Melding priority queues *

Ran Mendelson † *Robert E. Tarjan* ‡ *Mikkel Thorup* § *Uri Zwick* †

June 11, 2004

Abstract

We show that any priority queue data structure that supports *insert*, *delete*, and *find-min* operations in $pq(n)$ time, when n is an upper bound on the number of elements in the priority queue, can be converted into a priority queue data structure that also supports fast *meld* operations with essentially no increase in the amortized cost of the other operations. More specifically, the new data structure supports *insert*, *meld* and *find-min* operations in $O(1)$ amortized time, and *delete* operations in $O(pq(n) + \alpha(n, n))$ amortized time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function. The construction is very simple, essentially just placing a non-meldable priority queue at each node of a union-find data structure. We also show that when all keys are integers in the range $[1, N]$, we can replace n in the bound stated above by $\min\{n, N\}$.

Applying this result to non-meldable priority queue data structures obtained recently by Thorup, and by Han and Thorup, we obtain meldable RAM priority queues with $O(\log\log n)$ amortized cost per operation, or $O(\sqrt{\log\log n})$ expected amortized cost per operation, respectively. As a by-product, we obtain improved algorithms for the minimum directed spanning tree problem in graphs with integer edge weights: A deterministic $O(m \log\log n)$ time algorithm and a randomized $O(m\sqrt{\log\log n})$ time algorithm. These bounds improve, for sparse enough graphs, on the $O(m + n \log n)$ running time of an algorithm by Gabow, Galil, Spencer and Tarjan that works for arbitrary edge weights.

Key Words: Priority queues, heaps, union-find, word RAM model, optimum branchings, minimum directed spanning trees.

1 Introduction

Priority queues are basic data structures used by many algorithms. The most basic operations, supported by all priority queues, are *insert*, which inserts an element with an associated key into the priority queue, and *extract-min*, which returns the element with the smallest key currently in the queue, and deletes it. These two operations can be used, for example, to sort n elements by performing n *insert* operations followed by n *extract-min* operations. Most priority queues also support a *delete* operation, that deletes a given element, not necessarily with the minimum key, from the queue, and *find-min*, which finds, but does not delete, an element with minimum key.

Using the *insert* and *delete* operations we can easily implement a *decrease-key* operation, or more generally a *change-key* operation, that decreases, or arbitrarily changes, the key of a queue element. (We simply delete the element from the queue and re-insert it with its new key.) As the *decrease-key* operation is the bottleneck operation in efficient implementations of Dijkstra's single-source shortest paths algorithm [9], and Prim's algorithm [25] for finding a minimum spanning tree, many priority queues support this operation directly, sometimes in constant time.

The efficient implementation of several algorithms, such as the algorithm of Edmonds [10] for computing optimum branching and minimum directed spanning trees, require the maintenance of a collection of priority queues. In

*This paper is the combined journal version of [23] and [22].

†School of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel. E-Mail: {ranm,zwick}@cs.tau.ac.il.

‡Department of Computer Science, Princeton University, Princeton, NJ 08540, USA and Hewlett Packard, Palo Alto, CA 94304, USA.
Research at Princeton University partially supported by the Aladdin project, NSF Grant CCR-9626862. E-mail: ret@cs.princeton.edu.

§AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932, USA. E-mail: mthorup@research.att.com.

addition to the standard operations performed on individual priority queues in this collection, these algorithms also need, quite often, to *meld*, or unite, two priority queues from this collection. This provides a strong motivation for studying *meldable* priority queues.

Fibonacci heaps, developed by Fredman and Tarjan [11], are very elegant and efficient meldable priority queues. They support *delete* operations in $O(\log n)$ *amortized* time, and all other operations, including meld operations, in $O(1)$ amortized time, where n is the size of the priority queue from which an element is deleted. (For a general discussion of amortized time bounds, see [29].) Brodal [4] obtained a much more complicated data structure that supports *delete* operations in $O(\log n)$ *worst-case* time, and all other operations in $O(1)$ worst-case time. Both these data structures are comparison-based and can handle elements with arbitrary real keys. In this setting they are asymptotically optimal.

While $O(\log n)$ is the best delete time possible in the comparison model, much better time bounds can be obtained in the word RAM model of computation, as was first demonstrated by Fredman and Willard [12, 13]. In this model each key is assumed to be an integer that fits into a single word of memory. Each word of memory is assumed to contain $w \geq \log n$ bits. The model allows random access to memory, as in the standard RAM model of computation. The set of basic word operations that can be performed in constant time are the standard word operations available in typical programming languages (e.g., C): addition, multiplication, bit-wise and/or operations, shifts, and their like.

Thorup [31, 32] obtained a general equivalence between priority queues and sorting. More specifically, he showed that if up to n elements can be sorted in $O(n \cdot s(n))$ time, then the basic priority queue operations can be implemented in $O(s(n))$ time. Using a recent $O(n \log \log n)$ sorting algorithm of Han [15], this gives priority queues that support all operations in $O(\log \log n)$ time. Thorup [34] extends this result by presenting a priority queue data structure that supports *insert*, *find-min* and *decrease-key* operations in $O(1)$ time and *delete* operations in $O(\log \log n)$ time. (This result is not implied directly by the equivalence to sorting.) Han and Thorup [16] obtained recently a randomized $O(n\sqrt{\log \log n})$ time sorting algorithm. This yields basic priority queues that support all operations in $O(\sqrt{\log \log n})$ expected amortized time.

Alstrup *et al.* [1] describe a simple transformation that show that any priority queue data structure that supports the basic priority queue operations in $O(pq(n))$ time can be converted into a data structure that supports *insert* and *find-min* operations is $O(1)$ amortized time and *delete* operations is $O(pq(n))$ amortized time.

1.1 Adding a meld operation

The fast priority queues mentioned above, with the $O(\log \log n)$ time or $O(\sqrt{\log \log n})$ expected time per operation, do *not* support meld operations. Our main result is a general transformation that takes these priority queues, or any other priority queue data structure, and produces new priority queue data structures that support *meld* operations in constant amortized time, without increasing the amortized cost of the other priority queue operations! More specifically, we show that any priority queue data structure that supports *insert*, *delete*, and *find-min* operations in $pq(n)$ time, where n is the number of elements in the priority queue, can be converted into a priority queue data structure that supports *insert*, *meld* and *find-min* operations in $O(1)$ amortized time, and *delete* operations in $O(pq(n) + \alpha(n, n))$ amortized time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function (see [26]). The function $\alpha(n, n)$ is an extremely slow growing function. In particular, $\alpha(n, n) = o(\log^* n)$.

Applying our generic transformation to the non-meldable priority queues mentioned above, we get meldable priority queues that support *insert*, *meld* and *find-min* operations in $O(1)$ amortized time, and *delete* operations in $O(\log \log n)$ amortized time, or $O(\sqrt{\log \log n})$ expected amortized time.

In combination with Thorup's [34] reduction from non-meldable priority queues to sorting, we get that if there is a sorting algorithm that can sort up to n elements in $O(n \cdot s(n))$ time, then there are also meldable priority queues supporting *insert*, *meld* and *find-min* operations in $O(1)$ amortized time, and *delete* operations in $O(s(n) + \alpha(n, n))$ amortized time.

