Three Sorting Algorithms Using Priority Queues

Amr Elmasry

Computer Science Department
Alexandria University
Alexandria, Egypt.
elmasry@cs.rutgers.edu

Abstract. We establish a lower bound of $B(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$[1] on the number of comparisons performed by any algorithm that uses priority queues to sort n elements. Three sorting algorithms using priority queues are introduced. The first algorithm performs the same comparisons as the classical Mergesort algorithm, but in a different order. The second algorithm performs at most $2n \log n + O(n)$ comparisons, with the advantage of being adaptive; meaning that it runs faster when the input sequence has some presortedness. In particular, we show that this algorithm sorts an already sorted sequence in linear time; a fact that is not obvious since there is no special checks to guarantee this behavior. The third algorithm is almost implicit; it can be implemented using the input array and less than n extra bits. The number of comparisons performed by this algorithm is at most $B(n) + 2.5n$. The three algorithms have the advantage of producing every element of the sorted output, after the first, in $O(\log n)$, and can be implemented to be practically efficient.

1 Introduction

A well known sorting paradigm is sorting using priority queues, with plenty of references in the literature [11]. A priority queue is a heap-ordered general tree. The values in the heap are stored one value per tree node. The value stored in the parent of a node is smaller than or equal to the value stored in the node itself. We thus find the minimum heap value stored in the tree root. There is no restriction on the number of children a node may have, and the children of a node are maintained in a list of siblings. These selection sort algorithms produce the sorted sequence by repeatedly deleting the root of the queue and outputting its value then reconstructing the priority queue to maintain the heap property, in an operation that we call *deletemin*. The classical example of tree selection is by using a tournament tree. A tournament tree can be constructed by starting with all the elements at the bottom level. Every adjacent pair of elements is compared and the smaller element is promoted to the next level. When a winner moves up from one level to another it is replaced by the one that should eventually move up into its former place (namely the smaller of the two keys below). Once the smallest element reaches the root and is deleted, we can proceed to sort by a

[1] All logarithms in this paper are to the base 2.

T. Ibaraki, N. Katoh, and H. Ono (Eds.): ISAAC 2003, LNCS 2906, pp. 209–220, 2003.

top down method. The smallest descendent of the root is moved up, the smallest descendent of this latter element is moved up, and the process is repeated until reaching two empty nodes. This system of promotions is repeated with every *deletemin* operation. The number of comparisons performed by the tournament tree selection sort is at most $B(n) = n\lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$ [11].

We show that $B(n)$ is a lower bound on any selection sort algorithm that uses priority queues. It is well known that $\lceil \log n! \rceil < n \log n - 1.44n$ key comparisons is a lower bound for any comparison-based sequential sorting algorithm.

Our first algorithm performs the same comparisons performed by the Mergesort algorithm, but in a different order. See [7] for an analogous correspondence between other transformations on priority queues. In an abstract form, the Mergesort algorithm works as follows; the input sequence is split in two equal halves. Each half is recursively sorted, and these two sequences are then merged using the linear merge algorithm. Let $L(n)$ be the upper bound on the number of comparisons performed by the Mergesort algorithm. It is known that $B(n) \leq L(n)$ and $B(n) = L(n) = n \log n - n + 1$ when n is a power of 2 [11]. Our algorithm has an advantage over the Mergesort in that it requires $n - 1$ comparisons to produce the smallest element, and at most $\lceil \log n \rceil$ comparisons to produce each of the other elements. Hence, it is more suitable for order statistics.

Our second algorithm requires at most $2n \log n + O(n)$ comparisons. This algorithm is better than the first one when the input has fewer inversions. For an input sequence X of length n, the number of inversions $Inv(X)$ is defined

$$Inv(X) = |\{(i, j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\}|.$$

In this sense, our second algorithm is an adaptive sorting algorithm. There are many defined measures of presortedness, and plenty of known adaptive sorting algorithms, see [4,5,12,13,15]. In particular, a common property of all adaptive sorting algorithms is that such algorithms run in linear time when the input sequence is sorted. We show that our second algorithm has this property. Experimental results illustrate that our second algorithm performs fewer comparisons as the number of inversions in the input decreases. Inspired by the experimental results, we conjecture that this algorithm is optimal with respect to the number of inversions. In other words, we conjecture that it runs in $O(n \log \frac{Inv(X)}{n} + n)$.

