Two Constant-Factor-Optimal Realizations of Adaptive Heapsort*

Stefan Edelkamp[1]**, Amr Elmasry[2], and Jyrki Katajainen[2]***

[1] TZI, Universität Bremen, Germany
[2] Department of Computer Science, University of Copenhagen, Denmark

Abstract. In this paper we introduce two efficient priority queues. For both, *insert* requires $O(1)$ amortized time and *extract-min* $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons, where n is the number of elements stored. One priority queue is based on a weak heap (array-based) and the other on a weak queue (pointer-based). In both, the main idea is to temporarily store the inserted elements in a buffer, and once it is full to move its elements to the main queue using an efficient bulk-insertion procedure. By employing the new priority queues in adaptive heapsort, we guarantee, for several measures of disorder, that the formula expressing the number of element comparisons performed by the algorithm is optimal up to the constant factor of the high-order term. We denote such performance as constant-factor optimality. Unlike some previous constant-factor-optimal adaptive sorting algorithms, adaptive heapsort relying on the developed priority queues is practically workable.

Keywords. Priority queues, weak heaps, weak queues, adaptive sorting, adaptive heapsort, constant-factor optimality

1 Introduction

A sorting algorithm is *adaptive* if it changes its performance according to the presortedness within the input. When sorting n elements, the running time of such algorithm is $O(n)$ for sequences that are sorted or almost sorted, and $O(n \lg n)$ for sequences that have a high degree of disorder. It is important to note that such algorithms do not know in advance about the amount of existing disorder.

In the literature, several *measures of disorder* that characterize the input sequence have been considered [18]. An adaptive sorting algorithm is said to be *asymptotically optimal*, or simply *optimal*, if its running time asymptotically matches the lower bound derived using the number of input elements and the amount of disorder as parameters. In this paper we focus on a stronger form of optimality. We call an asymptotically-optimal algorithm *constant-factor-optimal*, if

** Partially supported by DFG grant ED 74/8-1.
*** Partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools").

the number of element comparisons performed matches the information-theoretic lower bound up to the constant factor hidden behind the big-Oh notation, i.e. the constant factor multiplied by the high-order term.

Some of the known measures of disorder are the number of oscillations *Osc* [15], the number of inversions *Inv* [14], the number of runs *Runs* [14], the number of blocks *Block* [3], and the measures *Max*, *Exc* and *Rem* [3]. Some measures dominate the others: every *Osc*-optimal algorithm is *Inv* optimal and *Runs* optimal; every *Inv*-optimal algorithm is *Max* optimal [15]; and every *Block*-optimal algorithm is *Exc* optimal and *Rem* optimal [3]. Natural mergesort, described by Knuth [14, Section 5.2.4], is an example of an adaptive sorting algorithm that is constant-factor-optimal; this is with respect to the measure *Runs*.

For a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$, the number of inversions is the number of pairs of elements that are in the wrong order, i.e. $Inv(X) = |\{(i,j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\}|$ [14, Section 5.1.1]. An optimal algorithm with respect to the measure *Inv* sorts a sequence X in $\Theta(n \lg \big(Inv(X)/n\big) + n)$ time. The optimality is implied by the information-theoretic lower bound $\Omega(n \lg \big(Inv(X)/n\big) + n)$ known for the number of element comparisons performed by any sorting algorithm with respect to the parameters n and *Inv* [12]. A constant-factor-optimal algorithm should perform at most $n \lg \big(Inv(X)/n\big) + O(n)$ element comparisons.

Several adaptive sorting algorithms are known to be asymptotically optimal with respect to the measure *Inv*. The known approaches are inspired by insertionsort [7, 8, 12, 19, 20, 23], quicksort [16], mergesort [8, 21], or heapsort [6, 15]. However, only a few of the known algorithms are constant-factor-optimal; these are the insertionsort-based and mergesort-based algorithms of Elmasry and Fredman [8], and the heapsort-based algorithm of Levcopoulos and Petersson [15] when implemented with the multipartite priority queue of Elmasry et al. [10]. Most of these algorithms are complicated and have never been implemented.

Adaptive heapsort [15] is optimal with respect to all the aforementioned measures of disorder. We recall the description and analysis of adaptive heapsort in Section 2. In this paper we present two new realizations of adaptive heapsort. Our main motivation is to use, instead of a worst-case efficient priority queue [10], a simpler priority queue that can support *insert* in $O(1)$ amortized time and *extract-min* in $O(\lg n)$ worst-case time including at most $\lg n + O(1)$ element comparisons. From these bounds and the analysis given in [15], the constant-factor optimality follows for the following measures of disorder: *Osc*, *Inv*, *Runs*, and *Max*. Our main contribution is to present two priority queues with the required performance guarantees. The first priority queue improves over a weak heap (an array-based priority queue described in the context of sorting by Dutton [4] and further analysed by Edelkamp and Wegener [5]). We modify and analyse this priority queue in Section 3. The second priority queue improves over a weak queue (a binomial queue implemented using binary trees as suggested by Vuillemin [24]). We modify and analyse this priority queue in Section 4. The simple—but powerful—tool we used in both data structures is a buffer, into which the new elements are inserted. When the buffer becomes full, all its elements are moved

to the main queue. In accordance, for both priority queues, we give an efficient bulk-insertion procedure.

