Melding priority queues

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

Abstract

We show that any priority queue data structure that supports *insert*, *delete*, and *find-min* operations in $pq(n)$ time, when there are up to n elements in the priority queue, can be converted into a priority queue data structure that also supports *meld* operations at essentially no extra cost, at least in the amortized sense. More specifically, the new data structure supports *insert*, *meld* and *find-min* operations in $O(1)$ amortized time, and *delete* operations in $O(pq(n)\,\alpha(n, n/pq(n)))$ amortized time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function appearing in the analysis of the union-find data structure. For all conceivable values of $pq(n)$, the multiplicative term $\alpha(n, n/pq(n))$ is constant. This holds, for example, if $pq(n) = \Omega(\log^* n)$. In such cases, adding the meld operation does not increase the amortized asymptotic cost of the priority queue operations. The result is obtained by an improved analysis of a construction suggested recently by three of the authors (see [MTZ04]). This construction places 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 all the bounds stated above by 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.

1 Introduction

Priority queues are very basic data structures that are 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 contained 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 [Dij59], and Prim's algorithm [Pri57] for finding a minimum spanning tree, many priority queues support this operation directly, sometimes in constant time.

[*]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 08544, USA, and InterTrust Technologies Corporation, 4750 Patrick Henry Drive, Santa Clara 95054-1851, USA. E-mail: `ret@cs.princeton.edu`.

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

The efficient implementation of several algorithms, such as the algorithm of Edmonds [Edm67] for computing optimum branching and minimum directed spanning trees, require the maintenance of a collection of priority queues. In addition to the standard operations performed on individual priority queues in this collection, we 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 [FT87], 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 or amortized time bounds, see [Tar85].) Brodal [Bro96] 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 optimal.

While $O(\log n)$ is the best delete time possible if only comparisons between keys are allowed, much better time bounds can be obtained in the word RAM model of computation, as was first demonstrated by Fredman and Willard [FW93, FW94]). 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 (see, e.g., [AHU74]). 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 [Tho00, Tho02] obtained a general equivalence between priority queues and sorting. More specifically, he showed that if n elements can be sorted in $O(nf(n))$ time, where $f(n)$ is a non-decreasing, then the basic priority queue operations can be implemented in $O(f(n))$ time. Using a recent $O(n \log\log n)$ sorting algorithm of Han [Han02], this gives priority queues that support all operations in $O(\log\log n)$ time. Thorup [Tho03] 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 but the equivalence to sorting.) Han and Thorup [HT02] obtained recently a randomized $O(n\sqrt{\log\log n})$ time sorting algorithm. This translates into priority queues with $O(\sqrt{\log\log n})$ expected time per operation.

Adding a meld operation

The priority queues mentioned in the previous paragraph do *not* support meld operations. The main result of this paper is a general transformation that takes these priority queues, and in fact any priority queue data structure, and produces new priority queue data structures that do support the *meld* operation with essentially no increase in the amortized cost of the operations! 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 also supports *meld* operations at essentially no extra cost, at least in the amortized sense. More specifically, the new data structure supports *insert*, *meld* and *find-min* operations in $O(1)$ amortized time, and *delete* operations in $O(pq(n)\,\alpha(n, n/pq(n)))$ amortized time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function appearing in the analysis of the union-find data structure. For all conceivable values of $pq(n)$, the multiplicative term $\alpha(n, n/pq(n))$ is constant. This holds, for example, if $pq(n) = \Omega(\log^* n)$. In such cases, adding the meld operation does not increase the amortized asymptotic cost of the priority queue operations. If the original priority queue is deterministic, then so is the new one.

The result is obtained by an improved analysis of a construction suggested recently by three of the authors (see [MTZ04]). This construction places a non-meldable priority queue at each node of a union-find data structure. The simple analysis given in [MTZ04] gave an upper bound of $O(pq(n)\,\alpha(n, n)))$ on the cost of all priority queue operations. Here we reduce the amortized cost of *insert*, *meld* and *find-min* operations to $O(1)$, and more importantly, reduce the amortized cost of *delete* operations to $O(pq(n)\,\alpha(n, n/pq(n)))$. In other words, we replace the factor $\alpha(n,n)$ by $\alpha(n, n/pq(n))$. This is significant as $\alpha(n, n/pq(n))$ is *constant* for all conceivable values of $pq(n)$, e.g., if $pq(n) = \Omega(\log^* n)$. In such cases we get the meld operation for *free*!