Another challenge is to implement the selection sort algorithms within the input array without using extra storage. In this line of thinking the following algorithms were introduced. The worst case number of key comparisons in Heapsort independently introduced by Floyd [6] and Williams [22] is bounded by $2n \log n + O(n)$. Bottom-up-Heapsort (Wegener [21]) is a variant of Heapsort with at most $1.5n \log n + O(n)$ comparisons in the worst case. MDR-Heapsort proposed by McDiarmid and Reed [14] and analyzed by Wegener [20] performs less than $n \log n + 1.1n$ comparisons in the worst case and extends the Bottom-up-Heapsort by using, with every element, one bit to encode on which branch the smaller element of its children can be found and another one to mark if this information is unknown. Weak-Heapsort introduced by Dutton [2] and analyzed by Edelkamp and Wegener [3] is more elegant and faster. Instead of two bits

per element, Weak-Heapsort uses only one and requires at most $n \log n + 0.1n$ comparisons. Our third algorithm uses less than n extra bits in addition to the input array and requires at most $B(n) + 2.5n < n \log n + 1.6n$ comparisons.

All three algorithms rely on the notion of binomial queues. A binomial queue is a heap ordered tree that has the following structure. A binomial queue of rank r is constructed recursively by making the root of a binomial queue of rank $r-1$ the leftmost child of another binomial queue of rank $r-1$. A binomial queue of rank 0 consists of a single node. The rank of any node in a binomial queue equals the number of the children of this node. There are $\frac{n}{2^{i+1}}$ nodes of rank i in an n-node binomial queue, for all i from 0 to $\log n - 1$.

A basic operation for our algorithms is the pairing operation in which two queues are combined by making the root with the larger key value the leftmost child of the other root. Given a sequence of priority queues, a *halving* pass is implemented by combining these queues in pairs; every two adjacent queues are paired together starting from left to right (if the number of queues is odd, the rightmost queue is not paired). A *right-to-left incremental pairing* pass is implemented by combining the queues, in order from right to left, to form a single queue; each queue is paired with the single queue resulting from combining the queues to its right. A *multi-pass pairing* phase is implemented by repeatedly performing halving passes until a single queue is left. Given a sequence of n elements each stored in a single node, applying a multi-pass pairing phase, which requires $n-1$ comparisons, the heap becomes a binomial queue. If n is not a power of 2 there will be some missing nodes from the above definition of a binomial queue. We call the queue at this moment the initial binomial queue. (Notice the similarity between building the initial binomial queue and building the Weak-Heap of Dutton [2].)

2 Algorithm 1

Given a sequence of n elements each stored in a single node, this selection sort algorithm starts by performing a multi-pass pairing phase, deleting the smallest element and printing its value. Then, a right-to-left incremental pairing pass is repeated $n-1$ times, where each pass is followed by deleting the current smallest element from the queue and printing its value.

Lemma 1 *Using Algorithm 1, every element after the smallest requires at most* $\lceil \log n \rceil$ *comparisons to be produced.*

Proof. Omitted. □

Lemma 2 *Algorithm 1 performs the same comparisons performed by the Mergesort algorithm.*

Proof. Consider the multi-pass pairing phase. It is straight forward to observe that all the comparisons performed during this phase are also performed by the Mergesort algorithm. We define the following sorting algorithm, which we

call algorithm *, as follows. When algorithm * is applied on a priority queue it outputs the element in the root, applies the same algorithm recursively on each of the resulting sub-queues from right to left, and finally merges the resulting sequences in an incremental fashion from right to left using the classical linear merge algorithm. In fact, algorithm * is another way to describe the Mergesort algorithm. What is left is to show that algorithm *, when applied on the initial binomial queue, performs the same set of comparisons performed by the right-to-left incremental pairing passes of Algorithm 1.

