag, New York, 417–431.

KATAJAINEN, J., AND PASANEN, T. 1999. In-place sorting with fewer moves. *Inf. Process. Lett. 70*, 31–37.

KATAJAINEN, J., PASANEN, T., AND TEUHOLA, J. 1996. Practical in-place Mergesort. *Nord. J. Comput. 3*, 27–40.

KNUTH, D. 1973. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass. (Second edition: 1998).

LI, M., AND VITÁNYI, P. 1993. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, Section 6.3.1: Heapsort, 334–338.

MUNRO, J., AND RAMAN, V. 1992. Sorting with minimum data movement. *J. Algorithms 13*, 374–393.

MUNRO, J., AND RAMAN, V. 1996a. Fast stable in-place sorting with $O(n)$ data moves. *Algorithmica 16*, 151–160.

MUNRO, J., AND RAMAN, V. 1996b. Selection from read-only memory and sorting with minimum data movement. *Theoret. Comput. Sci. 165*, 311–323.

RAMAN, V. 1991. Sorting in-place with minimum data movement. Ph.D. dissertation, Univ. Waterloo, Dept. Comput. Sci. (Tech. Rep. 91-12).

REINHARDT, K. 1992. Sorting in-place with a worst case complexity of $n \log n - 1.3n + O(\log n)$ comparisons and $\varepsilon n \log n + O(1)$ transports. In *Proceedings of the International Symposium on the Algorithms and Computers*. Lecture Notes in Computer Science, Vol. 650. Springer-Verlag, New York, 489–498.

SCHAFFER, R., AND SEDGEWICK, R. 1993. The analysis of Heapsort. *J. Algorithms 15*, 76–100.

WEGENER, I. 1993. Bottom-Up-Heapsort, A new variant of Heapsort beating, on an average, Quicksort (if n is not very small). *Theoret. Comput. Sci. 118*, 81–98.

WILLARD, D. 1982. Maintaining dense sequential files in a dynamic environment. In *Proceedings of the Symposium on the Theory of Computing*. ACM, New York, 114–121.

WILLIAMS, J. 1964. Heapsort (Algorithm 232). *Commun. ACM 7*, 347–348.

RECEIVED MAY 2003; REVISED JUNE 2005; ACCEPTED MAY 2005