We demonstrate the effectiveness of our approach by comparing the new realizations to the best implementations of known efficient sorting algorithms (splaysort [20] and introsort [22]). Our experimental settings, measurements, and outcomes are discussed in Section 5.

2 Adaptive Heapsort

In this section we describe the basic version of adaptive heapsort [15] and summarize the analysis of its performance.

The algorithm begins by building the *Cartesian tree* [25] for the input $X = \langle x_1, \ldots, x_n \rangle$. The root of the Cartesian tree stores $x_k = \min\{x_1, \ldots, x_n\}$, the left subtree of the root is the Cartesian tree for $\langle x_1, \ldots, x_{k-1} \rangle$, and the right subtree of the root is the Cartesian tree for $\langle x_{k+1}, \ldots, x_n \rangle$. Such a tree can be built in $O(n)$ time [11] by scanning the input in order and inserting each element x_i into the existing tree as follows. The nodes along the *right spine* of the tree (the path from the root to the rightmost leaf) are traversed bottom up, until a node with an element x_j that is not larger than x_i is found. In such case, the right subtree of the node of x_j is made the left subtree of the node of x_i, and the node of x_i is made the right child of the node of x_j. If x_i is smaller than all the elements on the right spine, the whole tree is made the left subtree of the node of x_i. This procedure requires at most $2n - 3$ element comparisons [15].

The algorithm proceeds by moving the smallest element at the root of the Cartesian tree into a priority queue. The algorithm then continues by repeatedly outputting and deleting the minimum from the priority queue. After each deletion, the elements at the children of the Cartesian-tree node corresponding to the deleted element are inserted into the priority queue. As for the priority-queue operations, n *insert* and n *extract-min* operations are performed. But, the heap will be small if the input sequence has a high amount of existing order.

The following improvement to the algorithm [15] is both theoretically and practically effective; even though, in this paper, it only affects the constant in the linear term. Since at least $\lfloor n/2 \rfloor$ of the *extract-min* operations are immediately followed by an *insert* operation (deleting a node that is not a leaf of the Cartesian tree must be followed by an insertion), every such *extract-min* can be combined with the following *insert*. This can be implemented by replacing the minimum of the priority queue with the new element and thereafter reestablishing the heap properties. Accordingly, the cost for half of the insertions will be saved.

The worst-case running time of the algorithm is $O(n \lg \big(Osc(X)/n\big) + n) = O(n \lg \big(Inv(X)/n\big) + n)$ [15]. For a constant β, the number of element comparisons performed is $\beta n \lg \big(Osc(X)/n\big) + O(n) = \beta n \lg \big(Inv(X)/n\big) + O(n)$. Levcopolous and Petersson suggested using a binary heap [26], which results in $\beta = 3$ (can be improved to $\beta = 2.5$ by combining *extract-min* and *insert* whenever possible). By using a binomial queue [24], we get $\beta = 2$. By using a weak heap [4], we get $\beta = 2$ (can be improved to $\beta = 1.5$ by combining *extract-min*

and *insert*). By using the complicated multipartite priority queue [10], we indeed get the optimal $\beta = 1$. The question then arises whether we can achieve the constant-factor optimality, i.e. $\beta = 1$, and in the meantime ensure practicality!

In addition to the priority queue, the storage required by the algorithm is $2n$ extra pointers for the Cartesian tree. (We need not keep parent pointers since, during the construction, on the right spine of the tree we can temporarily revert each right-child pointer to point to the parent.) We also need to store the n elements inside the nodes of the Cartesian tree, either directly or indirectly.

3 Weak Heaps with Bulk Insertions

A *weak heap* [4] is a binary tree, where each node stores an element. A weak heap is obtained by relaxing the requirements of a binary heap [26]. The root has no left child, and the leaves are found at the last two levels only. The height of a weak heap that has n elements is therefore $\lceil \lg n \rceil + 1$. The *weak-heap property* enforces that the element stored at a node is not larger than all the elements stored in the right subtree of that node. In our implementation, illustrated in Fig. 1, besides the element array a an array r of *reverse bits* is used, i.e. $r_i \in \{0, 1\}$ for $i \in \{0, \ldots, n-1\}$. We use a_i to refer to either the element at index i of array a or to a node in the corresponding tree structure. A weak heap is laid out such that, for a_i, the index of its left child is $2i + r_i$, the index of its right child is $2i + 1 - r_i$, and (assuming $i \neq 0$) the index of its parent is $\lfloor i/2 \rfloor$. Using the fact that the indices of the left and the right children of a_i are exchanged when flipping r_i, subtrees can be swapped in constant time by setting $r_i \leftarrow 1 - r_i$.