Applying this result to non-meldable priority queue data structures obtained recently by Thorup [Tho02], and by Han and Thorup [HT02], 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. Furthermore, Thorup's equivalence between priority queues and sorting and the transformation presented here imply that any sorting algorithm that can sort n elements in $O(nf(n))$ time, where $f(n)$ is a non-decreasing function, can be used to construct *meldable* priority queues with $O(1)$ amortized cost for *insert*, *find-min* and *meld* operations, and $O(f(n)\,\alpha(n, n/f(n)))$ amortized cost for *delete* operations.

As a by-product of the improved meldable priority queues mentioned above, 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 [GGST86] that works for arbitrary edge weights. More details (and references) on algorithms for computing minimum directed spanning trees can be found in [MTZ04].

Although the most interesting results are obtained by applying our transformation to RAM priority queues, the transformation itself only uses the capabilities of a pointer machine. Furthermore, no operations are performed directly on the keys of the elements. In particular, by applying our transformation to the classical binary heap data structure (Williams [Wil64]) we obtain, without too much effort, comparison-based meldable priority queues that support all operations in $O(\log n)$ amortized time. (Binomial heaps (Vuillemin [Vui78]) are, of course, better as they support these operations in $O(\log n)$ worst case time, and the heaps of Brodal [Bro96] as even better as they support *decrease-key* and all other operations, except *delete*, in constant time.) This simple example is brought here to show that our transformation yields interesting and non-trivial results even when applied to relatively simple data structures.

Improvement for smaller integer keys

We also show, using an independent transformation, that when all keys are integers in the range $[1, N]$, all occurrences of n in the bounds stated above can be replaced by N, or more generally, by $\min\{n, N\}$. Such a result did not appear in [MTZ04]. This, in conjunction with the previous transformation, allows us, for example, to add a meld operation, with constant amortized cost, to the priority queue data structure of van Emde Boas [van77, vKZ77] which has $pq(n) = O(\log\log N)$. The amortized cost of a delete operation is then:

$$O(\log\log N \cdot \alpha(\min\{n,N\}, \min\{n,N\}/\log\log N) \;=\; O(\log\log N \cdot \alpha(N, N/\log\log N)) \;=\; O(\log\log N)\ .$$

(The original data structure of van Emde Boas requires randomized hashing to run in linear space [MN90]. A deterministic version is presented in [Tho02].)

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 our transformation. In Section 4 we present our improved analysis. In Section 5 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 6 with some concluding remarks and open problems.

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

2 The Union-find data structure

A union-find data structure is a data structure that 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 element 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. These parent pointers define trees that correspond to the sets of elements 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 points 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 [Tar75] 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.

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

Lemma 2.1 *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 $O((f + \frac{n}{2^k}) \cdot \alpha(f + \frac{n}{2^k}, \frac{n}{2^k}))$.*

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

We claim that the operations performed on the top parts of the trees correspond to a sequence of at most $f + n/2^{k-1}$ operations performed on the high elements. By the standard analysis of the union-find data structure, the total cost of at most $f + n/2^{k-1}$ union-find operations on $n/2^k$ elements is at most $O((f + \frac{n}{2^{k-1}}) \cdot \alpha(f + \frac{n}{2^{k-1}}, \frac{n}{2^k})) = O((f + \frac{n}{2^k}) \cdot \alpha(f + \frac{n}{2^k}, \frac{n}{2^k}))$, 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. □

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 [Tar75], [Tar83], or [CLRS01]) to produce a *meldable* priority queue data structure with essentially no increase in the amortized operation cost. This transformation is essentially the transformation described in [MTZ04] with some minor modifications. An improved analysis of this transformation appears in the next section.

This, transformation, which we denote by $\mathcal{T}$, 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 the priority queue 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 associated with the element x in the priority queue 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 in this section.)

