Computing (2008) 83:193–204
DOI 10.1007/s00607-008-0019-2

Two new methods for constructing double-ended priority queues from priority queues

Amr Elmasry · Claus Jensen · Jyrki Katajainen

Received: 10 August 2007 / Accepted: 2 October 2008 / Published online: 19 November 2008

Abstract We introduce two data-structural transformations to construct double-ended priority queues from priority queues. To apply our transformations the priority queues exploited must support the extraction of an unspecified element, in addition to the standard priority-queue operations. With the first transformation we obtain a double-ended priority queue which guarantees the worst-case cost of $O(1)$ for *find-min*, *find-max*, *insert*, *extract*; and the worst-case cost of $O(\lg n)$ with at most $\lg n + O(1)$ element comparisons for *delete*. With the second transformation we get a meldable double-ended priority queue which guarantees the worst-case cost of $O(1)$ for *find-min*, *find-max*, *insert*, *extract*; the worst-case cost of $O(\lg n)$ with 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.

Keywords Data structures · Priority queues · Double-ended priority queues · Min-max priority queues · Priority deques · Meticulous analysis · Comparison complexity

Mathematics Subject Classification (2000) 68P05 · 68P10 · 68W40 · 68Q25

The work of the authors was 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). A. Elmasry was supported by Alexander von Humboldt fellowship.

A. Elmasry
Max-Planck Institut für Informatik, Saarbrücken, Germany

C. Jensen · J. Katajainen (✉)
Datalogisk Institut, Københavns Universitet, Universitetsparken 1, 2100 Copenhagen East, Denmark
e-mail: jyrki@diku.dk

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*, and *delete*. For single-ended priority queues only *find-min* or *find-max* is supported. A (double-ended) priority queue is called *meldable* if it also supports operation *meld*. Even though the data-structural transformations to be presented are fully general, our main focus is on the comparison complexity of double-ended priority-queue operations. Throughout this paper we use m and n to denote the number of elements stored in a data structure prior to an operation and $\lg n$ as a shorthand for $\log_2(\max\{2, n\})$. Many data structures [2,3,5,7–11,17–21,23–25] have been proposed for the realization of a double-ended priority queue, but none of them achieve $\lg n + o(\lg n)$ element comparisons per *delete*, if *find-min*, *find-max*, and *insert* must have the worst-case cost of $O(1)$.

We use the *word RAM* as our model of computation as defined in [15]. We assume the availability of instructions found in contemporary computers, including built-in functions for allocating and freeing memory. 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 [14]. 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.

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

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

insert(Q, p). Add an element with locator p to 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 with locator p from double-ended priority queue Q (without destroying the element).

Of these operations, *extract* is non-standard, but we are confident that it is useful, for example, for data-structural transformations. The following operations may also be provided.

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.

decrease(Q, p, x) (*increase*(Q, p, x)). Replace the element with locator p by element x, which should not be greater (smaller) than the old element.

Any double-ended priority queue can be used for sorting (say, a set of size n). So if *find-min* (*find-max*) and *insert* have a cost of $O(1)$, *delete* must perform at least $\lg n - O(1)$ element comparisons in the worst case in the decision-tree model. Similarly, as observed in [21], if *find-min* (*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. 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, [6]).

1.1 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 [10]: dual correspondence, total correspondence, and leaf correspondence. The correspondence between two elements 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 [5] 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 [9], Brodal's double-ended priority queue uses almost twice as much space as his 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 $8 \lg n - O(1)$ 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}, that 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 [10] and Makris et al. [21] showed that with this approach the space efficiency of Brodal's data structure can be improved. Now the elements are stored only once so the amount of extra space used is nearly cut in half. The results reported in [10,21] are rephrased in Table 1.

A third possibility is to employ the *leaf correspondence* approach, where only the elements stored at 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 [10] 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.