The *distinguished ancestor* of a_i, $i \neq 0$, is the parent of a_i if a_i is a right child, and the distinguished ancestor of the parent of a_i if a_i is a left child. We use *d-ancestor*(i) to denote the index of such ancestor. The weak-heap property enforces that no element is smaller than that at its distinguished ancestor.

To *insert* a new element e, we first add e to the next available array entry, making it a leaf in the heap. To reestablish the weak-heap property, as long as e is smaller than the element at its distinguished ancestor, we swap the two

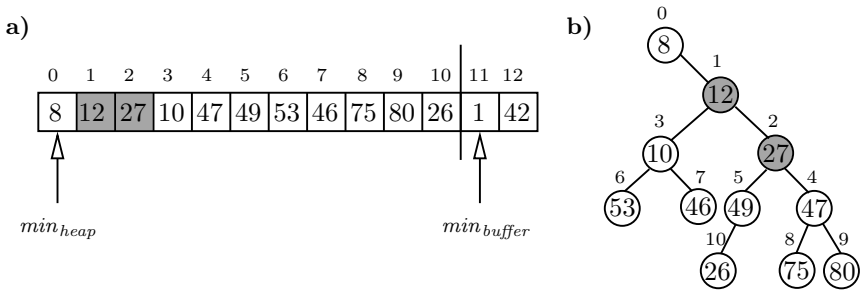

Fig. 1. A weak heap of size 11 and a buffer of size 2: **a)** the array representation and **b)** the corresponding tree representation of the weak heap; the nodes, for which the reverse bits are set, are highlighted.

elements and repeat this for the new location of e. It follows that *insert* requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons.

The subroutine *link* combines two weak heaps into one weak heap conditioned on the following settings. Let a_i and a_j be two elements in a weak heap, such that a_i is not larger than all the elements in the left subtree of a_j. Conceptually, a_j and its right subtree form a weak heap, while a_i and the left subtree of a_j form another weak heap. (Note that a_i could be at any location of the array.) If $a_j < a_i$, the subroutine *link* swaps the two elements and flips r_j; otherwise it does nothing. As a result, a_j will not be larger than any of the elements in its right subtree, and a_i will not be larger than any of the elements in the subtree rooted at a_j. All in all, *link* requires $O(1)$ time and involves one element comparison.

To perform *extract-min*, the element stored at the root of the weak heap is swapped with that stored at the last occupied array entry. To restore the weak-heap property, repeated *link* operations are performed that involve the current root of the weak heap; the details follow. The last node on the *left spine* (the path from a node to the leftmost leaf) of the right child of the root is identified. Starting from the child of the root, this is done by repeatedly traversing left children until reaching a node that has no left child. The path from this node to the child of the root is traversed upwards, and *link* operations are repeatedly performed between the root of the weak heap and the nodes along this path. The correctness of the *extract-min* operation follows from the fact that, after each *link*, the element at the root of the heap is not larger than all the elements in the left subtree of the node to be considered in the next *link*. Thus, *extract-min* requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons.

The cost of *insert* can be improved to an amortized constant. The key idea is to use a *buffer* that supports constant-time insertion. The buffer can be implemented as a resizable array. Additionally, a pointer to the minimum element in the buffer is maintained. The maximum size of the buffer is set to $\lceil \lg n \rceil$, where n is the total number of elements stored. A new element is inserted into the buffer as long as its size is below the threshold. Once the threshold is reached, a *bulk insertion* is performed by moving all the elements of the buffer to the weak heap. For the *extract-min* operation, the minimum of the buffer is compared with the minimum of the weak heap, and accordingly the operation is performed either in the buffer or in the weak heap. Deleting the minimum of the buffer is done by removing the minimum and scanning the buffer to determine the new minimum. Almost matching the bounds for the weak heap, deleting the minimum of the buffer requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil - 2$ element comparisons. Thus, *extract-min* involves at most $\lceil \lg n \rceil + 1$ element comparisons.

Let us now consider how to perform a bulk insertion in $O(\lg n)$ time (see Fig. 2). First, we move the elements of the buffer to the next available entries of the array that stores the weak heap. The main idea is to reestablish the weak-heap property bottom-up level-by-level. Starting with the inserted nodes, for each node we *link* its distinguished ancestor to it. We then consider the parents of these nodes on the next upper level, and for each parent we *link* its distinguished ancestor to it, restoring the weak-heap property up to this level.

```
input: a: array of elements, r: array of bits, buffer: array of elements
right ← size(a) + size(buffer) − 1
left ← max{size(a), ⌊right/2⌋}
while size(buffer) > 0
    size(a)++
    a[size(a) − 1] ← buffer[size(buffer) − 1]
    size(buffer)−−
    size(r)++
    r[size(r) − 1] ← 0
while right > left + 1
    for j ∈ {right, right − 1, . . . , left}
        i ← d-ancestor(j)
        if a[j] < a[i]
            swap(a[i], a[j])
            r[j] ← 1 − r[j]
    left ← ⌊left/2⌋
    right ← ⌊right/2⌋
for j ∈ {left, right}
    while j ≠ 0
        i ← d-ancestor(j)
        if a[j] < a[i]
            swap(a[i], a[j])
            r[j] ← 1 − r[j]
        j ← i
```

Fig. 2. The pseudo-code for bulk insertion in a weak heap.

