

Two new methods for transforming priority queues into double-ended priority queues*

Amr Elmasry¹ Claus Jensen² Jyrki Katajainen²

¹ *Department of Computer Engineering and Systems, Alexandria University
Alexandria, Egypt*

² *Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark*

Abstract. Two new ways of transforming a priority queue into a double-ended priority queue are introduced. These methods can be used to improve all known bounds for the comparison complexity of double-ended priority-queue operations. Using an efficient priority queue, the first transformation can produce a double-ended priority queue which guarantees the worst-case cost of $O(1)$ for *find-min*, *find-max*, and *insert*; and the worst-case cost of $O(\lg n)$ including at most $\lg n + O(1)$ element comparisons for *delete*, but the data structure cannot support *meld* efficiently. Using a meldable priority queue that supports *decrease* efficiently, the second transformation can produce a meldable double-ended priority queue which guarantees the worst-case cost of $O(1)$ for *find-min*, *find-max*, and *insert*; the worst-case cost of $O(\lg n)$ including at most $\lg n + O(\lg \lg n)$ element comparisons for *delete*; and the worst-case cost of $O(\min\{\lg m, \lg n\})$ for *meld*. Here, m and n denote the number of elements stored in the data structures prior to the operation in question, and $\lg n$ is a shorthand for $\log_2(\max\{2, n\})$.

1. Introduction

In this paper we study efficient realizations of data structures that can be used to maintain a collection of double-ended priority queues. The fundamental operations to be supported include *find-min*, *find-max*, *insert*, *delete*, and *meld*. For single-ended priority queues only *find-min* or *find-max*, but not both, needs to be supported. Our main focus is on the comparison complexity of double-ended priority-queue operations. To our surprise, even though many data structures [2, 3, 6, 8, 10, 11, 12, 13, 15, 24, 26, 27, 29, 30, 32, 33, 35] have been proposed for the realization of a (meldable) double-ended priority queue, none of them achieve $\lg n + o(\lg n)$ element comparisons per *delete*, if *find-min*, *find-max*, and *insert* are required to have the worst-case cost of $O(1)$.

We use the *word RAM* as our model of computation as defined in [22]. We

*Partially supported by the Danish Natural Science Research Council under contracts 21-02-0501 (project Practical data structures and algorithms) and 272-05-0272 (project Generic programming—algorithms and tools).

assume the availability of standard instructions plus memory allocation and deallocation functions as defined in C++ [5]. Observe that the instructions executed inside the element constructor, the element destructor, and the function used in element comparisons are not included in our instruction counts. We use the term *cost* to denote the sum of instructions, element constructions, element destructions, and element comparisons performed.

When defining a (double-ended) priority queue, we use the locator abstraction discussed in [21]. A *locator* is a mechanism for maintaining the association between an element and its current position in a data structure. A locator follows its element even if the element changes its position inside the data structure. That is, the locator remains valid at all times and refers to the element for which it was created. One way of implementing the locator abstraction is to use handles as proposed in [14, Section 6.5]. The nodes of a data structure only store pointers to elements which again have pointers back to the nodes. In this case, a locator would simply be a pointer to an element. When swapping the position of two elements the handles stored at the nodes are swapped, which keeps the locators untouched and valid. Another way of implementing the abstraction is to avoid element moves altogether and move nodes instead.

Our goal is to develop realizations of a double-ended priority queue that support the following methods:

find-min(Q)/find-max(Q). Return a locator to an element that, of all elements in double-ended priority queue Q , has the minimum/maximum value. If Q is empty, return a *null* locator.

insert(Q, p). Insert an element indicated by locator p into double-ended priority queue Q .

extract(Q). Extract an *unspecified* element from double-ended priority queue Q and return a locator to that element. If Q is empty, return a *null* locator.

delete(Q, p). Remove the element indicated by locator p from double-ended priority queue Q .

meld(Q, R). Move all elements from double-ended priority queues Q and R to a new double-ended priority queue S , destroy Q and R , and return S .

Of these methods, *extract* is non-standard, but we expect it to be useful, for example, for data-structural transformations. We assume that the priority queues used by our transformations also support *extract*.

The following four methods may also be provided.

delete-min(Q)/delete-max(Q). Remove a minimum/maximum element from double-ended priority queue Q , and return a locator to that element. If Q is empty, return a *null* locator.

decrease(Q, p, x)/increase(Q, p, x). Replace the element indicated by locator p with element x , which is assumed to be no greater/smaller than the old element.