Our transformation, which constructs meldable priority queues using non-meldable ones, is very simple and natural. Essentially, it just places a non-meldable basic priority queue at each node of a union-find data structure. (A very similar idea is used by van Emde Boas *et al.* [37], see below.) If the non-meldable priority queue placed at each node of the union-find data structure supports delete operations in $O(pq(n))$ time, then the amortized delete time of the obtained data structure is $O(pq(n)\,\alpha(n, n/pq(n)))$. To get the slightly improved bound of $O(pq(n) + \alpha(n, n))$, we need to replace the small non-meldable priority queues that appear at the bottom of the union-find trees by the *atomic heaps* of Fredman and Willard [13] which have a constant operation time, but can hold only $O(\log^2 n)$ elements. We note, however, that the factor $\alpha(n, n/pq(n))$ is *constant* for all conceivable values of $pq(n)$, e.g., for $pq(n) = \Omega(\log^* n)$.

1.2 Machine models

Our transformation uses the non-meldable priority queues supplied to it as black boxes. In the basic, combinatorial, version of our transformation, in which atomic heaps are not used, the machine model used should just be strong enough to implement the non-meldable priority queues supplied to it. All other operations can be carried out on a pointer machine. As mentioned above, the $O(pq(n)\,\alpha(n, n/pq(n)))$ bound obtained in this case may be considered satisfactory, at least until priority queues with, say, $o(\log^* n)$ time per operation are obtained, if that is at all possible.

As mentioned, the amortized delete time of the meldable priority queues obtained using our transformation can be reduced to $O(pq(n) + \alpha(n, n))$ by using the atomic heaps of Fredman and Willard [13]. The machine model used should then be strong enough to implement both the non-meldable priority queues supplied to the transformation, and atomic heaps. The implementation of atomic heaps is known to require the use of multiplication, or non-standard AC^0 operations (see discussion in Thorup [35]).

To demonstrate the difference between the $O(pq(n)\,\alpha(n, n/pq(n)))$ and $O(pq(n) + \alpha(n, n))$ bounds, we note that if, some time in the future, non-meldable priority queues with an operation time of $O(\alpha(n, n))$ are obtained, then the operation time of the meldable priority queues that do not use atomic heaps would be $O((\alpha(n, n))^2)$, while with atomic heaps, the operation time would be reduced to $O(\alpha(n, n))$.

The sorting algorithms of Han [15] and Han and Thorup [16] mentioned above also require multiplication. Thorup [33], on the other hand, presents a randomized sorting algorithm with $O(n \log\log n)$ expected time that uses only addition, shifts, and bit-wise Boolean operations. Han and Thorup [16] present such deterministic sorting algorithms with a running time of $O(n(\log\log n)^{1+\varepsilon})$, for any fixed $\varepsilon > 0$. Combined with our combinatorial reduction, we thus get meldable priority queues with $O((\log\log n)^{1+\varepsilon})$ amortized operation time, or $O(\log\log n)$ expected amortized time, on a machine equipped only with addition, shifts, and bit-wise Boolean operations.

It is also instructive to note that if we start with a comparison based $O(n \log n)$ time sorting algorithm, such as heap-sort, merge-sort or a deterministic version of quick-sort (see, e.g., [8]), then using the reduction of Thorup [32] we get comparison-based basic priority queues with an operation time of $O(\log n)$. Using our transformation, we then get comparison-based meldable priority queues with an amortized operation time of $O(\log n)$. These priority queues can be implemented on a pointer machine. (The Binomial heaps of Vuillemin [38] are, of course, better as they support these operations in $O(\log n)$ worst case time, and the heaps of Brodal [4] are even better as they support *decrease-key* and all other operations, except *delete*, in constant time.)

To complete the picture we should mention that the transformation of Thorup [32] from sorting algorithms to priority queues also uses atomic heaps. Thorup [32] also describes, however, an alternative combinatorial transformation that can be implemented on a pointer machine. This version of the transformation transforms an algorithm that sorts up to n keys in $O(n \cdot s(n))$ time into a basic priority queue data structure with a delete time of $O(pq_s(n))$, where the function $pq_s(n)$ satisfies the recursion $pq_s(n) = O(pq_s(\log^2 n) + s(n))$. We note that $pq_s(n) = O(s(n))$ if $s(n) = \log^{(i)} n$, for any $i \geq 1$, i.e., if $s(n)$ is any iterated log-function. However, $pq_s(n) = \Omega(\log^* n)$ even if $s(n) = O(1)$, so in combination with our own combinatorial reduction, we get a meldable priority queue data structure with an amortized delete time of $O(pq_s(n)\,\alpha(n, n/pq_s(n))) = O(pq_s(n))$.

1.3 Improvement for smaller integer keys

We also describe a second independent transformation that shows that a meldable priority queue data structure that supports all operations in $pq(n)$ time, where n is a bound on the number of elements in the priority queues, can be used to construct a priority queue data structure that supports *insert*, *meld* and *find-min* operations in $O(1)$

amortized time, and *delete* operations in $O(pq(\min\{N,n\}))$ amortized time, when all keys are integers from the range $[1,N]$.

In conjunction with the first transformation, this allows us, for example, to add a meld operation, with constant amortized cost, to the priority queue of van Emde Boas [36, 37] which has $pq(n) = O(\log\log N)$. The amortized cost of a delete operation is then:

$$\begin{aligned} & O(\log\log N \cdot \alpha(\min\{n,N\}, \min\{n,N\}/\log\log N)) \\ =\ & O(\log\log N \cdot \alpha(N, N/\log\log N)) \ =\ O(\log\log N)\ . \end{aligned}$$

(The original data structure of van Emde Boas requires randomized hashing to run in linear space [21]. A deterministic version is presented in [32].) This improves an $O(\log\log N\,\alpha(n,n))$ bound obtained by van Emde Boas *et al.* [37], and matches a $O(\log\log N)$ bound obtained by Bright [3] (see below).

We can also use this second transformation to obtain a meldable priority queues that support *insert*, *meld* and *find-min* operations in $O(1)$ amortized time, and *delete* operations in $O(\sqrt{\log\log N})$ expected amortized delete time.

1.4 Optimum branchings and minimum directed spanning trees

The problem of finding a minimum (or maximum) spanning tree in an *undirected* graph is an extremely well studied problem. Chazelle [6] obtained a deterministic $O(m\,\alpha(m,n))$ time algorithm for the problem. An asymptotically optimal algorithm for finding minimum spanning trees, with an unknown running time, was given by Pettie and Ramachandran [24]. Karger *et al.* [19] obtained a randomized algorithm that runs, with very high probability, in $O(m+n)$ time. All these algorithms are comparison based and can handle arbitrary real edge weights. Fredman and Willard [13] obtained a deterministic $O(m+n)$ time algorithm for the problem in the word RAM model.

The directed version of the minimum spanning tree problem received much less attention in recent years. This version comes in two, or in fact three, variants. We are either given a root r and asked to find a directed spanning tree of minimum weight rooted at r, or we are asked to find a directed spanning tree of minimum weight rooted at an arbitrary vertex. (It is assumed, in both cases, that the desired directed spanning trees do exist.) A very closely related problem is the problem of finding an *optimum branching*, i.e., a branching of maximum total weight. A branching B in a directed graph is a collection of edges that satisfies the following two properties: (i) B does not contain a cycle; (ii) No two edges of B are directed into the same vertex. It is not difficult to show that these three versions are essentially equivalent. We refer to the three of them collectively as the *minimum directed spanning tree* problem. All results stated in this paper apply to all three versions.