This is repeated until the number of nodes that we need to deal with at a level is two (or less). At this point, we switch to a more efficient strategy. For each of these two nodes, we reestablish the weak-heap property by traversing the path from such node towards the root. If the value of the current node x is smaller than that at its distinguished ancestor, we *link* the distinguished ancestor to x. We then repeat after setting x to be its old distinguished ancestor.

The correctness of the bulk-insertion procedure follows since, before considering the ith level, the value at any node below level i is not smaller than that at its distinguished ancestor. Hence, the value at the distinguished ancestor of a node x at level i is guaranteed not to be larger than the value at any node of the left subtree of x; this ensures the validity of the *link* operations to be performed at level i. Once we reach the root, the weak-heap property is valid for all nodes.

Let k be the number of elements moved from the buffer to the weak heap by the bulk-insertion procedure. The number of element comparisons performed at the ith iteration equals the number of *link* operations at the ith last level of the weak heap, which is at most $\lfloor (k-2)/2^{i-1} \rfloor + 2$. Here, we use the fact that the number of parents of a contiguous block of b elements in the array of a weak heap is at most $\lfloor (b-2)/2 \rfloor + 2$. Since the number of iterations is at most $\lceil \lg n \rceil$, the total number of element comparisons is less than $\sum_{i=1}^{\lceil \lg n \rceil} (1/2^{i-1} \cdot k + 2) < 2k + 2\lceil \lg n \rceil$.

When $k = \lceil \lg n \rceil$, the number of element comparisons is less than $4\lceil \lg n \rceil$; this accounts for four comparisons per element in the amortized sense. Due to the bulk insertion and the check for whether the minimum of the buffer is up to date or not, *insert* involves amortized five element comparisons in total.

The running time of the bulk insertion is dominated by the localization of the distinguished ancestors of the involved nodes. To find the distinguished ancestor of a node, we repeatedly go to the parent and check whether the current node is its right child or not. We call such an operation an *ancestor check*. We separately consider two parts of the procedure. The first part comprises the process of finding the distinguished ancestors for the levels with more than two involved nodes. Recall that the total number of those nodes at the ith last level is $k/2^{i-1} + O(1)$, for a total of $2k + o(k)$. Among the nodes involved, at most $(2k + o(k))/2^j$ need j ancestor checks to get to the distinguished ancestor, where $j \geq 1$. This accounts for at most $\sum_{j \geq 1} j/2^{j-1} \cdot (k + o(k)) < 4(k + o(k)) = 4\lceil \lg n \rceil + o(\lg n)$ ancestor checks. The second part comprises two path traversals towards the root, which involve at most $2\lceil \lg n \rceil$ ancestor checks in total. We conclude that the amortized cost accounted per element is a constant.

4 Weak Queues with Bulk Insertions

In this section we resort to a binomial queue that is implemented using binary trees [24]; we call this variant a *weak queue*. This data structure is a collection of perfect weak heaps (binomial trees in the binary-tree form), where the size of each tree is a power of two. The binary representation of n specifies the sizes of the perfect weak heaps that are present. A 1-bit at position r indicates that a perfect weak heap of size 2^r is present. The *rank* of a perfect weak heap of size 2^r is r. In our implementation, illustrated in Fig. 3, every node stores a pointer to its left child, a pointer to its right child, and (a pointer to) an element.

Two perfect weak heaps of rank r can be *linked* to form a perfect weak heap of rank $r + 1$, by making the root that has the smaller element the root of the resulting weak heap, the other root the right child of the new root, and the previous right child of the new root the left child of the other root.

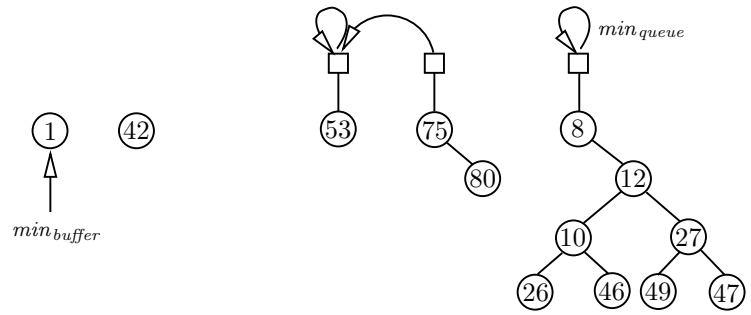

Fig. 3. A buffer of size 2 and a weak queue of size 11 (1011 in binary).

To *insert* a node into a weak queue, we let the new node form a single-node tree. This may trigger a sequence of *link* operations until no two trees of the same rank exist. Still, the amortized cost per *insert* is a constant [24].