The method *delete-min* (resp. *delete-max*) can be carried out by invoking *find-min* (*find-max*) followed by *delete* with the locator returned by *find-min* (*find-max*). Similarly, the method *decrease* (resp. *increase*) can be implemented by invoking *delete* followed by *insert*. If *find-min*, *find-max*, and *insert* are required to have the worst-case cost of $O(1)$, the asymptotic computational complexity of *delete-min*, *delete-max*, *decrease*, and *increase* would be dominated by that of *delete*.

Any double-ended priority queue can be used for sorting (say, a set of size n). So if *find-min* (resp. *find-max*) and *insert* have a cost of $O(1)$, *delete-min* (*delete-max*)—and hence also *delete*—must perform at least $\lg n - O(1)$ element comparisons in the worst case in the decision-tree model. Similarly, as observed in [30], if *find-min* (resp. *find-max*) and *insert* have a cost of $O(1)$, *increase* (*decrease*) must perform at least $\lg n - O(1)$ element comparisons in the worst case. That is, specialized implementations of *delete-min*, *delete-max*, *decrease*, and *increase* cannot be much faster than the above-mentioned reductions, provided that *delete* is fast. Recall, however, that single-ended priority queues can support *find-min*, *insert*, and *decrease* (or *find-max*, *insert*, and *increase*) at the worst-case cost of $O(1)$ (see, for example, [7]).

Previous approaches

Most realizations of a (meldable) double-ended priority queue—but not all—use two priority queues, minimum priority queue Q_{min} and maximum priority queue Q_{max} , that contain the minimum and maximum candidates, respectively. The approaches to guarantee that a minimum element is in Q_{min} and a maximum element in Q_{max} can be classified into three main categories [13]: dual correspondence, total correspondence, and leaf correspondence. The correspondence between two nodes can be maintained implicitly, as done in many implicit or space-efficient data structures, or explicitly relying on pointers.

In the *dual correspondence* approach a copy of each element is kept both in Q_{min} and Q_{max} , and *clone pointers* are maintained between the corresponding copies. Using this approach Brodal [6] showed that *find-min*, *find-max*, *insert*, and *meld* can be realized at the worst-case cost of $O(1)$, and *delete* at the worst-case cost of $O(\lg n)$. Asymptotically, Brodal’s double-ended priority queue is optimal with respect to all operations. However, as pointed out by Cho and Sahni [12], the double-ended priority queue uses almost twice as much space as the single-ended priority queue, and the leading constant in the bound on the complexity of *delete* is high (according to our analysis the number of element comparisons performed in the worst case is at least $4 \lg n - O(1)$ for the priority queue and twice as much for the double-ended priority queue).

In the *total correspondence* approach, both Q_{min} and Q_{max} contain $\lfloor n/2 \rfloor$ elements and, if n is odd, one element is kept outside these structures. Every element x in Q_{min} has a *twin* y in Q_{max} , the element stored at x is no greater

than that stored at y , and there is a *twin pointer* from x to y and vice versa. Both Chong and Sahni [13] and Makris et al. [30] showed that with this approach the space efficiency of Brodal’s data structure can be improved. Now the elements are stored only once and the amount of extra space used is nearly cut in half. Actually, Makris et al. [30] relied on the simplification of Brodal’s data structure proposed by Fagerberg [20]. The results reported in [13, 30] are rephrased in Table 1 (on p. 6).

A third possibility is to employ the *leaf correspondence* approach, where only the leaves of the data structures used for realizing Q_{min} and Q_{max} have their corresponding twins. This approach is less general and requires that some type of tree is used to represent the two priority queues. Chong and Sahni [13] showed that Brodal’s data structure could be customized to rely on the leaf correspondence as well, but the worst-case complexity of *delete* is still about twice as high as that in the original priority queue.

In addition to these general transformations, several ad-hoc modifications of existing priority queues have been proposed. These modifications inherit their properties, like the operation repertoire and the space requirements, directly from the modified priority queue. Most notably, many of the double-ended priority queues proposed do not support general *delete* or *meld*, nor *insert* at the worst-case cost of $O(1)$.