For any priority queue α, let $r(\alpha)$ represents the element at the root of α, $S(\alpha)$ represents the sorted sequence of the elements of α, and let α^* represents the set of comparisons performed by algorithm * when applied on α. Given two queues α and β, let $\alpha.\beta$ denote the queue resulting from pairing the two queues, and $r(\alpha) \# r(\beta)$ represents the comparison between $r(\alpha)$ and $r(\beta)$. We show that

$$(r(\alpha) \# r(\beta)) \cup (\alpha.\beta)^* = \alpha^* \cup \beta^* \cup merge(S(\alpha), S(\beta)) \quad (1)$$

Where *merge(a,b)* stands for the set of comparisons performed by the classical linear merge algorithm when applied on the sorted sequences a and b. Assume without loss of generality that $r(\alpha) < r(\beta)$, in which case β will be linked to the root of α as its leftmost child. The way algorithm * works implies

$$(\alpha.\beta)^* = \alpha^* \cup \beta^* \cup merge(S(\alpha) - r(\alpha), S(\beta)).$$

The way the linear merge works implies

$$merge(S(\alpha), S(\beta)) = (r(\alpha) \# r(\beta)) \cup merge(S(\alpha) - r(\alpha), S(\beta)).$$

Equation (1) follows from the above facts.

Next, we show by backward induction on the comparisons performed by the right-to-left incremental pairing passes of Algorithm 1 that the same comparisons are performed if we apply algorithm *. The base case, which is in fact the last comparison, happens between two single nodes representing the largest two elements in the queues; in which case the two algorithms are trivially equivalent. Consider the priority queue at any moment of the algorithm, and let α and β be the rightmost two sub-queues. Applying Algorithm 1 on this queue results in the comparison $r(\alpha) \# r(\beta)$. Assuming the two algorithms perform the same comparisons from the point after this comparison and using (1), they are also equivalent before the comparison, and the induction hypothesis is true. □

3 Algorithm 2

The pairing heap [8] is a self adjusting heap structure that uses pairing in its implementation. In the standard two-pass variant of the pairing heaps, the *deletemin* operation proceeds by applying a halving pass on the sequence of queues, followed by a right-to-left incremental pairing pass. It has been proven [8] that the amortized cost of the number of comparisons involved in the *deletemin* operation for the two-pass variant of the pairing heaps is at most $2 \log n + O(1)$.

Given a sequence of n elements each stored in a single node, Algorithm 2 starts by performing a multi-pass pairing phase to delete the smallest element and print its value. Then, a two-pass pairing phase, similar to the pairing heap *deletemin* operation is repeated $n-1$ times, where each two-pass phase is followed by deleting the current smallest element from the queue and printing its value.

Lemma 3 *Algorithm* 2 *runs in* $O(n \log n)$ *and requires at most* $2n \log n + O(n)$ *comparisons.*

Proof. We mention below the basic idea of the proof while the details are omitted. We use the fact that the amortized cost of a *deletemin* operation is $2 \log n + O(1)$ comparisons. An extra $O(n)$ credits are required to pay for the potential after the multi-pass pairing phase, while the actual cost of this phase is $O(n)$. □

Lemma 4 *Given an input sequence of* n *elements that are sorted in ascending order from right to left, represented as* n *single nodes. Applying Algorithm 2 on this sequence requires less than* $6n$ *comparisons.*

Proof. Throughout the algorithm, when the value of a node is compared with its right sibling, the one to the left will have the larger value. Hence, the left node will be linked to the right node as its leftmost child.

The left spine of a node is defined to be the path from that node to the leftmost leaf of the sub-tree defined by this node. In other words, every node on the path is the leftmost child of its predecessor. The right spine is defined analogously. The halving passes are numbered, starting from t_0, the last halving pass of the first *deletemin* operation. We assume that the halving pass t takes place at time t. Consider any node x in the heap. Let $g_x(t)$ be the number of nodes on the left spine of x, after the halving pass t. Let $h_x(t)$ be the number of nodes on the left spine of the right sibling of x, after the same halving pass. If x does not have a right sibling, then $h_x(t)$ is equal to 0. Define $v_x(t)$ to be equal to $g_x(t) - h_x(t)$. We only consider the halving passes due to the fact that the values of $g(t)$ and $h(t)$ for all the nodes do not change during the right-to-left incremental pairing passes (except for the left sibling of the root of the rightmost queue, whose $h(t)$ decreases by one). We show by induction on time that for any node x, $v_x(t)$ is a positive non-decreasing function with time, and that this function increases if x is involved in a comparison during a halving pass. For the initial binomial queue, a property of binomial queues implies that the value of $v_x(t_0)$ is positive. This establishes the base case. Consider any node w and its right sibling z, such that w is linked to z as its leftmost child during the halving pass $t+1$, for any $t \geq t_0$. The following relations follow:

$$h_w(t) = g_z(t) \tag{2}$$

$$g_w(t+1) = g_w(t) \tag{3}$$

$$h_w(t+1) \leq h_w(t) - 1 \tag{4}$$

$$g_z(t+1) = g_w(t) + 1 \tag{5}$$

$$h_z(t+1) \leq h_z(t) + 1 \tag{6}$$

Relation (6) is a result of the fact that the number of nodes of the left spine of the right sibling of z may increase by at most one in a halving pass. Using (3) and (4), then $v_w(t+1) > v_w(t)$, and the hypothesis is true for the node w at time $t+1$. Using the induction hypothesis for node w at time t, then $v_w(t) > 0$. This relation together with (2) and (5) implies $g_z(t+1) > g_z(t) + 1$. Using the latter relation and (6), then $v_z(t+1) > v_z(t)$, and the hypothesis is true for node z at time $t+1$. The hypothesis is then true for all the nodes.

Of the links of the halving passes, we distinguish between two types of links. If $v_z(t) = 1$, we call the link an A-link. If $v_z(t) \geq 2$, we call the link a B-link. The above analysis indicates that any node may gain at most one child by an A-link. Hence, the number of comparisons accompanying A-links is at most n.

We use the accounting method [19] for bounding the number of comparisons accompanying B-links. After the first *deletemin* operation, we maintain the invariant that the number of credits on a node x after any halving pass t is $\frac{h_x^2(t)}{2}$. In the initial binomial queue, $h_x(t_0)$ equals the rank of x (the number of children of x), r_x. Let C be the number of credits needed to keep the invariant hold for the initial binomial queue, then

$$\begin{aligned} C &= \sum_x \frac{r_x^2}{2}, \\ &= \sum_{i=1}^{\log n - 1} \frac{i^2/2}{2^{i+1}} n \\ &< 1.5n. \end{aligned}$$

Next, we show that these credits are enough to pay for all the B-links, while maintaining the invariant. Let d be the difference between the sum of the number of credits on w and z before pass $t+1$ and those needed after pass $t+1$. Then

$$d = \frac{h_z^2(t)}{2} + \frac{h_w^2(t)}{2} - \frac{h_z^2(t+1)}{2} - \frac{h_w^2(t+1)}{2}.$$

Using (4) and (6), then

$$d \geq h_w(t) - h_z(t) - 1.$$

Using (2) together with the fact that for all the B-links $v_z(t) \geq 2$, then

$$d \geq 1.$$

This extra credit is used to pay for the comparison between w and z.

The above analysis implies that the total number of comparisons performed in the having passes is $2.5n$. This follows by adding the n comparisons bounding the A-links, and the $1.5n$ comparisons bounding the B-links. The number of comparisons performed in the right-to-left incremental pairing passes is bounded by the number of comparisons performed by the halving passes, for a total of at most $5n$ for both passes. The theorem follows by adding the $n-1$ comparisons done in the multi-pass pairing phase. □

4 Implementation Issues and Experimental Findings

The data structure we use to make these implementations efficient is the child, sibling representation, also known as the binary tree representation [11]. Each node has a left pointer pointing to its leftmost child and a right pointer pointing to its right sibling. The effect of the representation is to convert a heap-ordered tree into a half-ordered binary tree with empty right sub-tree, where by half-ordered we mean that the key of any node is at least as small as the key of any node in its left sub-tree.