To perform *extract-min*, we scan the roots of the perfect weak heaps to find the minimum. We then borrow the root of the smallest tree; let that root be x. In accordance, every node on the left spine of x's right subtree becomes the root of a perfect weak heap, and these heaps are added to the collection. Hereafter, we detach the root with the minimum value; let that root be y. Now every node on the left spine of y's right subtree is the root of a perfect weak heap. Using repeated *link* operations, the node x is combined with the roots of these heaps to create a perfect weak heap that has the same size as the heap rooted at y before the deletion. (A *link* operation is performed between x and the root of the smallest such heap, and the resulting heap is repeatedly linked with the next larger remaining heap, and so on.) It follows that *extract-min* requires $O(\lg n)$ time and involves at most $2\lceil \lg n \rceil - 2$ element comparisons.

To speed things up, we maintain prefix-minimum pointers for the roots of the perfect weak heaps (for the origin of this idea, consult [10] and the references therein). The prefix-minimum pointer of the root of a heap of size 2^r points to the root with the smallest value among the heaps of size 2^j for $j \leq r$. The overall minimum can be located by following the prefix-minimum pointer of the root of the largest heap. Now we have to borrow a node such that the prefix-minimum pointers need not be updated. As before, the borrowed node is repeatedly linked with the roots of the heaps resulting from detaching the minimum node. This requires r element comparisons if the rank of the deleted node is r. We still have to update the prefix-minimum pointers. The key idea is that we only need $\lceil \lg n \rceil - r$ element comparisons to update the prefix-minimum pointers of the larger heaps. Hence, *extract-min* involves at most $\lceil \lg n \rceil$ element comparisons.

If we implement *insert* in the normal way, we then have to update the prefix-minimum pointers; this would require a logarithmic number of element comparisons. Our way out is again to rely on bulk insertions (see Fig. 4). We collect at most $\lceil \lg n \rceil$ elements into a buffer, where n is the total number of elements stored. The buffer is implemented as a circular singly-linked list, having its minimum first. When the buffer becomes full, we clear it by repeatedly inserting its elements into the weak queue in the normal way, without updating the prefix-minimum pointers. After finishing these insertions, the prefix-minimum pointers are updated once. For the bulk insertion, an amortized analysis accounts for a constant amortized cost per element, involving amortized two element comparisons. Since it is necessary to maintain the minimum of the buffer, an insertion into the buffer involves one element comparison. This together with the bulk insertion accounts for three element comparisons per *insert*.

We have to implement borrowing carefully so that it does not invalidate the prefix-minimum pointers. While performing no element comparisons, it takes $O(\lg n)$ time. If the buffer is non-empty, a node is borrowed from there. Otherwise, a node is borrowed from the main queue. If the size of the smallest heap is larger than one, the last node from the left spine of the right subtree of its

```
input: Q: queue of perfect weak heaps, buffer: list of nodes
while size(buffer) > 0
    x ← pop(buffer)
    insert(Q, x)
update-prefix-minimum-pointers(Q)
```

Fig. 4. The pseudo-code for bulk insertion in a weak queue. For a list, subroutine *pop* removes and returns its last node. For a queue, subroutine *insert* adds the given node to the queue and makes the necessary linkings leaving at most one heap per rank. Subroutine *update-prefix-minimum-pointers* updates all the prefix-minimum pointers as they may not be up to date after *insert* operations.

root is borrowed. The root and the other nodes on the left spine are added as new roots to the main structure, and the prefix-minimum pointers associated with each of them is set to point to the old root (rooting a heap of size one now). Otherwise, the smallest heap is a singleton. This singleton is borrowed if the prefix-minimum pointer of the second smallest heap does not point to it. Otherwise, these two roots are swapped and the current singleton is borrowed.

For the *extract-min* operation, the minimum of the buffer is compared with the minimum of the weak queue, and accordingly the operation is performed either in the buffer or in the weak queue. After these modifications, *extract-min* involves at most $\lceil \lg n \rceil + 1$ element comparisons.

5 Experimental Findings

We implemented two versions of adaptive heapsort, one using a weak heap and another using a weak queue. In this section we discuss the settings and outcomes of our performance tests. In these tests we measured the actual running time of the programs and the number of element comparisons performed. The main purpose for carrying out these experiments was to validate our theoretical results.

Our implementation of adaptive heapsort using a weak heap was array-based. Each entry of the array representing the Cartesian tree stored a copy of an element and two references to other entries in the tree. The arrays representing the weak heap and the buffer stored references to the Cartesian tree, and a separate array was used for the reverse bits. In total, the space usage per element was three references, a copy of the element, and one bit. Dynamic memory allocation was avoided by preallocating all arrays from the stack. Users should be aware that, due to the large space requirements, the algorithm has a restricted utility depending on the amount of memory available.

In our implementation of adaptive heapsort using a weak queue, some non-trivial enhancements were made. First, we used two pointers per node: one pointing to the left child and another to the right child. As advised by Vuillemin [24], because of the lack of parent pointers, we reverted the left-child pointers to temporarily point to the parents while performing repeated linkings in *extract-min*. Second, we used an array of pointers to access the roots of the trees. This array

also infers the ranks of these roots; the nodes themselves did not store any rank information. Third, we used the same nodes to store the pointers needed by the Cartesian tree and the buffer. Fourth, all memory was preallocated from the stack. In total, the space usage per node was four pointers and a copy of an element; another $O(\lg n)$ space was used by the array of root pointers and the array of prefix-minimum pointers. Accordingly, this implementation used even more memory than the version employing a weak heap.