For the adaptations of binary heaps—as twin heaps [28, Section 5.2.3, Exercise 31], min-max heaps [3], deaps [8] (see also [9]), interval heaps [29], diamond dequeues [11], symmetric min-max heaps [2], min-max-fine heaps [32], and min-max-pair heaps [35]—the primary concern has been simplicity and space-efficiency. Of the space-efficient double-ended priority queues, deaps, min-max-fine heaps, and min-max-pair heaps are the fastest: *find-min* and *find-max* have the worst-case cost of $O(1)$; *insert*, *delete-min*, and *delete-max* have the worst-case cost of $O(\lg n)$; *insert* performs at most $\lg \lg n + O(1)$ element comparisons; and *delete-min* and *delete-max* perform at most $\lg n + \lg \lg n + O(1)$ element comparisons. The main advantage of the adaptations of leftist trees [12, 13] and binomial queues [27] is that these can support *meld* faster than the adaptations of binary heaps. Even min-max heaps can be modified to support *meld* at logarithmic cost [15].

All known double-ended priority queues supporting *insert* at the worst-case cost of $O(1)$ are obtained via general transformations from priority queues supporting *insert* at the worst-case cost of $O(1)$. Such priority queues include those described in [6, 7, 10, 16, 17, 18, 19, 20, 24, 25]. Also, priority queues supporting *insert* at the worst-case cost of $O(\lg n)$ can be modified to provide it at the worst-case cost of $O(1)$, as shown by Alstrup et al. [1], but as a result *delete* would perform $\Theta(\lg n)$ additional element comparisons.

Efficient priority queues

Even though our data-structural transformations from priority queues to double-ended priority queues are general, to obtain our best results we rely on our earlier work on efficient priority queues [17, 19]. Our main goal in

these two earlier papers was to reduce the number of element comparisons performed by *delete* without sacrificing the asymptotic bounds for other supported operations. As a result, we can use these priority queues as building blocks to achieve the same goal for double-ended priority queues.

Both our data-structural transformations (described in Sections 2 and 3) require that the priority queue used provides the method *extract*, which extracts an unspecified element from the priority queue. This operation is used for moving elements from one priority queue to another and for reconstructing a priority queue incrementally.

The performance of the priority queues described in [17, 19] is summarized in the following theorems. Here and later on, m and n denote the number of elements stored in the data structures prior to the operation in question.

Theorem 1. [17] *There exists a priority queue that supports find-min , insert , and extract at the worst-case cost of $O(1)$; and delete at the worst-case cost of $O(\lg n)$ including at most $\lg n + O(1)$ element comparisons.*

Theorem 2. [19] *There exists a priority queue that supports find-min , insert , extract , and decrease at the worst-case cost of $O(1)$; delete at the worst-case cost of $O(\lg n)$ including at most $\lg n + O(\lg \lg n)$ element comparisons; and meld at the worst-case cost of $O(\min\{\lg m, \lg n\})$.*

Our results

In this paper we present two general transformations that show how priority queues can be employed to obtain double-ended priority queues. The data structure produced by our first transformation does not support *meld*, but it is nearly optimal with respect to all other operations. The data structure produced by our second transformation supports *meld*, but in order to apply the transformation the priority queue used should support both *decrease* and *meld*, so *delete* is not necessarily as efficient as that for the first one.

In our first transformation we divide the elements into three candidate collections and keep the candidates in their respective priority queues Q_{\min} , Q_{mid} , and Q_{\max} . We maintain a special element, called *pivot*, such that all elements stored in Q_{\min} are smaller than *pivot*, all elements stored in Q_{mid} are equal to *pivot*, and all elements stored in Q_{\max} are greater than *pivot*. It turns out to be cheaper to maintain a single pivot element than to maintain many twin relationships as done in the correspondence-based approaches. When developing this transformation we were inspired by the priority queue described in [31], where a related partitioning scheme is used. The way we use partitioning allows efficient deamortization; in accordance our bounds are worst-case rather than amortized in contrast to the bounds derived in [31]. Our second transformation combines the total correspondence approach with an efficient priority queue supporting *decrease*. This seems to be a new application of priority queues supporting *decrease*.

The characteristics of our data-structural transformations are summarized in Table 1. By applying the transformations described in Sections 2 and 3

Table 1. Characteristics of general transformations from priority queues to double-ended priority queues. We let C_{op}^n denote the worst-case cost of double-ended priority-queue operation op for a given problem size n (which is an upper bound), and c_{op}^n the corresponding cost of the priority-queue operation op . Throughout the paper we assume that functions c_{op}^n are non-decreasing and smooth, i.e. that for non-negative integers m and n , $m \leq n \leq 2m$, $c_{op}^m \leq c_{op}^n \leq O(1) \cdot c_{op}^m$. Naturally, if $c_{find-max}^n = c_{find-min}^n$, then $C_{find-max}^n = C_{find-min}^n$.