Chu and Liu [7], Edmonds [10] and Bock [2] independently obtained an essentially identical polynomial time algorithm for the MDST problem. (In the sequel we refer to this paper, somewhat unfairly, as Edmonds' algorithm.) Karp [20] gave a simple formulation of the algorithm and a direct combinatorial proof of its correctness. All subsequent results, including ours, are just more efficient implementations of variants of this algorithm using improved *meldable* priority queue data structures.

Tarjan [27] (see also Camerini *et al.* [5]) describes a natural way of implementing the algorithm of Edmonds using a meldable priority queue data structure. The complexity of the algorithm is dominated by the cost of performing a sequence of at most $O(m)$ *insert* operations, $O(m)$ *extract-min* operations, $O(n)$ *meld* operations, and $O(n)$ *add* operations. An *add* operations adds a given constant to keys of all elements contained in a certain priority queue. Adding an *add* operation to existing priority queue data structures is not a difficult task. (For the details, see [27].) In particular, it is not difficult to augment the priority queues obtained using our transformations with a constant time *add* operation. Using simple meldable priority queues with $O(\log n)$ time per operation, Tarjan [27] obtains an $O(\min\{m\log n, n^2\})$ algorithm for the problem.

Gabow *et al.* [14] give a more sophisticated algorithm that implements a variant of Edmonds's algorithm using at most $O(n)$ *insert* operations, $O(n)$ *extract-min* operations, $O(n)$ *delete* operations, $O(n)$ *add* operations, and finally $O(m)$ *move* operations. A *move* is a non-standard priority queue operation that moves an element from one priority queue to another. There are no known constructions that support such an operation in constant time, without severely deteriorating the cost of the other operations. Although Fibonacci heaps do not support a general move operation in constant time, Gabow *et al.* [14] show that in the special context of their algorithm, the required move operations may be implemented in constant time. As a result, they obtain an $O(m + n \log n)$ algorithm for the minimum directed spanning tree problem.

Our improved RAM algorithms for the minimum directed spanning tree problem are obtained by plugging in our improved meldable priority queues into Tarjan's [27] implementation of Edmonds' algorithm. We thus obtain a deterministic $O(m \log\log n)$ time algorithm and the randomized $O(m\sqrt{\log\log n})$ expected time algorithm.

We were not able to augment our improved meldable priority queues with a fast *move* operation required by the approach of Gabow *et al.* [14]. Obtaining algorithms with a running time of the form $O(m + nf(n))$, where $f(n) = o(n \log n)$, remains a challenging open problem.

1.5 Relation to previous work

The basic idea of placing an non-meldable priority queue in each node of a union-find data goes back to van Emde Boas *et al.* [37]. The analysis given there, however, is not tight. As a result, it is only shown there that the meldable priority queues obtained have an amortized operation cost of $O(\log\log N\,\alpha(n, n))$. (As noted above, our improved analysis reduces the amortized operation cost to $O(\log\log N)$.)

Meldable priority queues with an amortized operation cost of $O(\log\log N)$ were already obtained, using a different technique, by Bright [3]. Although his construction is specialized, his ideas can be used to obtain a transformation that combines a non-meldable priority queue data structure, with an operation cost of $pq(n)$, and a meldable priority queue data structure, with an operation cost of $mpq(n)$, to produce a meldable priority queue data structure with an amortized operation cost of $pq(n) + mpq(\log^2 n) + O(1)$. It is of course possible to apply this transformation recursively, but the results obtained are weaker than the results obtained using the transformation of van Emde Boas *et al.* [37], with the improved analysis presented here.

Quite recently, overlooking the results cited above, three of the current authors [23] rediscovered the idea of combining a non-meldable priority queue data structure with the union-find data structure to obtain a meldable priority queue data structure and described it as a general transformation and mentioning the application to optimum branchings and minimum directed spanning trees. The analysis given in [23] was again not tight. Shortly thereafter, the current set of authors [22] obtained the improved analysis giving an amortized delete time of $O(pq(n) + \alpha(n, n))$, or $O(pq(n) + \alpha(n, n/pq(n)))$ without using atomic heaps, and a constant amortized cost for all the other operations.

In [23], we also presented a meldable priority queue data structure that supports *decrease-key* operations in constant time, but has a much slower expected amortized delete time of $O((\log n)^{1/2+\varepsilon})$, for every $\varepsilon > 0$. However, after discovering Bright's paper [3], we noticed that his transformation produces meldable priority queues that support *decrease-key* operations in constant amortized time, if the priority queues supplied to it also have this property. Thus, by combining the non-meldable priority queues of Thorup [34], which support *decrease-key* operations in constant time, with the Fibonacci heaps of Fredman and Tarjan [11], which also do so, we obtain meldable priority queues that support *insert*, *meld*, *find-min* and *decrease-key* operations in constant amortized time, and *delete* operations in $O(\log\log n)$ time. Since we are not aware of any applications of meldable priority queues with constant time *decrease-key* operations, we do not pursue the matter any further.

This paper is the combined journal version of [23] and [22]. The main result of the paper is an improved analysis of the transformation, first suggested by van Emde Boas *et al.* [37], that converts non-meldable priority queues into meldable ones. The improved analysis shows that a constant time *meld* operation can be added essentially for free! I.e., without affecting the amortized cost of the other operations. We also obtain a new transformation that produces extremely fast range restricted priority queues.

1.6 Organization of paper

The rest of this paper is organized as follows. In the next we review the classical union-find data structure and prove a simple lemma on its behavior. As mentioned, our meldable priority queues are obtained by 'planting' a non-meldable priority queue at each node of the union-find data structure. In Section 3 we describe the transformation from non-meldable priority queues into meldable ones. In Section 4 we present our improved analysis of the transformation. In Section 5 we present the slightly improved version of the transformation that employs atomic heaps. In Section 6

make-set(x) :	*link*(x,y) :	*find*(x) :
$p[x] \leftarrow x$	if $rank[x] > rank[y]$	if $p[x] \neq x$
$rank[x] \leftarrow 0$	then $p[y] \leftarrow x$	then $p[x] \leftarrow find(p[x])$
	else $p[x] \leftarrow y$	return $p[x]$
union(x,y) :	if $rank[x] = rank[y]$	
link(*find*(x), *find*(y))	then $rank[y] \leftarrow rank[y] + 1$	

Figure 1: The classical union-find data structure

we describe an independent simple transformation that allows us to obtain time bounds that depend only on N, the maximum key value, and not on n, the number of elements in the priority queue. We end in Section 7 with some concluding remarks and open problems.

2 The Union-find data structure

A union-find data structure supports the following operations:

make-set(x)	–	Create a set that contains the single element x.
union(x,y)	–	Unite the sets containing the elements x and y.
find(x)	–	Return a representative of the set containing the element x.

A classical, simple, and extremely efficient implementation of a union-find data structure is given in Figure 1. Each element x has a parent pointer $p[x]$ and a rank $rank[x]$ associated with it. The parent pointers define trees that correspond to the sets maintained by the data structure. The representative element of each set is taken to be the root of the tree containing the elements of the set. To find the representative element of a set, we simply follow the parent pointers until we get to a root. To speed-up future *find* operations, we employ the *path compression* heuristic that makes all the vertices encountered on the way to the root direct children of the root. Unions are implemented using the *union by rank* heuristic. The rank $rank[x]$ associated with each element x is an upper bound on the depth of its subtree.

