Fast Meldable Priority Queues

Gerth Stølting Brodal

BRICS Report Series RS-95-12

ISSN 0909-0878 February 1995

Fast Meldable Priority Queues

Gerth Stølting Brodal[*]

BRICS[†]
Department of Computer Science, University of Aarhus
Ny Munkegade, DK-8000 Århus C, Denmark

15th February 1995

Abstract

We present priority queues that support the operations **MakeQueue**, **FindMin**, **Insert** and **Meld** in worst case time O(1) and **Delete** and **DeleteMin** in worst case time $O(\log n)$. They can be implemented on the pointer machine and require linear space. The time bounds are optimal for all implementations where **Meld** takes worst case time $o(n)$.

To our knowledge this is the first priority queue implementation that supports **Meld** in worst case constant time and **DeleteMin** in logarithmic time.

[*]This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 7141 (project ALCOM II) and by the Danish Natural Science Research Council (Grant No. 9400044).

[†]**B**asic **R**esearch **i**n **C**omputer **S**cience, Centre of the Danish National Research Foundation.

Introduction

We consider the problem of implementing meldable priority queues. The operations that should be supported are:

MakeQueue	Creates a new empty priority queue.
FindMin(Q)	Returns the minimum element contained in priority queue Q.
Insert(Q, e)	Inserts element e into priority queue Q.
Meld(Q_1, Q_2)	Melds the priority queues Q_1 and Q_2 to one priority queue and returns the new priority queue.
DeleteMin(Q)	Deletes the minimum element of Q and returns the element.
Delete(Q, e)	Deletes element e from priority queue Q provided that it is known where e is stored in Q (priority queues *do not* support the searching for an element).

The implementation of priority queues is a classical problem in data structures. A few references are [13, 12, 8, 7, 5, 6, 10].

In the amortised sense, [11], the best performance is achieved by binomial heaps [12]. They support Delete and DeleteMin in amortised time $O(\log n)$ and all other operations in amortised constant time. If we want to perform Insert in worst case constant time two efficient data structures exist. The implicit priority queues of Carlsson and Munro [2] and the relaxed heaps of Driscoll *et al.* [5], but neither of these support Meld efficiently. However they do support MakeQueue, FindMin and Insert in worst case constant time and Delete and DeleteMin in worst case time $O(\log n)$.

Our implementation beats the above by supporting MakeQueue, FindMin, Insert and Meld in worst case time $O(1)$ and Delete and DeleteMin in worst case time $O(\log n)$. The computational model is the pointer machine and the space requirement is linear in the number of elements contained in the priority queues.

We assume that the priority queues contain elements from a totally ordered universe. The only allowed operation on the elements is the comparisons of two elements. We assume that comparisons can be performed in worst case constant time. For simplicity we assume that all priority queues are nonempty. For a given operation we let n denote the size of the priority queue of maximum size involved in the operation.

In Sect. 1 we describe the data structure and in Sect. 2 we show how to implement the operations. In Sect. 3 we show that our construction is optimal. Section 4 contains some final remarks.

1 The Data Structure

Our basic representation of a priority queue is a heap ordered tree where each node contains one element. This is slightly different from binomial heaps [12] and Fibonacci heaps [8] where the representation is a forest of heap ordered trees.

With each node we associate a rank and we partition the sons of a node into two types, type i and type ii. The heap ordered tree must satisfy the following structural constraints.

a) A node has at most one son of type i. This son may be of arbitrary rank.

b) The sons of type ii of a node of rank r have all rank less than r.

c) For a fixed node or rank r, let n_i denote the number of sons of type ii that have rank i. We maintain the regularity constraint that

$$\begin{array}{rc} i) & \forall i : (0 \leq i < r \Rightarrow 1 \leq n_i \leq 3), \\ ii) & \forall i, j : (i < j \wedge n_i = n_j = 3 \Rightarrow \exists k : i < k < j \wedge n_k = 1), \\ iii) & \forall i : (n_i = 3 \Rightarrow \exists k : k < i \wedge n_k = 1). \end{array}$$

d) The root has rank zero.

The heap order implies that the minimum element is at the root. Properties a), b) and c) bound the degree of a node by three times the rank of the node plus one. The size of the subtree rooted at a node is controlled by property c). Lemma 1 shows that the size is at least exponential in the rank. The last two properties are essential to achieve Meld in worst case constant time. The regularity constraint c) is a variation of the regularity constraint that Guibas *et al.* [9] used in their construction of finger search trees. The idea is that between two ranks where three sons have equal rank there is a rank of which there only is one son. Figure 1 shows a heap ordered tree that satisfies the requirements a) to d) (the elements contained in the tree are omitted).