Reference Cost	[13, 30]	this paper, Section 2	this paper, Section 3
$C_{find-min}^n$	$c_{find-min}^{n/2} + O(1)$	$2 \cdot c_{find-min}^n + O(1)$	$c_{find-min}^{n/2} + O(1)$
C_{insert}^n	$2 \cdot c_{insert}^{n/2} + O(1)$	$O(1) \cdot c_{extract}^n + O(1) \cdot c_{insert}^n + O(1)$	$2 \cdot c_{insert}^{n/2} + O(1)$
$C_{extract}^n$	not supported	$O(1) \cdot c_{extract}^n + O(1) \cdot c_{insert}^n + O(1)$	$2 \cdot c_{extract}^{n/2} + 2 \cdot c_{decrease}^{n/2} + O(1)$
C_{delete}^n	$2 \cdot c_{delete}^{n/2} + 2 \cdot c_{insert}^{n/2} + O(1)$	$c_{delete}^n + O(1) \cdot c_{extract}^n + O(1) \cdot c_{insert}^n + O(1)$	$c_{delete}^{n/2} + c_{extract}^{n/2} + c_{insert}^{n/2} + 2 \cdot c_{decrease}^{n/2} + O(1)$
$C_{meld}^{m,n}$	$2 \cdot c_{meld}^{\lceil m/2 \rceil, n/2} + 2 \cdot c_{insert}^{m/2} + O(1)$	not supported	$2 \cdot c_{meld}^{\lceil m/2 \rceil, n/2} + 2 \cdot c_{insert}^{m/2} + O(1)$

to the priority queues mentioned in Theorems 1 and 2, respectively, we get double-ended priority queues, the performance of which is summarized in the following theorems.

Theorem 3. *There exists a double-ended priority queue that supports $find-min$, $find-max$, $insert$, and $extract$ at the worst-case cost of $O(1)$; and $delete$ at the worst-case cost of $O(\lg n)$ including at most $\lg n + O(1)$ element comparisons.*

Theorem 4. *There exists a double-ended priority queue that supports $find-min$, $find-max$, $insert$, and $extract$ at the worst-case cost of $O(1)$; $delete$ at the worst-case cost of $O(\lg n)$ including at most $\lg n + O(\lg \lg n)$ element comparisons; and $meld$ at the worst-case cost of $O(\min \{\lg m, \lg n\})$.*

2. Pivot-based partitioning

In this section we show how a partitioning scheme can be used to transform a priority queue into a double-ended priority queue Q . The basic idea is to maintain a special *pivot* element and use it to partition the set of elements into three candidate collections: Q_{min} holds the elements that are smaller than *pivot*, Q_{mid} holds the elements that are equal to *pivot*, and Q_{max} holds the elements that are larger than *pivot*. Note that, even if the underlying priority queue is able to carry out *meld*, the resulting double-ended priority queue cannot provide *meld* efficiently.

To illustrate the general idea, let us first consider a realization that guarantees good amortized performance for all the modifying operations (*insert*, *extract*, and *delete*). We divide the execution of the operations into phases. Each phase consists of $\max\{1, \lfloor n_0/2 \rfloor\}$ operations, if at the beginning of a phase the data structure stored n_0 elements. At the beginning of a phase, a restructuring is done by repartitioning the elements using the median element (the $\lfloor n_0/2 \rfloor$ th smallest element) as *pivot*. That way we ensure that at any given time—except if there are no elements in Q —the minimum (resp. maximum) element is in Q_{min} (Q_{max}), or if it is empty in Q_{mid} , and hence all the operations of a phase can be performed safely.

To perform the partitioning efficiently, all the elements are copied to a temporary array A . This is done by employing *extract* to repeatedly remove elements from Q . Each element is copied to A and temporarily inserted into another data structure P for later use (P should efficiently support *extract*). We chose to implement P as a priority queue to reuse the same structure of the nodes as Q . A linear-time selection algorithm [4] is then used to set *pivot* to the value of the median element in array A . Actually, we rely on a space-efficient variant of the standard prune-and-search algorithm described in [23, Section 3.6]. For an input of size n , the extra space used by this variant is $O(\lg n)$ words. The array A is then destructed, and Q (which is now empty) is reconstructed by repeatedly re-extracting the elements from the temporary structure P and inserting them into Q (using the above-mentioned partitioning scheme and the new *pivot* that has just been found).