In a seminal paper, Tarjan [26] showed that the time taken by the algorithm of Figure 1 to process an intermixed sequence of m *make-set*, *union* and *find* operations, out of which n are *make-set* operations, is $O(m\,\alpha(m,n))$, where $\alpha(m,n)$ is the extremely slowly growing inverse of Ackermann's function, which we define next.

Ackermann's function is an extremely fast growing function which has many essentially equivalent definitions. One of them, which is taken from Tarjan [26], is:

$$\begin{array}{rcll} A(0,j) & = & 2j & , \text{ for } j \geq 1 \\ A(i,1) & = & 2 & , \text{ for } i \geq 1 \\ A(i,j) & = & A(i-1, A(i,j-1)) & , \text{ for } i \geq 1,\ j \geq 2 \end{array}$$

The inverse Ackermann function $\alpha(m,n)$ is then defined as follows:

$$\alpha(m,n) \;=\; \min\{\, i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log_2 n \,\} \,.$$

A slightly better bound on the number of operations performed by the union-find algorithm was obtained by Tarjan and van Leeuwen [30].

Theorem 2.1 (Tarjan and van Leeuwen [30]) *The union-find algorithm of Figure 1 processes an intermixed sequence of n make-set operations, up to n link operations, and f find operations in $O(n + f\,\alpha(f+n,n))$ time.*

The analysis of the next section relies on the following lemma:

Lemma 2.2 *Suppose that an intermixed sequence of n make-set operations, at most n link operations, and at most f find operations are performed on the standard union-find data structure. Then, the number of times the parent pointers of elements of rank k or more are changed is at most* $O(\frac{n}{2^k} + f\,\alpha(f + \frac{n}{2^k}, \frac{n}{2^k}))$.

Proof: As each rank k element has at least 2^k descendants, and each element has at most one ancestor of rank k, the number of elements of rank k is at most $n/2^k$. Let us say a node x is *high* if $rank[x] \geq k$, and *low*, otherwise. Note that all ancestors of high nodes are high and all descendants of low nodes are low. As each high element had rank k, at some stage, it follows that the number of high elements is also at most $n/2^k$. We consider each one of the trees formed by the union-find data structure to be composed of a (possibly empty) top part containing high nodes, and a bottom part containing low nodes.

We claim that the operations performed on the top parts of the trees correspond to a sequence of at most f *find* operations, and at most $n/2^k$ *link* operations, on the $n/2^k$ high elements. By Theorem 2.1, the number of pointers changed during these operations is at most $O(\frac{n}{2^k} + f\,\alpha(f + \frac{n}{2^k}, \frac{n}{2^k}))$, as required.

The corresponding sequence of operations performed on the top parts of the trees is obtained in the following way. When the rank of an element x becomes k, we add a *make-set*(x) operation to the constructed sequence. When a *link*(x, y) operation is performed, where both x and y are high, we add a *link*(x, y) operation to the constructed sequence. When a *find*(x) operation from the original sequence of operations returns a high element, we add a *find*(x') operation to the constructed sequence, where x' is the first high element on the path from x to its root before the *find*(x) operation. The constructed sequence contains at most $n/2^k$ *make-set* operations, at most $n/2^k$ *link* operations and at most f *find* operations, as required.

It is easy to see that when the union-find data structure is used to process the constructed sequence of operations on the high nodes, the operations performed are exactly those performed on the high elements while processing the original sequence of operations. This completes the proof of the lemma. □

We note in passing that the $O(m\alpha(m, n))$ bound of Tarjan [26] is sufficient for proving Lemma 2.2. This Lemma, in turn, can be used to obtain the slightly tighter bound of Tarjan and van Leeuven [30] (Theorem 2.1 above) on which we rely in Section 5. A further refinement of the analysis of the union-find data structure, by Kaplan *et al.* [18], is mentioned at the end of Section 4.

3 The transformation

In this section we describe a transformation that combines a non-meldable priority queue data structure with the classical union-find data structure (see [26], [28], or [8]) to produce a *meldable* priority queue data structure with essentially no increase in the amortized operation cost. As mentioned, this transformation was first described in van Emde Boas *et al.* [37], though it is presented there in a somewhat specialized setting. An improved analysis of this transformation appears in the next section.

The transformation $\mathcal{T}$ receives a non-meldable priority queue data structure $\mathcal{P}$ and produces a meldable priority queue data structure $\mathcal{T}(\mathcal{P})$. We assume that the non-meldable data structure $\mathcal{P}$ supports the following operations:

make-pq(x)	– Create a priority queue that contains the single element x.
insert(PQ, x)	– Insert the element x into the priority queue PQ.
delete(PQ, x)	– Delete the element x from the priority queue PQ.
find-min(PQ)	– Find an element with the smallest key contained in PQ.

It is assumed, of course, that each element x has a key $key[x]$ associated with it. We can easily add the following operation to the repertoire of the operations supported by this priority queue:

change-key(PQ, x, k) – Change the key of element x in PQ to k.

This is done by deleting the element x from the priority queue PQ, changing its key by setting $key[x] \leftarrow k$, and then reinserting it into the priority queue. (Some priority queues directly support operations like *decrease-key*. We shall not assume such capabilities here.)

```
MAKE-PQ(x) :
p[x] ← x
rank[x] ← 0
PQ[x] ← make-pq(x)

INSERT(x, y) :
MAKE-PQ(y)
MELD(x, y)

DELETE(x) :
CHNG-KEY(x, +∞)

FIND-MIN(x) :
find-min(PQ[x])
```

```
CHNG-KEY(x, k) :
change-key(PQ[x], x, k)
FIND(x)

MELD(x, y) :
if rank[x] > rank[y]
  then
    HANG(y, x)
  else
    HANG(x, y)
    if rank[x] = rank[y]
    then
      rank[y]←rank[y]+1

FIND(x) :
CUT-PATH(x)
COMPRESS-PATH(x)
return p[x]
```

```
CUT-PATH(x) :
if p[x] ≠ x then
  CUT-PATH(p[x])
  UNHANG(x, p[x])

COMPRESS-PATH(x) :
if p[x] ≠ x then
  COMPRESS-PATH(p[x])
  HANG(x, p[p[x]])

HANG(x, y) :
insert(PQ[y], find-min(PQ[x]))
p[x] ← y

UNHANG(x, y) :
delete(PQ[y], find-min(PQ[x]))
```

Figure 2: A meldable priority queue obtained by placing a non-meldable priority queue at each node of the union-find data structure.

We combine this non-meldable priority queue with the union-find data structure to obtain a meldable priority queue that supports the following operations:

MAKE-PQ(x)	– Create a priority queue containing the single element x.
INSERT(x, y)	– Insert element y into the priority queue whose root is x.
DELETE(x)	– Delete element x from the priority queue containing it.
FIND-MIN(x)	– Find element with smallest key in queue with root x.
MELD(x, y)	– Meld the queues whose root elements are x and y.
CHNG-KEY(x, k)	– Change the key associated with element x to k.

As in the union-find data structure, each priority queue will have a representative, or root, element. The operations *INSERT*(x, y) and *FIND-MIN*(x) assume that x is the root element of its priority queue. Similarly, *MELD*(x, y) assumes that x and y are root elements. It is possible to extend the data structure with an additional union-find data structure that supports a *find*(x) operation that returns the root element of the priority queue containing x. (As shown in Kaplan *et al.* [17], a meldable priority queue data structure that supports a *MELD*(x, y) operation that melds the priority queues containing the elements x and y, where x and y are not necessarily representative elements must include, at least implicitly, an implementation of a union-find data structure.)