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 that contains the single element x.
INSERT(x, y)	Insert the element y into the priority queue whose root element is x.
DELETE(x)	Delete the element x from the priority queue containing it.
FIND-MIN(x)	Find an element with the smallest key in the priority queue with root x.
MELD(x, y)	Meld the priority queues whose root elements are x and y.
CHANGE-KEY(x, k)	Change the key associated with the 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 explained in Kaplan *et al.* [KST02a], 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

MAKE-PQ(x) :

$p[x] \leftarrow x$
$rank[x] \leftarrow 0$
$PQ[x] \leftarrow$ *make-pq*(x)

INSERT(x, y) :

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

DELETE(x) :

CHANGE-KEY$(x, +\infty)$

FIND-MIN(x) :

find-min$(PQ[x])$

CHANGE-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 $rank[y] \leftarrow rank[y] + 1$

FIND(x) :

CUT-PATH(x)
COMPRESS-PATH(x)
return $p[x]$

CUT-PATH(x) :

if $p[x] \neq x$ then
  CUT-PATH$(p[x])$
  UNHANG$(x, p[x])$

COMPRESS-PATH(x) :

if $p[x] \neq x$ then
  COMPRESS-PATH$(p[x])$
  HANG$(x, p[p[x]])$

HANG(x, y) :

insert$(PQ[y],$ *find-min*$(PQ[x]))$
$p[x] \leftarrow y$

UNHANG(x, y) :

delete$(PQ[y],$ *find-min*$(PQ[x]))$

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

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 get 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, which employs union by rank, 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 hang on y, then to maintain the invariant condition stated above, we insert the minimal element in $PQ[x]$ into $PQ[y]$, an *insert*$(PQ[y],$ *find-min*$(PQ[x]))$ operation. (If y is hang on x we perform a *insert*$(PQ[x],$ *find-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 *CHANGE-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 on 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. This affects the amortized cost of all the operations by only a constant factor. (For more details see Kaplan *et al.* [KST02b].)

How do we implement a *CHANGE-KEY*(x, k) operation then? If x is a root element, we simply change the key of x in $PQ[x]$ using a *change-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 to x_{i+1}. This is done by removing *find-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 *find-min*$(PQ[x_i])$ from $PQ[x_k]$. (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 does not cause any problems as it would immediately be deleted.

A simple implementation of all these operations is given in Figure 2. The important thing to note 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 structures makes only an amortized number of $O(\alpha(n, n))$ hanging 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 was the result obtained in [MTZ04]. Here, 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 simple transformation described in Alstrup *et al.* [AHR98] 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 $O(pq(n))$ (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 $O(pq(n)\,\alpha(n, n/pq(n)))$ (expected) amortized time, where $\alpha(m, n)$ is the inverse Ackermann function appearing in the analysis of the union-find data structure, and n here is the maximum number of elements contained in the priority queue.*

Proof: Consider a sequence of n operations on the data structure, of which $f \le n$ are *DELETE* or *CHANGE-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 the priority queues. In the full version of the paper we will give a slightly more complicated analysis that bounds the amortized complexity of each operation in terms of the actual number of elements contained in the priority queue at the time of the operation.

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 $UNHANG(x, y)$ which performs a delete operation on $PQ[y]$, a priority queue of size at most $size(y)$, and then call $HANG(x, y')$ 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 [MTZ04]. We want, however, to do better than that.

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

Let $p = pq(n) + n/f$, $S = p^2$ and $L = \log S$. We say that an element x is *big* if $size(x) \geq S$. Otherwise, it is said to be *small*. We say that an element x is *high* if $rank(x) \geq L$. 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 $SMALL \subseteq LOW$ and $HIGH \subseteq BIG$ but $LOW \cap BIG$ may be non-empty.

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

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

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 $rank[p[x]] > rank[x]$ and from the fact that high elements are big.) Thus, each path compression involves at most L unhang operations in which $x, p[x] \in SMALL$. 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: $x \in SMALL$ **and** $p[x] \in BIG$**.**

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: $x, p[x] \in BIG \cap LOW$**.**

To bound the total cost of these operations we bound the number of elements that are contained at some stage in $BIG \cap LOW$. 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 $BIG \cap LOW$.