Table 1 Complexity of general transformations from priority queues to double-ended priority queues

Complexity	Reference		
	[10,21]	This paper, Sect. 2	This paper, Sect. 3
$C^n_{find\text{-}min}$	$c^{n/2}_{find\text{-}min} + O(1)$	$2 \cdot c^n_{find\text{-}min} + O(1)$	$c^{n/2}_{find\text{-}min} + O(1)$
C^n_{insert}	$2 \cdot c^{n/2}_{insert} + O(1)$	$O(1) \cdot c^n_{extract}$ $+O(1) \cdot c^n_{insert} + O(1)$	$2 \cdot c^{n/2}_{insert} + O(1)$
$C^n_{extract}$	Not supported	$O(1) \cdot c^n_{extract}$ $+O(1) \cdot c^n_{insert} + O(1)$	$2 \cdot c^{n/2}_{extract} + 2 \cdot c^{n/2}_{decrease} + O(1)$
C^n_{delete}	$2 \cdot c^{n/2}_{delete}$ $+2 \cdot c^{n/2}_{insert} + O(1)$	$c^n_{delete} + O(1) \cdot c^n_{extract}$ $+O(1) \cdot c^n_{insert} + O(1)$	$c^{n/2}_{delete} + c^{n/2}_{extract} + c^{n/2}_{insert}$ $+2 \cdot c^{n/2}_{decrease} + O(1)$
$C^{m,n}_{meld}$	$2 \cdot c^{\lceil m/2 \rceil, n/2}_{meld}$ $+2 \cdot c^{m/2}_{insert} + O(1)$	Not supported	$2 \cdot c^{\lceil m/2 \rceil, n/2}_{meld} + 2 \cdot c^{m/2}_{insert} + O(1)$
Extra space	$n + O(1)$ words	$(4/3)n$ elements $O(\lg n)$ words $3n$ bits	$n + O(1)$ words n bits

Here C^n_{op} denotes the worst-case cost of double-ended priority-queue operation *op* for a given problem size n, and c^n_{op} the corresponding cost of the priority-queue operation *op*. Throughout the paper, we assume that functions c^n_{op} are non-decreasing and smooth, i.e. that for non-negative integers m and n, $m \leq n \leq 2m$, $c^m_{op} \leq c^n_{op} \leq O(1) \cdot c^m_{op}$. Naturally, if $c^n_{find\text{-}max} = c^n_{find\text{-}min}$, then $C^n_{find\text{-}max} = C^n_{find\text{-}min}$

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*, *meld*, nor *insert* at the worst-case cost of $O(1)$. Even when such priority queues can be modified to provide *insert* at the worst-case cost of $O(1)$, as shown by Alstrup et al. [1], *delete* would perform $\Theta(\lg n)$ additional element comparisons as a result.

1.2 Efficient priority queues

Our data-structural transformations are general, but to obtain our best results we rely on our earlier work on efficient priority queues [12,13]. 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 the other supported operations. In this paper, we use these priority queues as building blocks to achieve the same goal for double-ended priority queues.

Both our data-structural transformations require that the priority queues used support *extract*, which extracts an unspecified element from the given priority queue and returns a locator to that element. This operation is used for moving elements from

one priority queue to another and for reconstructing a priority queue incrementally. Many existing priority queues can be easily extended to support *extract*. When this is not immediately possible, the borrowing technique presented in [12,13] may be employed.

The performance of the priority queues described in [12,13] is summarized in the following lemmas.

Lemma 1 [12] *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.*

Lemma 2 [13] *There exists a meldable 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* $O(\min\{\lg m, \lg n\})$.

1.3 Our results

In this paper, we present two transformations that show how priority queues can be employed to obtain double-ended priority queues. With our first transformation we obtain a data structure for which all fundamental operations are nearly optimal with respect to the number of element comparisons performed. With our second transformation we obtain a data structure that also supports *meld*.