To select suitable competitors for our implementations, we consulted some earlier research papers concerning the practical performance of inversion-optimal sorting algorithms [9, 20, 23]. Based on this survey, we concluded that splaysort performs well in practice. In addition, the implementation of Moffat et al. [20] is highly tuned, practically efficient, and publicly available. Consequently, we selected their implementation of splaysort as our primary competitor. In the aforementioned experimental papers, splaysort has been reported to perform better than other tree-based algorithms (e.g. AVL-sort [7]), cache-oblivious algorithms (e.g. greedysort [1]), and partition-based algorithms (e.g. splitsort [16]).

When considering comparison-based sorting, one should not ignore quicksort [13]. Introsort [22] is a highly tuned variant of quicksort that is known to be fast in practice. It is based on half-recursive median-of-three quicksort, it coarsens the base case by leaving small subproblems unsorted, it calls insertionsort to finalize the sorting process, and it calls heapsort if the recursion depth becomes too large. Using the middle element as a candidate for the pivot, and using insertionsort at the back end, make introsort adaptive with respect to the number of element comparisons (though not optimally adaptive with respect to any known measure of disorder). Quicksort and its variants are also known to be optimally adaptive with respect to the number of element swaps performed [2]. For these reasons, we selected the standard-library implementation of introsort shipped with our C++ compiler as our secondary competitor.

In the experiments, the results of which are discussed here (see Figs. 5–8), we used 4-byte integers as input data. The results were similar for different input sizes; for the reported experiments the number of elements was fixed to 10^7 and 10^8. We ensured that all the input elements were distinct. Integer data was sufficient to back up our theoretical analysis. However, for other types of input data, the number of element comparisons performed and the number of cache misses incurred may have more significant influence on the running time.

We performed the experiments on one core of a desktop computer (model Intel i/7 CPU 2.67 GHz) running Ubuntu 10.10 (Linux kernel 2.6.32-23-generic). This computer had 32 KB L1 cache memory, 256 KB L2 cache memory, 8 MB (shared) L3 cache memory, and 12 GB main memory. With such memory capacity, there was no need to use virtual memory. We compiled all programs using GNU C++ compiler (`gcc` version 4.4.3 with option `-O3`).

To generate the input data, we used two types of generators:

Repeated swapping. We started with a sorted sequence of the integers from 1 to n, and repeatedly performed random transpositions of two consecutive elements. This generator was used to produce data with few inversions.

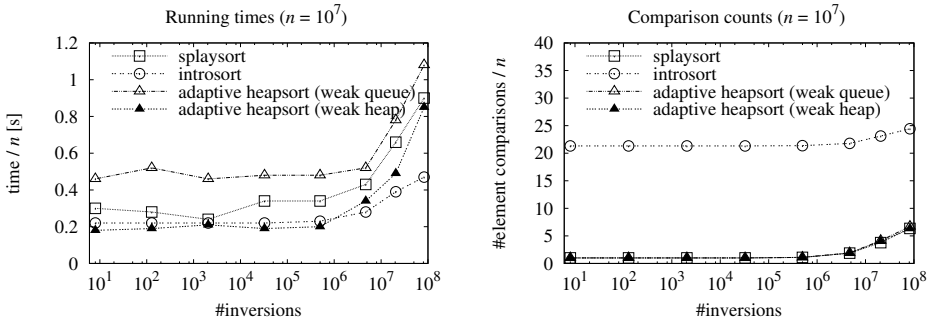

Fig. 5. Repeated swapping, $n = 10^7$: CPU time used and the number of element comparisons performed by different sorting algorithms.

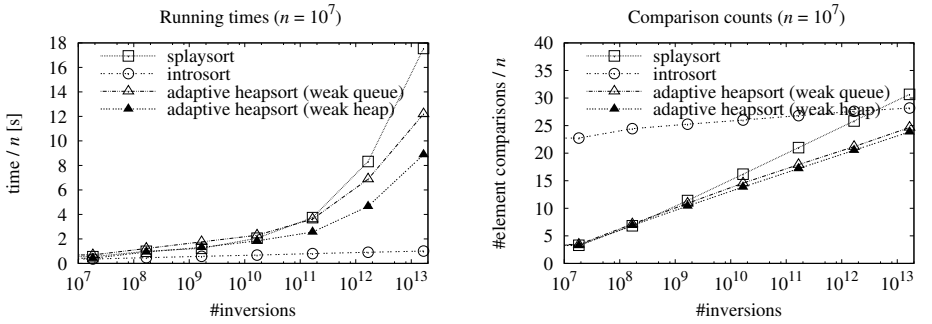

Fig. 6. Controlled shuffling, $n = 10^7$: CPU time used and the number of element comparisons performed by different sorting algorithms.