A collection of meldable priority queues is now maintained as follows. Each priority queue of the collection is maintained as a tree of a union-find data structure. Each element x contained in such a tree thus has a parent pointer $p[x]$ assigned to it by the union-find data structure and a rank $rank[x]$. In addition to that, each element x has a 'local' priority queue $PQ[x]$ associated with it. This priority queue contains the element x itself, and the minimal element of each subtree of x. (Thus if x has d children, $PQ[x]$ contains $d+1$ elements.) If x is at the root of a union-find tree, then to find the minimal element in the priority queue of x, a *FIND-MIN*(x) operation, we simply need to find the minimal element is the priority queue $PQ[x]$, a *find-min*$(PQ[x])$ operation.

When an element x is first inserted into a priority queue, by a *MAKE-PQ*(x) operation, we initialize the priority queue $PQ[x]$ of x to contain x, and no other element. We also set $p[x]$ to x, to signify that x is a root, and set $rank[x]$ to 0.

If x and y are root elements of the union-find trees containing them, then a *MELD*(x, y) operation is performed as follows. As in the union-find data structure, we compare the ranks of x and y and hang the element with the smaller rank on the element with the larger rank. If the ranks are equal we decide, arbitrarily, to hang x on y and we increment $rank[y]$. Finally, if x is hung on y, then to maintain the invariant condition stated above, we insert the minimal element in $PQ[x]$ into $PQ[y]$, an $insert(PQ[y], \mathit{find\text{-}min}(PQ[x]))$ operation. (If y is hung on x we perform an $insert(PQ[x], \mathit{find\text{-}min}(PQ[y]))$ operation.)

A $DELETE(x)$ operation, which deletes x from the priority queue containing it is implemented in the following indirect way. We change the key associated with x to $+\infty$, using a $CHNG\text{-}KEY(x, +\infty)$ operation, to signify that x was deleted, and we make the necessary changes to the data structure, as described below. Each priority queue in our collection keeps track of the total number of elements contained in it, and the number of deleted elements contained in it. When the fraction of deleted elements exceeds a half, we simply rebuild this priority queue. A standard argument shows that this rebuilding affects the amortized cost of all the operations by only a constant factor. (This simple idea is also used by Kaplan *et al.* [18].)

How do we implement a $CHNG\text{-}KEY(x, k)$ operation then? If x is a root element, we simply change the key of x in $PQ[x]$ using a $\mathit{change\text{-}key}(PQ[x], x, k)$ operation. If x is not a root, then before changing the key of x we perform a $FIND(x)$ operation. A $FIND(x)$ operation compresses the path connecting x to the root by cutting all the edges along the path and hanging all the elements encountered directly on the root. Let $x = x_1, x_2, \ldots, x_k$ be the sequence of elements on the path from x to the root of its tree. For $i = k-1, k-2, \ldots, 1$ we *unhang* x_i from x_{i+1}. This is done by removing $\mathit{find\text{-}min}(PQ[x_i])$ from $PQ[x_{i+1}]$. After that, we hang all the elements $x_1, x_2, \ldots, x_{k-1}$ on x_k. This is done by setting $p[x_i]$ to x_k and by adding $\mathit{find\text{-}min}(PQ[x_i])$ to $PQ[x_k]$. (Note that we also unhang x_{k-1} from x_k and then hang it back.)

If x is not a root element then after a $FIND(x)$ operation, x is a child of the root. Changing the key of x is now relatively simple. We again unhang x from $p[x]$, change the key of x and then hang x again on $p[x]$. A moment's reflection shows that it is, in fact, enough just to change the key of x in $PQ[x]$, and then perform a $FIND(x)$ operation. The element x may temporarily be contained in some priority queues with a wrong key, but this will immediately be corrected.

A simple implementation of all these operations is given in Figure 2. The important thing to note is that the operation of a meldable priority queue mimics the operation of a union-find data structure and that changing a pointer $p[x]$ from y to y' is accompanied by calls to $UNHANG(x, y)$ and $HANG(x, y')$.

Since the union-find data structure makes only an amortized number of $O(\alpha(n, n))$ hangings and unhangings per union or find operation, we immediately get that each meldable priority queue operation takes only $O(pq(n)\,\alpha(n, n))$ amortized time. This is precisely the analysis given in [37] (and also in [23]). In the next section, we tighten the analysis so as to get *no* asymptotic overhead with current priority queues.

4 The improved analysis

In this section we present an improved analysis of the data structure presented in the previous section. We assume that the non-meldable priority queue $\mathcal{P}$ supports *insert*, *delete* and *find-min* operations in $O(pq(n))$ (randomized) amortized time. By applying a very simple transformation described in [1], we can actually assume that the amortized cost of *insert* and *find-min* operations is $O(1)$ and that only the amortized cost of *delete* operations is $O(pq(n))$. We now claim:

Theorem 4.1 *If $\mathcal{P}$ is a priority queue data structure that supports insert and find-min operations in $O(1)$ (expected) amortized time and delete operations in $pq(n) = O(\log n)$ (expected) amortized time, then $\mathcal{T}(\mathcal{P})$ is a priority queue data structure that supports insert, find-min and meld operations in $O(1)$ (expected) amortized time and delete operations in $O(pq(n)\,\alpha(n, n/pq(n)))$ (expected) amortized time, where $\alpha(m, n)$ is the inverse Ackermann function appearing in the analysis of the union-find data structure, and n is the total number of operations performed on all the priority queues.*

Proof: Consider a sequence of n operations on the data structure, of which $f \le n$ are $DELETE$ or $CHNG\text{-}KEY$ operations. (Each such operation results in a $FIND$ operation being performed, hence the choice of the letter f.) Our aim is to show that the cost of carrying out all these operations is $O(n + f\,pq(n)\,\alpha(n, n/pq(n)))$. This bounds the amortized cost of each operation in terms of the maximum number of elements contained in all the priority queues. (See a discussion of this point at the end of the section.)

All the operations on the data structure are associated with changes made to the parent pointers $p[x]$ of the elements contained in the priority queues. To change the value of $p[x]$ from y to y', we first call $\mathit{UNHANG}(x, y)$ which performs a delete operation on $PQ[y]$, and then call $\mathit{HANG}(x, y')$ which performs an insert operation on $PQ[y']$ and sets $p[x]$ to y'. As insert operations are assumed to take constant time, we can concentrate our attention on the delete, or *UNHANG*, operations. As the total number of pointer changes made in the union-find data structure is at most $O(n\,\alpha(n, n))$, and as each priority queue acted upon is of size at most n, we get immediately an upper bound of $O(n\,pq(n)\,\alpha(n, n))$ on the total number of operations performed. This is essentially the analysis presented in [37] and [23]. We want to do better than that.

If element x is a root of one of the union-find trees, we let $\mathit{size}(x)$ be the number of elements contained in its tree. If x is no longer a root, we let $\mathit{size}(x)$ be the number of descendants it had just before it was hanged on another element. It is easy to see that we always have $\mathit{size}(x) \geq 2^{\mathit{rank}(x)}$.