In our first transformation, we divide the elements into three collections containing elements smaller than, equal to, and greater than a partitioning element. It turns out to be cheaper to maintain a single partitioning 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 [22], where a related partitioning scheme is used. The way we implement partitioning allows efficient deamortization; in accordance our bounds are worst-case rather than amortized in contrast to the bounds derived in [22]. 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 fast *decrease*.

The complexity bounds attained are summarized in Table 1. The main difference between the earlier results and our results is that the leading constant in the cost of *delete* is reduced from two to one, provided that the priority queues used support *insert*, *extract*, and *decrease* at the worst-case cost of O(1). By constructing double-ended priority queues from the priority queues mentioned in Lemmas 1 and 2, respectively, we get the following theorems.

Theorem 1 *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 2 *There exists a meldable 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 double-ended priority queue, call it Q, can be constructed with the help of three priority queues Q_{min}, Q_{mid}, and Q_{max}. The basic idea is to maintain a special *pivot* element and use it to partition the elements held in Q into three candidate collections: Q_{min} holding the elements smaller than *pivot*, Q_{mid} those equal to *pivot*, and Q_{max} those larger than *pivot*. Note that, even if the priority queues are meldable, 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 modifying operations (*insert*, *extract*, and *delete*). We divide the execution of the operations into phases. Each phase consists of max $\{1, \lfloor n_0/2 \rfloor\}$ modifying operations, if at the beginning of a phase the data structure stored n_0 elements (initially, $n_0 = 0$). At the end of each phase, a restructuring is done by partitioning the elements using the median element as *pivot*. That way we ensure that at any given time—except if there are no elements in Q—the minimum (maximum) element is in Q_{min} (Q_{max}) or, if it is empty, in Q_{mid}. Thus all operations of a phase can be performed correctly.

Now it is straightforward to carry out the double-ended priority-queue operations by relying on the priority-queue operations.

find-min(Q) (*find-max*(Q)). If Q_{min} (Q_{max}) is non-empty, the minimum (maximum) of Q_{min} (Q_{max}) is returned; otherwise, the minimum of Q_{mid} is returned.

insert(Q, p). If the element with locator p is smaller than, equal to, or greater than *pivot*, the element is inserted into Q_{min}, Q_{mid}, or Q_{max}, respectively.

extract(Q). If Q_{min} is non-empty, an element is extracted from Q_{min}; otherwise, an element is extracted from Q_{mid}.

delete(Q, p). Depending on in which component Q_{min}, Q_{mid}, or Q_{max} the element with locator p is stored, the element is removed from that component. To implement this efficiently, we assume that each node of the priority queues is augmented by an extra field that gives the name of the component in which that node is stored.

A more detailed description of the operations on Q is given in Fig. 1.

After each modifying operation it is checked whether the end of a phase is reached, and if this is the case, a partitioning is carried out. To perform the partitioning efficiently, all the current 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. 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 [16, Sect. 3.6]. For an input of size n, the extra space used by this variant is $O(\lg n)$ words. After partitioning, A is destroyed,

```
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 ≥ ⌊n_0/2⌋
    reorganize(Q)
extract(Q):
  p ← extract(Q_min)
  if p = null
    p ← extract(Q_mid)
  count++
  if count ≥ ⌊n_0/2⌋
    reorganize(Q)
  return p
delete(Q, p):
  R ← component(Q, p)
  delete(R, p)
  count++
  if count ≥ ⌊n_0/2⌋
    reorganize(Q)
```

```
reorganize(Q):
  n_0 ← size(Q)
  count ← 0
  construct an empty priority queue P
  allocate an element array A of size n_0
  i ← 1
  for R ∈ {Q_min, Q_mid, Q_max}
    for j ∈ {1, ..., size(R)}
      p ← extract(R)
      A[i++] ← p.element()
      insert(P, p)
  pivot ← selection(A[1: n_0], ⌈n_0/2⌉)
  destroy A
  for i ∈ {1, ..., n_0}
    p ← extract(P)
    if p.element() < pivot
      insert(Q_min, p)
    else if pivot < p.element()
      insert(Q_max, p)
    else
      insert(Q_mid, p)
```

Fig. 1 This pseudo-code implements the amortized scheme. The subroutine *component*(Q, p) is assumed to return the component of Q in which the element with locator p is stored

and Q is reconstructed by repeatedly re-extracting the elements from the temporary structure P and inserting them into Q (using the new *pivot*).