It remains to investigate how our algorithms perform in practice. Since Algorithm 1 is another way to implement Mergesort, we need to relate it to the different ways known for implementing Mergesort. One of the well known methods to implement Mergesort is to use linked lists [11]. To save time in this implementation, the pointer manipulations are postponed as much as possible. Specifically, when the head of a list is found smaller than the head of the other, we keep traversing the first list until an element that is bigger than the head of the second list is encountered. The sub-list of the first list representing the elements smaller than the head of the second list are moved to the sorted output as a whole block. This saves several pointer manipulations, and improves the running time of the algorithm. On average, roughly speaking, more than half the work is saved about half the time. See [11] for more details. Several other tricks are used to improve the implementation of the linked list version of Mergesort [17]. Katajainen and Pasanen [9], and Reinhardt [16] show that Mergesort can be designed to achieve a bound of at most $n \log n - 1.3n + O(\log n)$ comparisons in the worst case. They [9,10] also gave a practical in-place Mergesort algorithm.

Consider the right-to-left incremental pairing passes. To save in the running time, we use a similar technique to the method that is mentioned above. After each iteration of the main loop, we keep the invariant that $value(l) > value(m)$, where l is the root of the rightmost queue, and m is the left sibling of l. The invariant is easily fulfilled before the loop, as follows. Let l be the root of the rightmost queue. We traverse the left siblings of l incrementally from right to left, as long as the values of the nodes are greater than the value of l. This list of siblings is linked to l, as $l's$ leftmost children, forming one sub-list in the same order. Within the main loop, a check is performed between the value of the left sibling of m and $value(m)$. If $value(m)$ is greater, l is linked to m as its leftmost child, and the iteration ends. Otherwise, we keep traversing the left sibling of the last traversed node, until a node whose value is smaller than $value(m)$ is encountered. The node l together with the whole list of left siblings of m (whose values are greater than $value(m)$) are linked as one sub-list forming the leftmost children of m, in the same order. This implementation saves several pointer manipulations, and improves the running time of the algorithm. On average, roughly speaking, more than half the work is saved about half the time. Another way of improving the running time of the algorithms is to use loop unfolding, a technique used in optimizing compilers. Here, we can save some commands and skip artificial variable renaming. By efficiently implementing Algorithm 1, we

were able to beat the running time of the best implementations for Mergesort we know about. The results of these experiments are omitted from this version.

Other experiments were performed on Algorithm 2, supporting the hypothesis that it is an efficient adaptive sorting algorithm. In our experiments we compare the results of sorting an input sequence that has some presortedness, when applied to Algorithm 2, Splaysort and Binomialsort. Slaysort is an adaptive sorting algorithm that relies on repeated insertions in a splay tree [18]. As a consequence of the dynamic finger theorem for splay trees (see Cole [1]) Splaysort is an optimal adaptive sorting algorithm. Moffat et al. [15] performed experiments showing that Splaysort is efficient in practice. Binomialsort [4] is another optimal adaptive sorting algorithm that is practically efficient and easy to implement. Both algorithms run in $O(n \log \frac{Inv(X)}{n} + n)$.

The input sequence is randomly generated such that the expected and worst case number of inversions is controlled. We start with a sorted sequence and perform two phases of permutations. For a given value of a parameter k, we want to permute the sorted sequence to have at most kn inversions. In the first phase, the sorted sequence is broken into consecutive blocks of $\frac{n}{k}$ elements each. From each block we select one element at random, for a total of k elements. These k elements are then randomly permuted. The number of inversions produced by this process is at most $\frac{nk}{2}$. In the second phase, the sequence is broken into consecutive blocks of k elements each. The elements of each block are randomly permuted, for a total of at most another $\frac{nk}{2}$ inversions. A value of $k = 0$ means that the input sequence is sorted in ascending order. A small value of k, with respect to n, means that the input sequence is almost sorted. A value of k, which is as big as n, means that the input sequence is randomly sorted. The experiment is repeated 100 times for a different value of k and the average number of comparisons performed by each of the three algorithms is reported verses $\log k$. The experiments are performed on two values of n; for Table 1 $n = 1024$, and for Table 2 $n = 32768$.

The experiments are repeated with the input sequence reversed before being fed to the program. In other words, a value of $k = 0$ would now mean that the input sequence is inversely sorted. A small value of k means that the input sequence is almost inversely sorted. See Table 3.