Let

$$p = pq(n) + \frac{n}{f} \quad , \quad S = p^2 \quad , \quad L = \log S \; .$$

We say that an element x is *big* if $\mathit{size}(x) \geq S$. Otherwise, it is said to be *small*. We say that an element x is *high* if $\mathit{rank}(x) \geq L$. Otherwise, it is said to be *low*. Note that if an element is big (or high), so are all its ancestors. We also note that all high elements are big, but big elements are not necessarily high. We let *SMALL*, *BIG*, *LOW* and *HIGH* be the sets of small, big, low and high vertices, respectively. As noted above, we have $\mathit{SMALL} \subseteq \mathit{LOW}$ and $\mathit{HIGH} \subseteq \mathit{BIG}$ but $\mathit{LOW} \cap \mathit{BIG}$ may be non-empty.

Below we bound the total cost of all the $\mathit{UNHANG}(x, p[x])$ operations. All other operations take only $O(n)$ time. We separate the analysis into the following five cases:

Case 1: $x, p[x] \in \mathit{SMALL}$

We are doing at most f path compressions. Each path in the union-find forest contains at most L small elements. (This follows from the invariant $\mathit{rank}[p[x]] > \mathit{rank}[x]$ and from the fact that high elements are big.) Thus, each path compression involves at most L unhang operations in which $x, p[x] \in \mathit{SMALL}$. As each priority queue involved is of size at most S, the total cost is

$$O(f \cdot L \cdot pq(S)) \;=\; O(f \cdot p) \;=\; O(n + f \cdot pq(n)) \; .$$

(Note that $L = \log S = O(\log p)$ and that $pq(S) = O(\log S) = O(\log p)$. (We assume that $pq(n) = O(\log n)$.) Hence $L \cdot pq(S) = O(\log^2 p) = O(p)$.)

Case 2: $x \in \mathit{SMALL}$ **and** $p[x] \in \mathit{BIG}$**.**

In each one of the f path compressions performed there is at most one unhang operation of this form. (As ancestors of big elements are also big.) Hence, the total cost here is $O(f\,pq(n))$.

Case 3: $x, p[x] \in \mathit{BIG} \cap \mathit{LOW}$**.**

To bound the total cost of these operations we bound the number of elements that are contained at some stage in $\mathit{BIG} \cap \mathit{LOW}$. An element is said to be a *minimally-big* element if it is big but all its descendants are small. As each element can have at most one minimally-big ancestor, and each minimally-big element has at least S descendants, it follows that there are at most n/S minimally-big elements. As each big element is an ancestor of a minimally-big element, it follows that there are at most Ln/S elements in $\mathit{BIG} \cap \mathit{LOW}$.

An element $x \in \mathit{BIG} \cap \mathit{LOW}$ can be unhanged from at most L other elements of $\mathit{BIG} \cap \mathit{LOW}$. (After each such operation $\mathit{rank}[p[x]]$ increases, so after at most L such operations $p[x]$ must be high.) The total number of operations of this form is at most $L^2 n/S < n/p$. Thus, the total cost of all these operations is $O(n\,pq(n)/p) = O(n)$.

Case 4: $x \in \mathit{BIG} \cap \mathit{LOW}$ **and** $p[x] \in \mathit{HIGH}$**.**

As in Case 2, each one of the f path compressions performed causes at most one unhang operation of this form. (As ancestors of high elements are also high.) Hence, the total cost here is $O(f\,pq(n))$.

Case 5: $x, p[x] \in \mathit{HIGH}$**.**

To bound the number of $UNHANG(x, p[x])$ operations in which $x, p[x] \in HIGH$, we rely on Lemma 2.2. As each $UNHANG(x, p[x])$ operation, where $x \in HIGH$ is associated with a parent pointer change of a high vertex, it follows that the total number of such operations is at most $O(\frac{n}{S} + f\,\alpha(f + \frac{n}{S}, \frac{n}{S})) = O(f\,\alpha(f, \frac{n}{S}))$. (Note that $S = 2^L$ and $f \geq n/S$.) Now

$$\alpha(f, \frac{n}{S}) \;\leq\; \alpha(\frac{n}{p}, \frac{n}{p^2}) \;\leq\; \alpha(n, \frac{n}{p}) \;\leq\; \alpha(n, \frac{n}{pq(n)}) \;.$$

This chain of inequalities follows from the fact that $f \geq n/p$ and from simple properties of the $\alpha(m, n)$ function. (The $\alpha(m, n)$ function is decreasing in its first argument, increasing in the second, and $\alpha(m, n) \leq \alpha(cm, cn)$, for $c \geq 1$.) As the cost of each *delete* operation is $O(pq(n))$, the cost of all unhang operations with $x, p[x] \in HIGH$ is at most $O(f \cdot pq(n) \cdot \alpha(n, n/pq(n)))$.

The total cost of all unhang operations is therefore $O(n + f\,pq(n)\,\alpha(n, n/pq(n)))$, as required. □

The analysis just given shows that the amortized cost of a delete operation is $O(pq(n)\,\alpha(n, n/pq(n)))$, where n is not the number of elements in the priority queue from which the element is deleted, but rather the total number of elements, in all the priority queues, at the end of the sequence of operations on which we amortize. This is sufficient, however, for all applications that we are aware of.

The main reason for getting an amortized bound in term of the total number of elements, and not the current number of elements, is our reliance on the standard analysis of the union-find data structure. This analysis bounds the amortized cost of a *find* operation by $O(\alpha(m, n))$, where m is the total number of operations performed, and n is the total number of elements acted upon.

A slightly more refined analysis of the union-find data structure appears in Kaplan *et al.* [18] where it is shown that the amortized cost of a *find* operation can actually be bounded by $O(\bar{\alpha}(\lfloor m/n \rfloor, \ell))$, where ℓ is the size of the set containing the element and $\bar{\alpha}(k, \ell) = \min\{\, i \geq 1 \mid A(i, k) > \log_2 \ell \,\}$. (Note that $\alpha(m, n) = \bar{\alpha}(\lfloor m/n \rfloor, n)$.) Using this refined analysis it seems possible to obtain bounds on the amortized time of the delete operations that depend on the current number of elements in the priority queues. We do not pursue this matter here.

5 Using atomic heaps

Atomic heaps, invented by Fredman and Willard [13], are non-meldable priority queues that support *insert*, *find-min* and *delete* operations in *constant* amortized time, when the number of elements in the priority queue is $O(\log^2 n)$. The constant amortized operation time is obtained by using a carefully prepared look-up table of size $O(n)$ that can be constructed in $O(n)$ time. The same table can be used to maintain many small priority queues. Atomic heaps support some additional operations, like predecessor search, but these are not needed here.

We can use atomic heaps to obtain a slightly improved version of the transformation from non-meldable priority queues to meldable ones. The modified transformation, which we denote by $\mathcal{T}^A$, is almost identical to the transformation of Section 3. The only difference is that atomic heaps are used in nodes of the union-find data structure that are of small size. As in the previous section, we define *size*(x) as follows: if x is a root then *size*(x) is the number of descendants of x, and if x is not a root, then *size*(x) is the number of descendants that x had just before it was hung on another node. Given a threshold S, we say that a node x is *small*, is *size*$(x) \leq S$, and *big*, otherwise. As deleted elements are only marked as deleted, until the whole priority queue is rebuilt, *size*(x) never decreases.