Controlled shuffling [9]**.** We started with a sorted sequence of the integers from 1 to n, and performed two types of perturbations; we call the sequences resulting from these two phases *local* and *global shuffles*. For local shuffles, the sorted sequence was broken into $\lceil n/m \rceil$ consecutive blocks each containing m elements (except possibly the last block), and the elements of each block were randomly permuted. For global shuffles, the sequence produced by the first phase was broken into m consecutive blocks each containing $\lceil n/m \rceil$ elements (except possibly the last block). From each block one element was selected at random, and these elements were randomly permuted. A small value of m means that the sequence is sorted or almost sorted, and a large value of m means that the sequence is random. Given a parameter m, this shuffling results in a sequence with expected $\Theta(n \cdot m)$ inversions.

Since in both cases the resulting sequence is a permutation of the integers from 1 to n, the number of inversions could be easily calculated as $\sum_{i=1}^{n} |x_i - i|/2$.

The experiments showed that our realizations of adaptive heapsort perform a low number of element comparisons. For both versions, the number of ele-

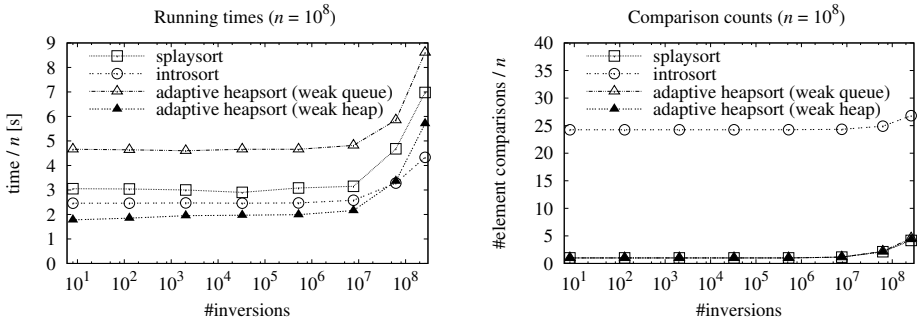

Fig. 7. Repeated swapping, $n = 10^8$: CPU time used and the number of element comparisons performed by different sorting algorithms.

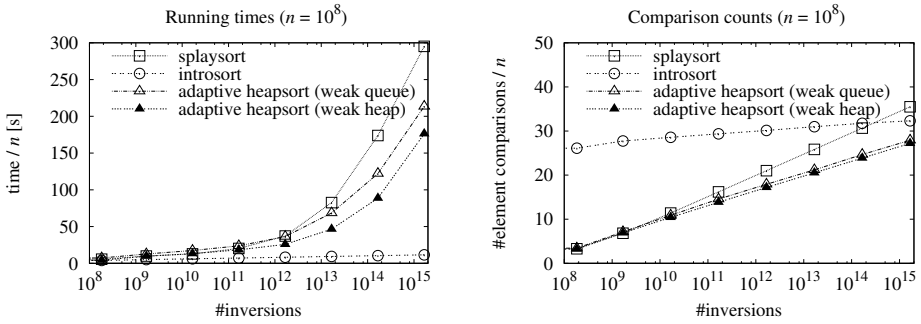

Fig. 8. Controlled shuffling, $n = 10^8$: CPU time used and the number of element comparisons performed by different sorting algorithms.

ment comparisons was about the same, as already verified analytically. When the number of inversions was small, splaysort performed about the same number of element comparisons as the two realizations of adaptive heapsort. When the number of inversions was large, splaysort performed a few more element comparisons than the two realizations of adaptive heapsort. In all our experiments, introsort was a bad performer with respect to the number of element comparisons; it showed very little adaptivity and came last in the competition.

As to the running time, the weak-heap version of adaptive heapsort was faster than the weak-queue version; about 60% faster when the number of inversions was small and about 20% faster when the number of inversions was large. The running times of splaysort were larger than the ones of the weak-heap version for almost all experiments. For random data, splaysort performed worst, and adaptive heapsort could be up to a factor of 15 slower than introsort. (In our supplementary experiments, for random data, normal heapsort was only a factor of 2–6 slower than introsort depending on the input size.) In most experiments, introsort was the fastest sorting method; it was only beaten by the weak-heap version when the number of inversions was very small (less than n).

6 Conclusions

We studied the optimality and practicality of adaptive heapsort. We introduced two new realizations for it, which are theoretically optimal and practically workable. Even though our realizations outperformed the state-of-the-art implementation of splaysort, the C++ standard-library introsort was faster for most inputs, at least on integer data. Despite decades of research, there is still a gap between the theory of adaptive sorting and the actual computing practice.

In spite of the optimality with respect to several measures of disorder, the high number of cache misses is not on our side. Compared to earlier implementations of adaptive heapsort, a buffer increased the locality of memory references and thus reduced the number of cache misses incurred. Still, introsort has considerably better cache behaviour. Earlier research has pointed out [9] that existing cache-efficient adaptive sorting algorithms are not competitive. The question arises whether constant-factor optimality with respect to the number of element comparisons can be achieved side by side to cache efficiency.