Assuming that priority-queue operations *insert* and *extract* have a cost of $O(1)$, the restructuring done at the end 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.

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

In the first subphase, the n_0 elements in Q at the beginning of the phase are to be incrementally copied to A. To facilitate this, we employ a supplementary data structure P that has the same form as Q and is composed of three components P_{min}, P_{mid}, and P_{max}. Accompanying each modifying operation, an adequate number of elements is copied from Q to A. This is accomplished by extracting an element from Q, copying

the element to A, and inserting the node into P using the same *pivot* as in Q. An *insert* directly adds the given element to P without copying it to A, and a *delete* copies the deleted element to A only if that element is in Q. 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 P leaving Q empty.

In the second subphase the median of the elements of A is found. That is, the selection is based on the contents of Q at the beginning of the phase. In this subphase the modifying operations are executed normally, except that they are performed on P, while they incrementally take part in the selection process. Each modifying operation takes its own share of the work such that the whole selection process is finished before reaching the end of this subphase.

The third subphase is reserved for clean-up. Each modifying operation carries out its share of the work such that the whole clean-up process is finished before the end of the phase. First, A is destroyed by gradually destructing the elements copied and freeing the space after that. Second, all elements held in P 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} (Q_{mid} if Q_{min} is empty) or P_{min} (P_{mid} if P_{min} is empty). 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 the contents of 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.

Let us now 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 $3n_0/4$. That is, the array can never have more than $4n/3$ elements. In addition to array A, the selection routine requires a work area of $O(\lg n)$ words and each node has to store 3 bits to record the component (Q_{min}, Q_{mid}, Q_{max}, P_{min}, P_{mid}, or P_{max}) the node is in.

For the deamortized scheme, it is straightforward to derive the bounds given in Table 1. By combining the results derived for the transformation (Table 1) with the bounds known for the priority queues (Lemma 1), the bounds of Theorem 1 are obtained.

3 Total correspondence

In this section we describe how an efficient meldable priority queue supporting *extract* and *decrease* 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 [10,21], we use only one *delete* and employ *extract* that may be followed by *decrease* and *increase*.

Each element stored in Q_{min} has a *twin* in Q_{max}. To maintain the twin relationships and to access twins fast, we assume that each node of the priority queues allows an attachment of one more pointer, a *twin pointer*. The element that has no twin is called a *singleton*. Using these concepts, the double-ended priority-queue operations on Q can be performed as follows (a more detailed description of the operations is given in Fig. 2):

find-min(Q) (*find-max*(Q)). If Q_{min} (Q_{max}) is empty or the singleton of Q is smaller (greater) than the minimum (maximum) element of Q_{min} (Q_{max}), the singleton is returned; otherwise, the minimum (maximum) of Q_{min} (Q_{max}) is returned.

insert(Q, p). If Q has no singleton, the element with locator p is made the singleton of Q and nothing else is done. If Q has a singleton and the given element is smaller than the singleton, the element is inserted into Q_{min} and the singleton is inserted into Q_{max}; otherwise, the element is inserted into Q_{max} and the singleton is inserted into Q_{min}. Finally, the element and the singleton are made twins of each other.

extract(Q). If Q has a singleton, it is extracted and nothing else is done. If Q has no singleton, an element is extracted from Q_{min}. If Q_{min} was non-empty, an element is also extracted from Q_{max}. The element extracted from Q_{max} is made the new singleton, and the element extracted from Q_{min} is returned. Before this the twins of the two extracted elements are made twins of each other and their positions are swapped if necessary, and the order in Q_{min} and Q_{max} is restored by decreasing and increasing the swapped elements (if any).