Let n be a bound on the total number of operations that will be performed on the collection of priority queues that we are supposed to maintain. Suppose, at first, that we know n in advance. (We will remove this assumption shortly.) Let $\mathcal{P}$ be the non-meldable priority queue data structure supplied to the transformation. We refer to priority queues maintained using this data structure as $\mathcal{P}$-heaps. Let $S = \log^2 n$. If x is a small node in the union-find forest, i.e., if *size*$(x) \leq S$, then instead of placing in x a $\mathcal{P}$-heap, we place in x an atomic heap. When a small node x becomes big, the elements in the atomic heap of x are moved into a newly created $\mathcal{P}$-heap. This transition is paid for by the original insertion of the elements into the atomic heap. As *size*(x) never decreases, a big node never becomes small, so a $\mathcal{P}$-heap is never converted back into an atomic heap.

If n, the total number of operations to be performed, is not known in advance, we can use a simple doubling technique. We let n be the number of operations performed so far. When the number of operations doubles, we rebuild all the priority queues. It is easy to see that this changes the amortized cost of each operation by only a constant factor. We now claim:

Theorem 5.1 *If $\mathcal{P}$ is a priority queue data structure that supports insert and find-min operations in $O(1)$ (expected) amortized time and delete operations in $pq(n) = O(\log n)$ (expected) amortized time, then $\mathcal{T}^A(\mathcal{P})$ is a priority queue data structure that supports insert, find-min and meld operations in $O(1)$ (expected) amortized time and delete operations in $O(pq(n) + \alpha(n,n))$ (expected) amortized time, where $\alpha(m,n)$ is the inverse Ackermann function appearing in the analysis of the union-find data structure, and n here is the total number of operations performed on all the priority queues.*

Proof: The proof is very similar to the proof of Theorem 4.1. Our goal is to show that the total cost of all the unhang operations is bounded by $O(n + f(pq(n) + \alpha(n,n)))$, where f is the number of *delete* operations performed. We use the same definitions of the sets *SMALL*, *BIG*, *LOW* and *HIGH* used in the proof of Theorem 4.1, but with the following modified parameters:

$$p = \log n \quad , \quad S = p^2 = \log^2 n \quad , \quad L = \log S = 2\log\log n \; .$$

We break the analysis into the same five cases:

Case 1: $x, p[x] \in \mathit{SMALL}$

As the cost of each operation on an atomic heap requires only $O(1)$ time, the total cost of all the operations here is $O(n + f\,\alpha(n,n))$.

Case 2: $x \in \mathit{SMALL}$ **and** $p[x] \in \mathit{BIG}$**.**

In each one of the f path compressions performed there is at most one unhang operation of this form. (As ancestors of big elements are also big.) Hence, the total cost here is $O(f\,pq(n))$.

Case 3: $x, p[x] \in \mathit{BIG} \cap \mathit{LOW}$**.**

The total number of unhang operations of this form is at most $L^2 n/S < n/p = n/\log n$. (See the proof of Theorem 4.1.) Thus, the total cost of all these operations is $O(n\,pq(n)/\log n) = O(n)$.

Case 4: $x \in \mathit{BIG} \cap \mathit{LOW}$ **and** $p[x] \in \mathit{HIGH}$**.**

As in Case 2, each one of the f path compressions performed causes at most one unhang operation of this form. (As ancestors of high elements are also high.) Hence, the total cost here is $O(f\,pq(n))$.

Case 5: $x, p[x] \in \mathit{HIGH}$**.**

By Lemma 2.2, the number of unhang operations of this form is at most $O(\frac{n}{S} + f\,\alpha(f + \frac{n}{S}, \frac{n}{S}))$. The cost of each such operation is at most $O(pq(n))$. If $f > n/(\log n)^{3/2}$, then $\alpha(f + \frac{n}{S}, \frac{n}{S}) = O(1)$, and the total cost is $O((\frac{n}{S} + f)pq(n)) = O((\frac{n}{\log^2 n} + f)pq(n)) = O(n + f\,pq(n))$, as required. If $f \le n/(\log n)^{3/2}$ then the total cost is $O(\frac{n}{(\log n)^{3/2}}\,pq(n)\,\alpha(n,n)) = O(n)$.

The total cost of all unhang operations is therefore $O(n + f(pq(n) + \alpha(n,n)))$, as required. □

6 Bounds in terms of the maximal key value

In this section we describe a simple transformation, independent of the transformation of Section 3, that speeds up the operation of a meldable priority queue data structure when the keys of the elements are integers taken from the range $[1, N]$, where N is small relative to n, the number of elements. More specifically, we show that if $\mathcal{P}$ is a meldable priority queue data structure that supports *delete* operations in $O(pq(n))$ amortized time, and all other operations in $O(1)$ amortized time, where n is the number of elements in the priority queue, then it is possible to transform it into a meldable priority queue data structure $\mathcal{T}'(\mathcal{P})$ that supports *delete* operations in $O(pq(\min\{n, N\}))$ amortized time, and all other operations in $O(1)$ time. To implement this transformation we need random access capabilities, so it cannot be implemented on a pointer machine.

To simplify the presentation of the transformation, we assume, at first, that a *delete* operation receives references to the element x to be deleted *and* to the priority queue containing it. This is a fairly standard assumption.[1] Note, however, that the *delete* operation obtained by our first transformation is stronger as it only requires a reference to the element, and not to the priority queue. We later show how to dispense with this assumption.

The new data structure $\mathcal{T}'(\mathcal{P})$ uses two different representations of priority queues. The first representation, called the *original*, or *non-compressed* representation is simply the representation used by $\mathcal{P}$. The second representation, called the *compressed* representation, is composed of an array of size N containing for each integer $k \in [1, N]$ a pointer to a doubly linked list of the elements with key k contained in the priority queue. (Some of the lists may, of course, be empty.) In addition to that, the compressed representation uses an original representation of a priority queue that holds the up to N distinct keys belonging to the elements of the priority queue.

Initially, all priority queues are held using the original representation. When, as a result of an *insert* or a *meld* operation, a priority queue contains more than N elements, we convert it to compressed representation. This can be easily carried out in $O(N)$ time. When, as a result of a *delete* operation, the size of a priority queue drops below $N/2$, we revert back to the original representation. This again takes $O(N)$ time. The original representation is therefore used to maintain *small* priority queues, i.e., priority queues containing up to N elements. The compressed representation is used to represent *large* priority queues, i.e., priority queues containing at least $N/2$ elements. (Priority queues containing between $N/2$ and N elements are both small and large.)

By assumption, we can insert elements to non-compressed priority queues in $O(1)$ amortized time, and delete elements from then in $O(pq(n)) = O(pq(N))$ amortized time. We can also insert an element into a compressed priority queue in $O(1)$ amortized time. We simply add the element into the appropriate linked list, and if the added element is the first element of the list, we also add the key of the element to the priority queue. Similarly, we can delete an element from a compressed priority queue in $O(pq(N))$ amortized time. We delete the element from the corresponding linked list. If that list is now empty, we delete the key from the non-compressed priority queue. As the compressed priority queue contains at most N keys, that can be done in $O(pq(N))$ amortized time. Since *insert* and *delete* operations are supplied with a reference to the priority queue to which an element should be inserted, or from which it should be deleted, we can keep a count of the number of elements contained in the priority queue. This can be done for both representations. (Here is where we use the assumption made earlier. As mentioned, we will explain later why this assumption is not really necessary.) These counts tell us when the representation of a priority queue should be changed.

A small priority queue and a large priority queue can be melded simply be inserting each element of the small priority queue into the large one. Even though this takes $O(n)$ time, where n is the number of elements in the small priority queue, we show below that the amortized cost of this operation is only $O(1)$.