Another drawback of adaptive heapsort is the extra space required by the Cartesian tree. In introsort the elements are kept in the input array, and sorting is carried out in-place. Overheads attributable to pointer manipulations, and a high memory footprint in general, deteriorate the performance of any implementation of adaptive heapsort. This is in particular true when the amount of disorder is high. As to the memory requirements, we used about $4n$ extra words of storage for pointers and n extra space for copies of elements. An in-place algorithm that is optimal with respect to the measure *Inv* exists [17], but it is not practical. The question arises whether the memory efficiency of adaptive heapsort can be improved without sacrificing the optimal adaptivity.

In another extension, one should carry out experiments on data types for which element comparisons are more expensive than other operations. We are not far away from $n \lg n$ element comparisons when the amount of disorder is high. (Our best bound on the number of element comparisons is $n \lg \big(1 + Osc(X)/n\big) + 5.5n$.) The question arises whether the constant factor for the linear term in the number of element comparisons can be improved; that is, how close we can get to the information-theoretic lower bound up to low-order terms.

Source code

The programs used in the experiments are available via the home page of the CPH STL (http://cphstl.dk/) in the form of a PDF document and a `tar` file.

References

1. Brodal, G.S., Fagerberg, R., Moruz, G.: Cache-aware and cache-oblivious adaptive sorting. In: ICALP 2005. LNCS, vol. 3580, pp. 576–588. Springer, Heidelberg (2005)
2. Brodal, G.S., Fagerberg, R., Moruz, G.: On the adaptiveness of Quicksort. ACM J. Exp. Algorithmics 12, Article 3.2 (2008)

3. Carlsson, S., Levcopoulos, C., Petersson, O.: Sublinear merging and natural mergesort. Algorithmica 9(6), 629–648 (1993)
4. Dutton, R.D.: Weak-heap sort. BIT 33(3), 372–381 (1993)
5. Edelkamp, S., Wegener, I.: On the performance of Weak-Heapsort. In: STACS 2000. LNCS, vol. 1770, pp. 254–266. Springer, Heidelberg (2000)
6. Elmasry, A.: Priority queues, pairing and adaptive sorting. In: ICALP 2002. LNCS, vol. 2380, pp. 183–194. Springer, Heidelberg (2002)
7. Elmasry, A.: Adaptive sorting with AVL trees. In: Exploring New Fronties of Theoretical Informatics. IFIP Int. Fed. Inf. Process., vol. 155, pp. 315–324. Springer, New York (2004)
8. Elmasry, A., Fredman, M.L.: Adaptive sorting: An information theoretic perspective. Acta Inform. 45(1), 33–42 (2008)
9. Elmasry, A., Hammad, A.: Inversion-sensitive sorting algorithms in practice. ACM J. Exp. Algorithmics 13, Article 1.11 (2009)
10. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Trans. Algorithms 5(1), Article 14 (2008)
11. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: 16th Annual ACM Symposium on Theory of Computing. pp. 135–143. ACM, New York (1984)
12. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A new representation for linear lists. In: 9th Annual ACM Symposium on Theory of Computing. pp. 49–60. ACM, New York (1977)
13. Hoare, C.A.R.: Quicksort. Comput. J. 5(1), 10–16 (1962)
14. Knuth, D.E.: Sorting and Searching, The Art of Computer Programming, vol. 3. Addison Wesley Longman, Reading, 2nd edn. (1998)
15. Levcopoulos, C., Petersson, O.: Adaptive heapsort. J. Algorithms 14(3), 395–413 (1993)
16. Levcopoulos, C., Petersson, O.: Splitsort: An adaptive sorting algorithm. Inform. Process. Lett. 39(4), 205–211 (1991)
17. Levcopoulos, C., Petersson, O.: Exploiting few inversions when sorting: Sequential and parallel algorithms. Theoret. Comput. Sci. 163(1–2), 211–238 (1996)
18. Mannila, H.: Measures of presortedness and optimal sorting algorithms. IEEE Trans. Comput. C-34(4), 318–325 (1985)
19. Mehlhorn, K.: Sorting presorted files. In: 4th GI Conference on Theoretical Computer Science. LNCS, vol. 67, pp. 199–212. Springer, Heidelberg (1979)
20. Moffat, A., Eddy, G., Petersson, O.: Splaysort: Fast, versatile, practical. Software Pract. Exper. 126(7), 781–797 (1996)
21. Moffat, A., Petersson, O., Wormald, N.C.: A tree-based mergesort. Acta Inform. 35(9), 775–793 (1998)
22. Musser, D.R.: Introspective sorting and selection algorithms. Software Pract. Exper. 27(8), 983–993 (1997)
23. Saikkonen, R., Soisalon-Soininen, E.: Bulk-insertion sort: Towards composite measures of presortedness. In: SEA 2009. LNCS, vol. 5526, pp. 269–280. Springer, Heidelberg (2009)
24. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)
25. Vuillemin, J.: A unifying look at data structures. Commun. ACM 23(4), 229–239 (1980)
26. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM 7(6), 347–348 (1964)