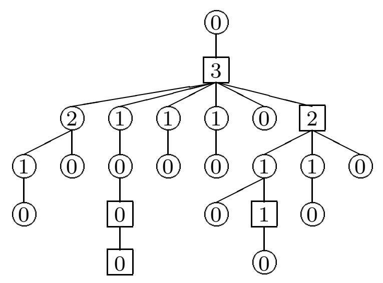

Figure 1: A heap ordered tree satisfying the properties a) to d). A box denotes a son of type i, a circle denotes a son of type ii, and the numbers are the ranks of the nodes.

Lemma 1 *Any subtree rooted at a node of rank r has size $\geq 2^r$.*

Proof: The proof is a simple induction in the structure of the tree. By c.i) leaves have rank zero and the lemma is true. For a node of rank r property c.i) implies that the node has at least one son of each rank less than r. By induction we get that the size is at least $1 + \sum_{i=0}^{r-1} 2^i = 2^r$. □

Corollary 1 *The only son of the root of a tree containing n elements has rank at most $\lfloor \log(n-1) \rfloor$.*

We now describe the details of how to represent a heap ordered tree. A son of type i is always the rightmost son. The sons of type ii appear in increasing rank order from right to left. See Fig. 1 and Fig. 2 for examples.

A node consists of the following seven fields: 1) the element associated with the node, 2) the rank of the node, 3) the type of the node, 4) a pointer to the father node, 5) a pointer to the leftmost son and 6) a pointer to the next sibling to the left. The next sibling pointer of the leftmost son points to the rightmost son in the list. This enables the access to the rightmost son of a node in constant time too. Field 7) is used to maintain a single linked list of triples of sons of type ii that have

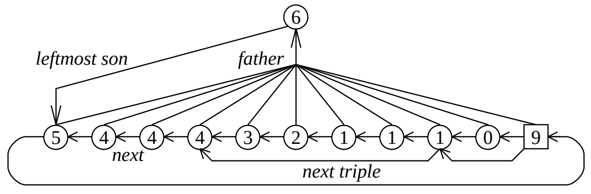

Figure 2: The arrangement of the sons of a node.

equal rank (see Fig. 2). The nodes appear in increasing rank order. We only maintain these pointers for the rightmost son and for the rightmost son in a triple of sons of equal rank. Figure 2 shows an example of how the sons of a node are arranged.

In the next section we describe how to implement the operations. There are two essential transformations. The first transformation is to add a son of rank r to a node of rank r. Because we have a pointer to the leftmost son of a node (that has rank $r-1$ when $r>0$) this can be done in constant time. Notice that this transformation cannot create three sons of equal rank. The second transformation is to find the smallest rank i where three sons have equal rank. Two of the sons are replaced by a son of rank $i+1$. Because we maintain a single linked list of triples of nodes of equal rank we can also do this in constant time.

2 Operations

In this section we describe how to implement the different operations. The basic operation we use is to link two nodes of equal rank r. This is done by comparing the elements associated with the two nodes and making the node with the largest element a son of the other node. By increasing the rank of the node with the smallest element to $r+1$ the properties a) to d) are satisfied. The operation is illustrated in Fig. 3. This is similar to the linking of trees in binomial heaps and Fibonacci heaps [12, 8].

We now describe how to implement the operations.

- MakeQueue is trivial. We just return the null pointer.
- FindMin(Q) returns the element located at the root of the tree

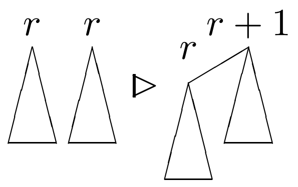

Figure 3: The linking of two nodes of equal rank.

representing Q.

- **Insert**(Q, e) is equal to **Meld** Q with a priority queue only consisting of a rank zero node containing e.
- **Meld**(Q_1, Q_2) can be implemented in two steps. In the first we insert one of the heap ordered trees into the other heap ordered tree. This can violate property c) at one node because the node gets one additional son of rank zero. In the second step we reestablish property c) at the node. Figure 4 shows an example of the first step.

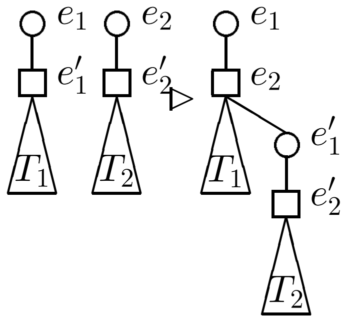

Figure 4: The first step of a **Meld** operation (the case $e_1 \leq e_2 < e'_1 \leq e'_2$).