Assuming that priority-queue operations *insert* and *extract* have a cost of $O(1)$, the restructuring done at the beginning of a phase has the worst-case cost of $O(n_0)$. So a single modifying operation can be expensive, but when the reorganization work is amortized over the $\max\{1, \lfloor n_0/2 \rfloor\}$ operations executed in a phase, the amortized cost is only $O(1)$ per modifying operation.

In the following pseudo code the implementation details for this amortized scheme are given. We assume that each node of the priority queue used can be augmented by an extra field that gives the name of the component, Q_{min} , Q_{mid} , or Q_{max} , in which the corresponding element is stored. The routine *component*(Q, p) returns the component of Q in which the element with locator p is stored.

find-min(Q): // *find-max*(Q) is similar

```

if size( $Q_{min}$ ) = 0
  return find-min( $Q_{mid}$ )
return find-min( $Q_{min}$ )

```

insert(Q, p):

```

if  $p.element() < pivot$ 
  insert( $Q_{min}, p$ )
else if  $pivot < p.element()$ 
  insert( $Q_{max}, p$ )
else
  insert( $Q_{mid}, p$ )
count++
if count  $\geq \lfloor n_0/2 \rfloor$ 
  reorganize( $Q$ )

```

```

extract(Q):
  p ← extract(Qmin)
  if p = null
    p ← extract(Qmid)
  count++
  if count ≥ ⌊n0/2⌋
    reorganize(Q)
  return p

delete(Q, p):
  R ← component(Q, p)
  delete(R, p)
  count++
  if count ≥ ⌊n0/2⌋
    reorganize(Q)

reorganize(Q):
  n0 ← size(Q)
  count ← 0
  construct an empty priority queue P
  allocate array A of size n0 for storing elements
  i ← 1
  for R ∈ {Qmin, Qmid, Qmax}
    for j ∈ {1, ..., size(R)}
      p ← extract(R)
      A[i++] ← p.element()
      insert(P, p)
  pivot ← selection(A[1: n0], ⌈n0/2⌉)
  destroy A
  for i ∈ {1, ..., n0}
    p ← extract(P)
    if p.element() < pivot
      insert(Qmin, p)
    else if pivot < p.element()
      insert(Qmax, p)
    else
      insert(Qmid, p)

```

Next we consider how we can get rid of the amortization. In our deamortization strategy, each phase consists of $\max\{1, \lfloor n_0/4 \rfloor\}$ operations. We maintain the *size invariant* that at the beginning of a phase, if there are n_0 elements in total, the size of Q_{min} (resp. Q_{max}) plus the size of Q_{mid} is at least $\max\{1, \lfloor n_0/4 \rfloor\}$ elements. This guarantees that the minimum (resp. maximum) element is in Q_{min} (Q_{max}), or if it is empty in Q_{mid} , and hence all operations can be performed safely. Throughout each phase, three subphases are performed in sequence.

In the first subphase, the n_0 elements contained in Q at the beginning of the phase are to be incrementally copied to A . To facilitate this, we employ a supplementary double-ended priority queue Q^T that is structured in the same way as Q and composed of three components Q_{min}^T , Q_{mid}^T , and Q_{max}^T . Accompanying each modifying operation, an adequate number of elements is copied from Q to A . This copying is accomplished by extracting an element from Q , copying the element to A , and inserting the node into Q^T using the same *pivot* as Q . An *insert* would directly insert the given element into Q^T without copying it to A , and a *delete* would copy the deleted element to A

only if that element was in Q and not Q^T . At the end of this subphase, A stores copies of all the elements that were in Q at the beginning of the phase, and all the elements that should be in the double-ended priority queue are now in Q^T leaving Q empty.

In the second subphase, the median of the elements of A is found. That is, at the beginning of this subphase, the content of A is frozen and the selection is based on its current content. In this subphase the modifying operations are executed normally, except that they are performed on Q^T , while incrementally taking part in the selection process. Each modifying operation takes its own share of the work, such that the whole selection process is finished before the end of this subphase.

The third subphase is reserved for clean-up. The modifying operations carry out their share of the work such that the whole clean-up process is finished before the end of the phase. First, array A is destroyed by gradually destructing the elements copied. Second, all elements held in Q^T are moved to Q . When moving the elements, *extract* and *insert* are used, and the median found in the second subphase is used as the partitioning element.