The results of our experiments imply that Algorithm 2 always performs a fewer number of comparisons than Splaysort. It is also doing better than Binomialsort, except when the number of inversions is small. This suggests that the constant hidden in the linear term of the number of comparisons used by

Table 1. Comparisons per item to sort random sequences of n=1024. $Inv(X) < kn$

$\log k$	0	1	2	3	4	5	6	7	8	9	10
Algorithm 2	3.2	3.2	3.5	3.9	4.7	5.6	6.5	8.0	8.7	10.8	11.1
Splaysort	2.0	3.1	3.9	5.0	5.7	6.8	8.4	10.4	11.8	13.7	15.5
Binomialsort	1.9	2.3	2.3	3.4	4.2	5.7	6.6	8.6	10.4	11.9	12.8

Table 2. Comparisons per item to sort random sequences of n=32768. $Inv(X) < kn$

$\log k$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Algorithm 2	3.3	3.3	3.5	3.9	4.5	5.1	6.0	6.9	7.8	8.8	9.9	11.1	12.5	14.0	15.1	17.3
Splaysort	2.0	3.2	4.4	4.9	5.8	6.5	7.6	8.8	10.2	11.7	13.3	15.1	16.9	18.9	20.8	25.2
Binomialsort	2.0	2.4	2.6	3.2	4.2	5.8	7.2	8.7	10.4	12.1	14.7	19.0	21.0	22.3	23.1	23.8

Table 3. Comparisons per item to sort random sequences of n=65536. $Inv(Rev(X)) \leq kn$, where $Rev(X)$ is the reversed sequence of X.

$\log k$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Algori. 2	1.5	1.6	2.4	3.4	4.3	5.2	6.1	7.0	7.9	8.9	10.1	11.0	12.5	13.6	14.8	17.5
Splay	2.0	3.1	3.87	5.1	5.8	6.6	7.6	8.8	10.1	11.5	13.2	14.9	16.7	18.7	21.3	23.2
Binomial	27.7	27.7	27.7	27.6	27.6	27.5	27.5	27.4	27.4	27.4	27.3	27.1	26.9	26.6	26.3	25.6

Binomialsort is smaller. When the input is almost inversely sorted, in contrast with Binomialsort, both Splaysort and Algorithm 2 are still adaptive.

5 Algorithm 3

The algorithm starts with building a binomial queue in the input array, by mapping every element of the binomial queue to a corresponding location in the array. The algorithm proceeds (similar to Heapsort) by repeatedly swapping the element that is the current minimum (first element of the array) with the last element of the unsorted part of the array. Each time, a *heapify* operation is performed on the binomial queue to maintain the heap property.

Heapifying Binomial Queues

Given an n-node binomial queue such that the value at its root is not the smallest value, we want to restore the heap property. The *heapify* operation proceeds by finding the node x with the smallest value among the children of the root and swapping its value with that of the root. This step is repeated with the node x as the current root, until either a leaf or a node that has a value smaller than or equal to all the values of its children is reached. For efficient implementation, an extra pointer is kept with every node x. This pointer points to the node with the smallest value among all the right siblings of x, including itself. We call this pointer, the pointer for the prefix minimum (pm). The pm pointer of the leftmost child of a node will, therefore, point to the node with the smallest value among all the children of the parent node. First, the path from the root to a leaf, where every node has the smallest value among its siblings, is determined by utilizing the pm pointers. No comparisons are required for this step. Next, the value at the root is compared with the values of the nodes of this path bottom up, until the correct position of the root is determined. The value at the root is then inserted

at this position, and all the values at the nodes above this position are shifted up. The *pm* pointers of the nodes whose values moved up and those of all their left siblings are updated. To maintain the correct values in the *pm* pointers, the *pm* pointer of a given node x is updated to point to the smaller of the value of x and the value of the node pointed to by the *pm* pointer of the right sibling of x. At each level of the queue (except possibly for the level of the final destination of the old value of the root), either a comparison with the old value of the root takes place or the *pm* pointers are updated, but not both. See [4] for the details.

Lemma 5 *The time used by the heapify operation is* $O(\log n)$. *It requires at most* $\lceil \log n \rceil + 1$ *comparisons, and an* $O(\log n)$ *additional storage.*

Building a Binomial Queue in an Array

Given an n-element array, we build a binomial queue of rank i if the *ith* bit in the binary representation of n is 1. The smaller rank queues are mapped first in the array. The nodes of a binomial queue are mapped in a preorder fashion (the root is mapped to the first position of the array). The order in which the sub-queues are mapped is from right to left (right sub-queues first).