delete(Q, p). If the element to be deleted is the singleton of Q, the singleton is removed and nothing else is done. If Q has a singleton, the element with locator p is removed from its component, the singleton is inserted into that component, and the singleton and the twin of the removed element are made twins of each other. As in *extract* the two twins are swapped and the order in Q_{min} and Q_{max} is restored if necessary. On the other hand, if Q has no singleton, the element with locator p is removed, another element is extracted from the component of its twin, the extracted element is made the new singleton, and, if necessary, the twin of the extracted element and the twin of the removed element are swapped and the order in Q_{min} and Q_{max} is restored as above.

meld(Q, R). Let S denote the outcome of this operation. Without loss of generality, assume that the size of Q is smaller than or equal to that of R. If Q and R together have exactly one singleton, this element becomes the singleton of S. If they have two singletons, these are compared, the non-greater is inserted into Q_{min}, the non-smaller is inserted into Q_{max}, and the inserted elements are made twins of each other. After these preparations, Q_{min} and R_{min} are melded to become S_{min}, and Q_{max} and R_{max} are melded to become S_{max}.

```
find-min(Q): // find-max(Q) is similar
  s ← singleton(Q)
  t ← find-min(Q_min)
  if t = null or
    (s ≠ null and s.element() < t.element())
    return s
  return t
insert(Q, p):
  s ← singleton(Q)
  if s = null
    make-singleton(Q, p)
    return
  if p.element() < s.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, null)
extract(Q):
  s ← singleton(Q)
  if s ≠ null
    make-singleton(Q, null)
    return s
  else
    p ← extract(Q_min)
    if p ≠ null
      q ← extract(Q_max)
      r ← twin(p)
      t ← twin(q)
      make-twins(r, t)
      swap-twins-if-necessary(Q, t)
      make-singleton(Q, q)
    return p
swap-twins-if-necessary(Q, s): // s in Q_min
  t ← twin(s)
  if t.element() < s.element()
    swap(s, t)
    decrease(Q_min, t, t.element())
    increase(Q_max, s, s.element())
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 = Q_min
    swap-twins-if-necessary(Q, s)
  else
    swap-twins-if-necessary(Q, t)
  delete(P, p)
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(Q_min, q)
        insert(Q_max, r)
      else
        insert(Q_min, r)
        insert(Q_max, q)
      make-twins(q, r)
  S_min ← meld(Q_min, R_min)
  S_max ← meld(Q_max, R_max)
  return S
```

Fig. 2 This pseudo-code implements our second data-structural transformation. The subroutines used are assumed to have the following effects: *twin*(p) returns a locator to the twin of the element with locator p; *make-twins*(p, q) assigns the twin pointers between the elements with locators p and q; *singleton*(Q) returns a locator to the singleton of Q; *make-singleton*(Q, p) makes the element with locator p the singleton of Q and sets the twin pointer of p to *null*; *swap*(p, q) puts the element with locator p in place of the element with locator q, and vice versa; and *component*(Q, p) returns the component of Q, in which the element with locator p is stored. One way of implementing *component* is to attach to each node of Q a bit indicating whether that node is in Q_{min} or Q_{max}, and let *insert* update these bits

It is straightforward to verify that the above implementation achieves the bounds of Table 1. By combining the results of Table 1 with the bounds known for the priority queues (Lemma 2), the bounds of Theorem 2 are obtained.