An element $x \in BIG \cap LOW$ can be unhanged from at most L other elements of $BIG \cap LOW$. (After each such operation $rank[p[x]]$ 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: $x, p[x] \in HIGH$**.**

To bound the number of $UNHANG(x, p[x])$ operations in which $x, p[x] \in HIGH$, we rely on Lemma 2.1. 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 $O((f + \frac{n}{S}) \cdot \alpha(f + \frac{n}{S}, \frac{n}{S})) = O(f \cdot \alpha(f, \frac{n}{S}))$.

(This follows as $n/S \le f$.) Now

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

This chain of inequalities follows from the fact that $f \ge n/p$ 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 $\alpha(m,n) \le \alpha(cm,cn)$, for $c \ge 1$.)

As the cost of each *delete* operation is $O(pq(n))$, the cost of all $UNHANG(x,p[x])$ operations with $x, p[x] \in HIGH$ is at most $O(f \cdot pq(n) \cdot \alpha(n, n/pq(n)))$, as required. □

5 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 slightly simplify the presentation of the transformation, we assume at first that a *delete* operation receives a reference 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 that this assumption is not really necessary, so the *delete* operations obtained using the transformation $\mathcal{T}'$ again require only a reference to the element to be deleted.

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 the compressed representation. It is easy to see that the transformation can be 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. (Note that priority queues containing between $N/2$ and N elements are both small and large.)

By definition, 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

[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.

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 contained 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 5.1 *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 defined to be 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.

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 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. $\square$

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, it is not clear how *delete* operations will 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. That's it!

By applying our two transformation in sequence on the van Emde Boas data structure [van77, vKZ77], we obtained *meldable* priority queues that support *delete* operations in $O(\log\log N)$ amortized time, and *insert*, *find-min* and *meld* operations $O(1)$ amortized time. The first transformation yields meldable priority queues with an amortized delete time of $O(\log\log N \cdot \alpha(n, n/\log\log N))$. For $n \leq N$, this is $O(\log\log N)$. The second transformation then yields meldable priority queues with an amortized delete time of $O(\log\log N)$ for any n.

6 Concluding remarks

We presented an improved analysis of a general transformation, first presented in [MTZ04], that adds a meld operation to priority queue data structures that do not support it, at essentially *no extra cost*. (The analysis of [MTZ04] 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. Any improved non-meldable priority queue data structure, and in fact any improved sorting algorithm, will immediately yield improved meldable priority queue data structures.

References

[AHR98] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. Technical Report DIKU 98-8, 1998.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[Bro96] G.S. Brodal. Worst-case efficient priority queues. In *Proc. of 7th SODA*, pages 52–58, 1996.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Edm67] J. Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards*, 71B:233–240, 1967.

[FT87] 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.

[FW93] 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.

[FW94] 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.

[GGST86] 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.

[Han02] Y. Han. Deterministic sorting in $o(n \log\log n)$ time and linear space. In *Proc. of 34th STOC*, pages 602–608, 2002.

[HT02] 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.

[KST02a] H. Kaplan, N. Shafrir, and R.E. Tarjan. Meldable heaps and boolean union-find. *Proc. of 34th STOC*, pages 573–582, 2002.

[KST02b] H. Kaplan, N. Shafrir, and R.E. Tarjan. Union-find with deletions. In *Proc. of 13th SODA*, pages 19–28, 2002.

[MN90] 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.

[MTZ04] 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.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[Tar83] R.E. Tarjan. Data structures and network algorithms. *CBMS 44, Society for Industrial and Applied Mathematics, Philadelphia, PA*, 1983.

[Tar85] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algenraic and Discrete Methods*, 6:306–318, 1985.

[Tho00] M. Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[Tho02] M. Thorup. Equivalence between priority queues and sorting. In *Proc. of 43rd FOCS*, pages 125–134, 2002.

[Tho03] 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.

[van77] 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.

[vKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.

[Wil64] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.