Being aware of the rank of a node of a binomial queue that is in location p in the array, the location of its right or left siblings as well as its leftmost child can be determined, as follows. If the rank of this node is r, the locations of its right sibling, left sibling and leftmost child will be $p - 2^{r-1}$, $p + 2^r$ and $p + 2^{r-1}$ respectively. During the onset of the algorithm, the sorted part is stored in the last locations of the array. Some nodes on the left spine of the largest queue will be losing their leftmost children. Hence the formula for the leftmost child for these nodes will be $p + 2^j$, where j is the largest integer less than or equal to $r - 1$ such that $p + 2^j$ is smaller than the boundary for the unsorted part of the array. The *pm* pointers are stored in the form of a number of bits per node that represents the difference in rank between the source node and the node it is pointing to. If the rank of the source node is r, its location in the array is p, and the value stored for the *pm* pointer is d, then the location of the node, that this *pm* pointer is pointing to, is $p - 2^r + 2^{r-d}$. The total number of bits representing these pointers is less than n. An initial binomial queue can be built in an array in a recursive manner. Given two binomial queues of rank $\log n - 1$ stored in the first and last $\frac{n}{2}$ locations of an array, the two queues are merged by comparing their roots and performing an $O(n)$ moves, if necessary, to maintain the above mapping criteria. The bits representing the *pm* pointer of the node that loses the comparison is calculated.

Lemma 6 *Algorithm 3 runs in* $O(n \log n)$ *and requires at most* $n \log n + 1.6n$ *comparisons. In addition to the input array, less than n extra bits are used.*

Proof. The phase of building the initial binomial queue in the array requires $n-1$ comparisons to build the queue, and $\frac{n}{2}$ comparisons to set the *pm* pointers (there

are $\frac{n}{2}$ nodes that do not have right siblings). The number of moves done in this initial phase is $O(n \log n)$. A *heapify* operation is then applied $n-1$ times on the remaining unsorted elements. Using Lemma 5, the total number of comparisons needed in these operations is bounded by $\sum_{1 \leq k \leq n}(\lceil \log k \rceil + 1) = B(n) + n < n \log n + .1n$ (See [20] for the derivation of the bound on $B(n)$.). The bound on the number of comparisons follows by adding the above bound with the $1.5n$ comparisons of the initial phase. To represent the pm pointers, we need at most $\lceil \log i \rceil$ bits for each of at most $\frac{n}{2^i}$ of the nodes, for all i from 2 to $\log n$. Hence, the total number of bits is bounded by $\sum_{2 \leq i \leq \log n} \frac{\lceil \log i \rceil}{2^i} n < n$. □

6 A Lower Bound on Sorting Using Priority Queues

There are two main paradigms for sorting using priority queues. The first paradigm, which we used in the first two algorithms, maintains the elements to be sorted in a set of priority queues. It permits comparisons only between the elements of the roots of these priority queues. After such a comparison, the root that has the larger element is linked to the other root (becomes one of its children). Comparisons are performed until all the elements are combined into one queue. The root of this queue is removed and the smallest element is output, again leaving a set of priority queues. The process of combining the queues is repeated to produce the second smallest element, and so on. To establish a lower bound on such paradigm, we adopt the adversary that whenever the roots of two queues are compared, the queue with the smaller number of nodes becomes a child of the root of the other queue. The number of comparisons that a specific node wins is exactly the number of children of this node at the moment when this node is deleted as the smallest element. If the size of the combined queue at this time is i, the number of children of the root is at least $\lceil \log i \rceil$. Therefore, the total number of comparisons required by any sorting algorithm that uses this paradigm is at least $\sum_{1 \leq k \leq n} \lceil \log k \rceil = n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$.