As to *find-min*, the current minimum can be found from one of the priority queues Q_{min} (if it is empty in Q_{mid}) or Q_{min}^T (if it is empty in Q_{mid}^T). Hence, *find-min* (similarly for *find-max*) can still be carried out efficiently.

Even if the median found is exact for A , it is only an approximate median for the whole collection at the end of a phase. Since after freezing A at most $\max\{1, \lfloor n_0/4 \rfloor\}$ elements are added to or removed from the data structure, it is easy to verify that the size invariant holds for the next phase.

Finally, let us analyse the space requirements of the deamortized construction. Let n denote the present size of the double-ended priority queue, and n_0 the size of A . The worst case is when all the operations performed during the phase are deletions, and hence n may be equal to $3n_0/4$. This means that the array can never have more than $4n/3$ elements.

For the deamortized scheme, the worst-case bounds for the costs of the operations are similar to those for the amortized scheme, but the constant factors involved are larger. By combining the bounds derived for the transformation (Table 1) with the bounds known for the priority queues (Theorem 1), the bounds of Theorem 3 are obtained.

3. Total correspondence

In this section we describe how an efficient priority queue supporting *extract* and *decrease*, in addition to other standard operations, can be utilized in the realization of a meldable double-ended priority queue Q . We use Q_{min} to denote the minimum priority queue of Q (supporting *find-min* and *decrease*) and Q_{max} the maximum priority queue of Q (supporting *find-max* and *increase*). We rely on the total correspondence approach, where each of Q_{min} and Q_{max} contains $\lfloor n/2 \rfloor$ elements, with the possibility of having one element outside these structures. To perform *delete* efficiently, instead of

using two priority-queue *delete* operations as in [13, 30], we use only one *delete* and employ *extract* that may be followed by *decrease* and *increase*.

The priority queues used should allow the attachment of one more pointer, a *twin pointer*, to each node. We assume that the routine *make-twins*(p, q) assigns the twin pointers between the elements indicated by locators p and q , and that the routine *twin*(p) returns the twin of the element with locator p . An element that has no twin is called a *singleton*. We assume that the routine *make-singleton*(Q, p) makes the element with locator p the singleton of Q and sets the twin pointer of p to *null*, and that the routine *singleton*(Q) returns a locator to the singleton of Q . In addition, we assume that the effect of the routine *swap*(p, q) is that the element with locator p takes the place of the element with locator q , and vice versa. Finally, the routine *component*(Q, p) returns the component of Q in which the element with locator p is stored. One way of implementing this is to attach to each node of the priority queue a bit indicating whether a node is in Q_{min} or in Q_{max} , and let *insert* update these bits.

```

find-min( $Q$ ): // find-max( $Q$ ) is similar
   $s \leftarrow \text{singleton}(Q)$ 
   $t \leftarrow \text{find-min}(Q_{min})$ 
  if  $t = \text{null}$  or ( $s \neq \text{null}$  and  $s.\text{element}() < t.\text{element}()$ )
    return  $s$ 
  return  $t$ 

```

```

insert( $Q, p$ ):
   $s \leftarrow \text{singleton}(Q)$ 
  if  $s = \text{null}$ 
    make-singleton( $Q, p$ )
  return
  if  $p.\text{element}() < s.\text{element}()$ 
    insert( $Q_{min}, p$ )
    insert( $Q_{max}, s$ )
  else
    insert( $Q_{min}, s$ )
    insert( $Q_{max}, p$ )
  make-twins( $p, s$ )
  make-singleton( $Q, \text{null}$ )

```

```

extract( $Q$ ):
   $s \leftarrow \text{singleton}(Q)$ 
  if  $s \neq \text{null}$ 
    make-singleton( $Q, \text{null}$ )
  return  $s$ 
else
   $p \leftarrow \text{extract}(Q_{min})$ 
  if  $p \neq \text{null}$ 
     $q \leftarrow \text{extract}(Q_{max})$ 
     $r \leftarrow \text{twin}(p)$ 
     $t \leftarrow \text{twin}(q)$ 
    make-twins( $r, t$ )
    swap-twins-if-necessary( $Q, t$ )
    make-singleton( $Q, q$ )
  return  $p$ 