4 Conclusions

We conclude the paper with four 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 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 [12] works on a pointer machine, but the meldable version using the priority queues introduced in [13] relies on the capabilities of a RAM. This is in contrast with Brodal's data structure [5] 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. Alstrup S, Husfeld T, Rauhe T, Thorup M (2005) Black box for constant-time insertion in priority queues. ACM Trans Algorithms 1:102–106
2. Arvind A, Pandu Rangan C (1999) Symmetric min-max heap: a simpler data structure for double-ended priority queue. Inf Process Lett 69:197–199
3. Atkinson MD, Sack J-R, Santoro N, Strothotte T (1986) Min-max heaps and generalized priority queues. Commun ACM 29:996–1000
4. Blum M, Floyd RW, Pratt V, Rivest RL, Tarjan RE (1973) Time bounds for selection. J Comput Syst Sci 7:448–461
5. Brodal GS (1995) Fast meldable priority queues. In: Proceedings of the 4th international workshop on algorithms and data structures. Lecture notes in computer science, vol 955. Springer, Berlin, pp 282–290
6. Brodal GS (1996) Worst-case efficient priority queues. In: Proceedings of the 7th ACM-SIAM symposium on discrete algorithms. ACM/SIAM, New York/Philadelphia, pp 52–58
7. Carlsson S (1987) The deap—a double-ended heap to implement double-ended priority queues. Inf Process Lett 26:33–36
8. Chang SC, Du MW (1993) Diamond deque: a simple data structure for priority deques. Inf Process Lett 46:231–237
9. Cho S, Sahni S (1999) Mergeable double-ended priority queues. Int J Found Comput Sci 10:1–18
10. Chong K-R, Sahni S (2000) Correspondence-based data structures for double-ended priority queues. ACM J Exp Algorithmics 5: article
11. Ding Y, Weiss MA (1993) The relaxed min-max heap: a mergeable double-ended priority queue. Acta Inform 30:215–231

12. Elmasry A, Jensen C, Katajainen J (2008) Multipartite priority queues. ACM Trans Algorithms (to appear)
13. Elmasry A, Jensen C, Katajainen J (2008) Two-tier relaxed heaps. Acta Inform 45:193–210
14. Goodrich MT, Tamassia R (2002) Algorithm design: foundations, analysis, and internet examples. Wiley, Hoboken
15. Hagerup T (1998) Sorting and searching on the word RAM. In: Proceedings of the 15th annual symposium on theoretical aspects of computer science. Lecture notes in computer science, vol 1373. Springer, Berlin, pp 366–398
16. Horowitz E, Sahni S, Rajasekaran S (1998) Computer algorithms/C++. Computer Science Press, New York
17. Høyer P (1995) A general technique for implementation of efficient priority queues. In: Proceedings of the 3rd Israel symposium on the theory of computing and systems, IEEE, Los Alamitos, pp 57–66
18. Katajainen J, Vitale F (2003) Navigation piles with applications to sorting, priority queues, and priority deques. Nordic J Comput 10:238–262
19. Khoong CM, Leong HW (1993) Double-ended binomial queues. In: Proceedings of the 4th international symposium on algorithms and computation. Lecture notes in computer science, vol 762. Springer, Berlin, pp 128–137
20. van Leeuwen J, Wood D (1993) Interval heaps. Comput J 36:209–216
21. Makris C, Tsakalidis A, Tsichlas K (2003) Reflected min-max heaps. Inf Process Lett 86:209–214
22. Mortensen CW, Pettie S (2005) The complexity of implicit and space-efficient priority queues. In: Proceedings of the 9th workshop on algorithms and data structures. Lecture notes in computer science, vol 3608. Springer, Berlin, pp 49–60
23. Nath SK, Chowdhury RA, Kaykobad M (2000) Min-max fine heaps. arXiv.org e-Print archive article cs.DS/0007043. http://arxiv.org
24. Olariu S, Overstreet CM, Wen Z (1991) A mergeable double-ended priority queue. Comput J 34: 423–427
25. Rahman MZ, Chowdhury RA, Kaykobad M (2003) Improvements in double ended priority queues. Int J Comput Math 80:1121–1129