The second paradigm, which we used in the third algorithm, uses a system of promotions. In such a paradigm, an initial priority queue structure is built using the first paradigm, the element in the root is deleted, and promotions start taking place. The promotions involve comparisons between elements in the sibling nodes, and the smallest among them is promoted to replace the vacant parent. We show next that this paradigm inherits the same rules as the first paradigm, and hence the same lower bound applies. More specifically, a feature of the first paradigm is that for two elements x and y to be compared, both x and y should have either never lost a comparison or otherwise the last time each of them has lost a comparison should have been to the same element. In a system of promotions, assume for the purpose of contradiction that x and y are to be compared together, and that the last time x lost the comparison to a and y lost the comparison to b. Now, for x and y to be compared, a and b should have been compared first. Assume without loss of generality that a wins with respect to b. It follows that x is to be compared with b. Since x is not to lose another comparison after it has lost to a, it follows that x wins with respect to

b (i.e. $x < b$). Since y lost the comparison to b (i.e. $b < y$), it follows that $x < y$. This precludes the possibility of x and y being compared. A contradiction!

Indeed, the two paradigms are equivalent and the stated lower bound applies for any algorithm that uses a mixture of these two paradigms.

References

1. R. Cole. *On the dynamic finger conjecture for splay trees. Part II: The proof.* SIAM J. Comput. 30 (2000), 44–85.
2. R. Dutton. *Weak-Heapsort.* BIT, 33 (1993), 372–381.
3. S. Edelkamp and I. Wegener. *On the performance of weak-heapsort.* STACS 2000.In LNCS 1770 (2000), 254–260.
4. A. Elmasry. *Priority queues, pairing and adaptive sorting.* 29th ICALP. In LNCS 2380 (2002), 183–194.
5. V. Estivill-Castro and D. Wood. *A survey of adaptive sorting algorithms.* ACM Comput. Surv. 24(4) (1992), 441–476.
6. R. Floyd. *ACM algorithm 245: Treesort 3.* Comm. of ACM, 7(12) (1964), 701.
7. M. Fredman. *A priority queue transform.* 3rd WAE, LNCS 1668 (1999), 243–257.
8. M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. *The pairing heap: a new form of self_adjusting heap.* Algorithmica 1,1 (1986), 111–129.
9. J. Katajainen, T. Pasanen. *In-place sorting with fewer moves.* Information Processing Letters, 70(1) (1999), 31–37.
10. J. Katajainen, T. Pasanen and J. Teuhola. *Practical in-place Mergesort.* Nordic Journal of Computing, 3(1) (1996), 27–40.
11. D. Knuth. *The Art of Computer Programming. Vol III: Sorting and Searching.* Addison-wesley, second edition (1998).
12. C. Levcopoulos and O. Petersson. *Adaptive Heapsort.* J. of Alg. 14 (1993), 395–413.
13. H. Mannila. *Measures of presortedness and optimal sorting algorithms.* IEEE Trans. Comput. C-34 (1985), 318–325.
14. C. McDiarmid and B. Reed. *Building heaps fast.* J. of Alg. 10 (1989), 352-365.
15. A. Moffat, G. Eddy and O. Petersson. *Splaysort: fast, verstile, practical.* Softw. Pract. and Exper. Vol 126(7) (1996), 781–797.
16. K. Reinhardt. *Sorting in-place with a worst case complexity of* $n \log n - 1.3n + O(\log n)$ *comparisons and* $\epsilon n \log n + O(1)$ *transports.* LNCS (650) (1992), 489–499.
17. S. Roura. *Improving Mergesort for linked lists.* 7th ESA (1999), 267–276.
18. D. Sleator and R. Tarjan. *Self-adjusting binary search trees.* J. ACM 32(3) (1985), 652–686.
19. R. Tarjan. *Amortized computational complexity.* SIAM J. Alg. Disc. Meth. 6 (1985), 306-318.
20. I. Wegener. *The worst case complexity of McDiarmid and Reed's variant of Bottom-up Heapsort is less than* $n \log n + 1.1n$. Inform. and Comput., 97(1) (1992), 86–96.
21. I. Wegener. *Bottom-up-Heapsort, a new variant of Heapsort, beating on an average, Quicksort (if n is not very small).* Theor. Comp. science (118) (1993), 81–98.
22. J. Williams. *ACM algorithm 232: Heapsort.* Comm. of ACM 7(6) (1964), 347–348.