```

```

delete(Q, p):
    s ← singleton(Q)
    if p = s
        make-singleton(Q, null)
    return
P ← component(Q, p)
t ← twin(p)
T ← component(Q, t)
if s = null
    q ← extract(T)
    s ← twin(q)
    make-singleton(Q, q)
else
    insert(P, s)
    make-singleton(Q, null)
make-twins(s, t)
if P = Qmin
    swap-twins-if-necessary(Q, s)
else
    swap-twins-if-necessary(Q, t)
delete(P, p)

swap-twins-if-necessary(Q, s): // s must be in Qmin
    t ← twin(s)
    if t.element() < s.element()
        swap(s, t)
        decrease(Qmin, t, t.element())
        increase(Qmax, s, s.element())

meld(Q, R):
    q ← singleton(Q)
    r ← singleton(R)
    if q ≠ null and r = null
        make-singleton(S, q)
    else if q = null and r ≠ null
        make-singleton(S, r)
    else
        make-singleton(S, null)
    if q ≠ null
        if q.element() < r.element()
            insert(Qmin, q)
            insert(Qmax, r)
        else
            insert(Qmin, r)
            insert(Qmax, q)
        make-twins(q, r)
    Smin ← meld(Qmin, Rmin)
    Smax ← meld(Qmax, Rmax)
    destroy Q
    destroy R
    return S
    
```

It is straightforward to verify that the above methods maintain the bounds of Table 1. By combining the bounds in Table 1 with the bounds known for the priority queues (Theorem 2), the bounds of Theorem 4 are obtained.

4. Conclusions

Our primary concern was to develop realizations of a (meldable) double-ended priority queue, for which the number of element comparisons performed by *insert* and *delete* is nearly optimal $O(1)$ and $\lg n + O(1)$, respectively. We were able to achieve this goal when efficient *meld* is not to be supported, which improves the complexity of all previous double-ended priority-queue structures, even those supporting *insert* at a cost of $\Theta(\lg n)$. If *meld* has to be supported, in our realization of a meldable double-ended priority queue *delete* performs at most $\lg n + O(\lg \lg n)$ element comparisons, and *meld* is still relatively fast having the worst-case cost of $O(\min \{\lg m, \lg n\})$.

We conclude the paper with three open problems, the solution of which would improve the results presented in this paper.

1. One drawback of our first transformation is the extra space used for elements, and the extra element constructions and destructions performed when copying elements. The reason for copying elements instead of pointers is that some elements may be deleted during the selection process. It would be interesting to know whether the selection problem could still be solved at linear cost when the input is allowed to be modified during the computation.
2. Our realization of a double-ended priority queue using the priority queues introduced in [17] works on a pointer machine (as defined, for example, in [34]), but the meldable version using the priority queues introduced in [19] relies on the capabilities of a RAM. This is in contrast with Brodal's data structure [6] which works on a pointer machine. Therefore, it is natural to ask whether random access could be avoided.
3. To obtain *meld* having the worst-case cost of $O(1)$, the price paid by Brodal [6] is a more expensive *delete*. It is unknown whether *meld* could be implemented at the worst-case cost of $O(1)$ such that at most $\lg n + o(\lg n)$ element comparisons are performed per *delete*.
4. If for a meldable double-ended priority queue *meld* is allowed to have the worst-case cost of $O(\min \{\lg m, \lg n\})$, it is still relevant to ask whether *delete* can be accomplished at logarithmic cost with at most $\lg n + O(1)$ element comparisons.

References

- [1] S. Alstrup, T. Husfeld, and T. Rauhe. Marked ancestor problems. Technical Report 98-8. Department of Computer Science, University of Copenhagen (1998). (The data-structural transformation referred to is not described in the conference version presented at FOCS'98.)
- [2] A. Arvind and C. Pandu Rangan. Symmetric min-max heap: A simpler data structure for double-ended priority queue. *Information Processing Letters* **69** (1999), 197–199.
- [3] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM* **29** (1986), 996–1000.
- [4] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences* **7** (1973), 448–461.