Two large priority queues can be easily melded in $O(N)$ time. We simply concatenate the corresponding linked lists and add the keys that are found, say, in the second priority queue, but not in the first, into the priority queue that holds the keys of the first priority queue. The second priority queue is then destroyed. We also update the size of the obtained queue. Again, we show below that the amortized cost of this is only $O(1)$.

Theorem 6.1 *If $\mathcal{P}$ is a priority queue data structure that supports insert, find-min and meld operations in $O(1)$ (expected) amortized time and delete operations in $O(pq(n))$ (expected) amortized time, then $\mathcal{T}'(\mathcal{P})$ is a priority queue data structure that supports insert, find-min and meld operations in $O(1)$ (expected) amortized time and delete operations in $O(pq(\min\{n, N\}))$ (expected) amortized time.*

Proof: We use a simple potential based argument. The potential of a priority queue held in original, non-compressed, representation is defined to be $1.5n$, where n the number of elements contained in it. The potential of a compressed priority queue is N, no matter how many elements it contain. The potential of the whole data structure is the sum of the potentials of all the priority queues.

The operations *insert*, *delete* and *find-min* have a constant actual cost and they change the potential of the data structure by at most an additive constant. Thus, their amortized cost is constant.

[1]A reference to the appropriate priority queue can be obtained using a separate union-find data structure. The amortized cost of finding a reference is then $O(\alpha(n, n))$. This is *not* good enough for us here as we are after bounds that are independent of n.

Compressing a priority queue containing $N \le n \le 2N$ elements requires $O(N)$ operations but it reduces the potential of the priority queue from $1.5n$ to N, a drop of at least $N/2$, so with proper scaling the amortized cost of this operation may be taken to be 0. Similarly, when a compressed priority queue containing $n \le N/2$ elements is converted to original representation, the potential of the priority queue drops from N to $1.5n$, a drop of at least $N/4$, so the amortized cost of this operation is again 0.

Melding two original priority queues has a constant actual cost. As the potential of the data structure does not change, the amortized cost is also constant. Melding two compressed priority queues has an actual cost of $O(N)$, but the potential of the data structure is decreased by N, so the amortized cost of such meld operations is again 0. Finally, merging a small priority queue of size $n \le N$, in original representation, and a compressed priority queue has an actual cost of $O(n)$ but the potential decreases by $1.5n$, giving again an amortized cost of 0. This completes the proof. □

We next explain the small modification needed in the transformation described above to accommodate *delete* operations that only get a reference to the element to be deleted. The problem is that without a reference to the priority queue acted upon, *delete* operations cannot decrement, in constant time, the counter holding the number of elements contained in the priority queue. Still, *insert* operations can increment such a counter as they do get a reference to the priority queue. We can thus maintain, for each non-compressed priority queue, a counter *ins* that counts the number of insertions made into it. When two priority queues are melded, we add these counters. We convert a priority queue into compressed representation when its counter *ins* exceeds N. The potential of a non-compressed priority queue is now defined to be $1.5\,ins$.

We have no way of knowing when the actual size of priority queue drops below $N/2$. The only reason, however, for converting compressed priority queues back into non-compressed representation is to save space. We can, however, reclaim space on a global rather than local basis. Instead of maintaining the individual size of each compressed priority queue, we maintain the number b of compressed priority queues, and their total size s. When the total size s drops below $bN/4$, i.e., when the space utilization drops below 1/4, we naively compute the size of each compressed priority queue, and convert it to a non-compressed representation if its size is below $N/2$. (The *ins* counter of such a priority queue is then set to its actual size.) This can be easily done in $O(bN)$ time. As at most $b/2$ of the priority queues can have more than $N/2$ elements, this reduces the potential of the whole data structure by at least $b/2 \cdot (N/4)$, so the amortized cost of this 'garbage collection' is 0. This completes the description of the required modification.

7 Concluding remarks

We presented an improved analysis of a general transformation, first presented in van Emde Boas [37], that adds a meld operation to priority queue data structures that do not support it, with essentially *no extra cost.* (The analysis of [37] only showed that the extra cost is tiny.) We also presented a second transformation that speeds up the operations of meldable priority queues when the range of the possible keys is small.

Combined with Thorup's [32] technique of transforming sorting algorithms into priority queue data structures, our result can be stated as follows: A sorting algorithm that sorts up to n elements in $O(n\,s(n))$ time, where $s(n) = \Omega(\alpha(n,n))$, can be converted into a *meldable* priority queue data structure that supports delete operations in $O(s(n))$ amortized time, and all other operations in $O(1)$ amortized time.

References

[1] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. Technical Report DIKU 98-8, Dept. Comput. Sc., Univ. Copenhagen, 1998. (The result needed from here is *not* included in the FOCS'98 extended abstract.)

[2] F. Bock. An algorithm to construct a minimum directed spanning tree in a directed network. *in: Developments in Operations Research, Gordon and Breach, New York*, pages 29–44, 1971.

[3] J.D. Bright. Range-restricted mergeable priority queues. *Information Processing Letters*, 47(3):159–164, 1993.

[4] G.S. Brodal. Worst-case efficient priority queues. In *Proc. of 7th SODA*, pages 52–58, 1996.

[5] P.M. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9:309–312, 1979.

[6] B. Chazelle. A minimum spanning tree algorithm with inverse ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[7] Y.J. Chu and T.H. Liu. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14:1396–1400, 1965.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms.* The MIT Press, second edition, 2001.

[9] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] J. Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards*, 71B:233–240, 1967.

[11] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[12] M.L. Fredman and D.E. Willard. Surpassing the information-theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[13] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[14] H.N. Gabow, Z. Galil, T.H. Spencer, and R.E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6:109–122, 1986.

[15] Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. In *Proc. of 34th STOC*, pages 602–608, 2002.

[16] Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *Proc. of 43rd FOCS*, pages 135–144, 2002.

[17] H. Kaplan, N. Shafrir, and R.E. Tarjan. Meldable heaps and boolean union-find. *Proc. of 34th STOC*, pages 573–582, 2002.

[18] H. Kaplan, N. Shafrir, and R.E. Tarjan. Union-find with deletions. In *Proc. of 13th SODA*, pages 19–28, 2002.

[19] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[20] R.M. Karp. A simple derivation of Edmonds' algorithm for optimum branchings. *Networks*, 1:265–272, 1971.

[21] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.

[22] R. Mendelson, R.E. Tarjan, M. Thorup, and U. Zwick. Melding priority queues. In *Proc. of 9th SWAT*, 2004. To appear.

[23] R. Mendelson, M. Thorup, and U. Zwick. Meldable RAM priority queues and minimum directed spanning trees. In *Proc. of 15th SODA*, pages 40–48, 2004.

[24] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[25] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[26] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[27] R.E. Tarjan. Finding optimum branchings. *Networks*, 7:25–35, 1977.

[28] R.E. Tarjan. *Data structures and network algorithms.* SIAM, 1983.

[29] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[30] R.E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31:245–281, 1984.

[31] M. Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[32] M. Thorup. Equivalence between priority queues and sorting. In *Proc. of 43rd FOCS*, pages 125–134, 2002.

[33] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *Journal of Algorithms*, 42(2):205–230, 2002.

[34] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proc. of 35th STOC*, pages 149–158, 2003.

[35] M. Thorup. On AC^0 implementations of fusion trees and atomic heaps. In *Proc. of 14th SODA*, pages 699–707, 2003.

[36] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[37] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[38] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.