- [5] British Standards Institute. *The C++ Standard: Incorporating Technical Corrigendum 1*, BS ISO/IEC 14882:2003, 2nd Edition. John Wiley and Sons, Ltd. (2003).
- [6] G.S. Brodal. Fast meldable priority queues. *Proceedings of the 4th International Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science* **955**. Springer-Verlag (1995), 282–290.
- [7] G.S. Brodal. Worst-case efficient priority queues. *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1996), 52–58.
- [8] S. Carlsson. The deap—A double-ended heap to implement double-ended priority queues. *Information Processing Letters* **26** (1987), 33–36.
- [9] S. Carlsson, J. Chen, and T. Strothotte. A note on the construction of data structure “deap”. *Information Processing Letters* **31** (1989), 315–317.
- [10] S. Carlsson, J. I. Munro, and P. V. Poblete. An implicit binomial queue with constant insertion time. *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science* **318**. Springer-Verlag (1988), 1–13.
- [11] S. C. Chang and M. W. Du. Diamond deque: A simple data structure for priority dequeues. *Information Processing Letters* **46** (1993), 231–237.
- [12] S. Cho and S. Sahni. Mergeable double-ended priority queues. *International Journal of Foundations of Computer Science* **10** (1999), 1–18.
- [13] K.-R. Chong and S. Sahni. Correspondence-based data structures for double-ended priority queues. *The ACM Journal of Experimental Algorithmics* **5** (2000), article 2.
- [14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press (2001).
- [15] Y. Ding and M. A. Weiss. The relaxed min-max heap: A mergeable double-ended priority queue. *Acta Informatica* **30** (1993), 215–231.
- [16] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM* **31** (1988), 1343–1354.
- [17] A. Elmasry, C. Jensen, and J. Katajainen. A framework for speeding up priority-queue operations. CPH STL Report 2004-3. Department of Computing, University of Copenhagen (2004). Available at <http://cphstl.dk>.
- [18] A. Elmasry, C. Jensen, and J. Katajainen. Relaxed weak queues: An alternative to run-relaxed heaps. CPH STL Report 2005-2. Department of Computing, University of Copenhagen (2005). Available at <http://cphstl.dk>.
- [19] A. Elmasry, C. Jensen, and J. Katajainen. Two-tier relaxed heaps. CPH STL Report 2006-6. Department of Computing, University of Copenhagen (2006). Available at <http://cphstl.dk>. (Operations *extract* and *meld* are not described in the conference version presented at ISAAC’06.)
- [20] R. Fagerberg. A note on worst case efficient meldable priority queues. Technical Report PP-1996-12. Department of Mathematics and Computer Science, Odense University (1996).
- [21] M. T. Goodrich and R. Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, Inc. (2002).
- [22] T. Hagerup. Sorting and searching on the word RAM. *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science* **1373**. Springer-Verlag (1998), 366–398.
- [23] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Computer Science Press (1998).
- [24] P. Høyer. A general technique for implementation of efficient priority queues. *Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems*. IEEE (1995), 57–66.
- [25] H. Kaplan, N. Shafir, and R. E. Tarjan. Meldable heaps and Boolean union-find. *Proceedings of 34th Annual ACM Symposium on Theory of Computing*. ACM (2002), 573–582.
- [26] J. Katajainen and F. Vitale. Navigation piles with applications to sorting, priority queues, and priority dequeues. *Nordic Journal of Computing* **10** (2003), 238–262.
- [27] C. M. Khoong and H. W. Leong. Double-ended binomial queues. *Proceedings of the*

- 4th International Symposium on Algorithms and Computation, Lecture Notes in Computer Science* **762**. Springer-Verlag (1993), 128–137.
- [28] D.E. Knuth. *Sorting and Searching, The Art of Computer Programming* **3**, 2nd Edition. Addison Wesley Longman (1998).
- [29] J. van Leeuwen and D. Wood. Interval heaps. *The Computer Journal* **36** (1993), 209–216.
- [30] C. Makris, A. Tsakalidis, and K. Tsihlias. Reflected min-max heaps. *Information Processing Letters* **86** (2003), 209–214.
- [31] C.W. Mortensen and S. Pettie. The complexity of implicit and space-efficient priority queues. *Proceedings of the 9th Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science* **3608**. Springer-Verlag (2005), 49–60.
- [32] S.K. Nath, R. A. Chowdhury, and M. Kaykobad. Min-max fine heaps. arXiv.org e-Print archive, article cs.DS/0007043 (2000). Available at <http://arxiv.org>.
- [33] S. Olariu, C.M. Overstreet, and Z. Wen. A mergeable double-ended priority queue. *The Computer Journal* **34** (1991), 423–427.
- [34] M. Penttonen and J. Katajainen. Notes on the complexity of sorting in abstract machines. *BIT* **25** (1985), 611–622.
- [35] M.Z. Rahman, R. A. Chowdhury, and M. Kaykobad. Improvements in double ended priority queues. *International Journal of Computer Mathematics* **80** (2003), 1121–1129.