Let e_1 and e_2 denote the roots of the trees representing Q_1 and Q_2 and let e'_1 and e'_2 denote the only sons of e_1 and e_2. Assume w.l.o.g. that e_1 is the smallest element. If $e_2 \geq e'_1$ we let e_2 become a rank zero son of e'_1, otherwise $e_2 < e'_1$. If $e'_2 < e'_1$ we can interchange the subtrees rooted at e'_2 and e'_1, so w.l.o.g. we assume $e_1 \leq e_2 < e'_1 \leq e'_2$. In this case we make e_2 a rank zero son of e'_1 and swap the elements e'_1 and e_2 (see Fig. 4). We have assumed that the sizes of Q_1 and Q_2 are at least two, but the other cases are just simplified cases of the general case.

The only invariants that can be violated now are the invariants b) and c) at the son of the root because it has got one additional rank zero son. Let v denote the son of the root. If v had rank zero we can satisfy the invariants by setting the rank of v to one. Otherwise only c) can be violated at v. Let n_i denote the number of sons of v that have rank i. By linking two nodes of rank i where i is the smallest rank where $n_i = 3$ it is easy to verify that c) can be reestablished. The linking reduces n_i by two and increments n_{i+1} by one.

If we let $(n_{r-1}, \ldots, n_0)$ be a string in $\{1,2,3\}^*$ the following table shows that c) is reestablished after the above described transformations. We let x denote a string in $\{1,2,3\}^*$ and y_i strings in $\{1,2\}^*$. The table shows all the possible cases. Recall that c) states that between every two $n_i = 3$ there is at least one $n_i = 1$. The different cases are also considered in [9].

$$\begin{aligned}
y_1 1 &\rhd y_1 2 \\
y_2 1 3 y_1 1 &\rhd y_2 2 1 y_1 2 \\
y_2 2 3 y_1 1 &\rhd y_2 3 1 y_1 2 \\
x 3 y_2 1 3 y_1 1 &\rhd x 3 y_2 2 1 y_1 2 \\
x 3 y_3 1 y_2 2 3 y_1 1 &\rhd x 3 y_3 1 y_2 3 1 y_1 2 \\
y_1 1 2 &\rhd y_1 2 1 \\
y_1 2 2 &\rhd y_1 3 1 \\
x 3 y_1 1 2 &\rhd x 3 y_1 2 1 \\
x 3 y_2 1 y_1 2 2 &\rhd x 3 y_2 1 y_1 3 1
\end{aligned}$$

After the linking only b) can be violated at v because a son of rank r has been created. This problem can be solved by increasing the rank of v by one.

Because of the given representation **Meld** can be performed in worst case time O(1).

- **DeleteMin**(Q) removes the root e_1 of the tree representing Q. The problem is that now property d) can be violated because the new root e_2 can have arbitrary rank. This problem is solved by the following transformations.

  First we remove the root e_2. This element later on becomes the new root of rank zero. At most $O(\log n)$ trees can be created by removing the root. Among these trees the root that contains the minimum element e_3 is found and removed. This again creates at most $O(\log n)$ trees. We now find the root (e_4) of maximum rank among all the trees and replaces it by the element e_3. A rank zero node containing e_4 is created.

The tree of maximum rank and with root e_3 is made the only son of e_2. All other trees are made sons of the node containing e_3. Notice that all the new sons of e_3 have rank less than the rank of e_3. By iterated linking of sons of equal rank where there are three sons with equal rank, we can guarantee that $n_i \in \{1, 2\}$ for all i less than the rank of e_3. Possibly, we have to increase the rank of e_3.

Finally, we return the element e_1.

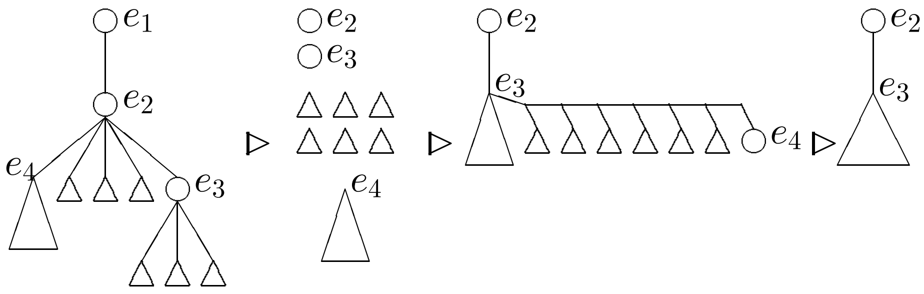

Figure 5: The implementation of DeleteMin.

Because the number of trees is at most $O(\log n)$ DeleteMin can be performed in worst case time $O(\log n)$. Figure 5 illustrates how DeleteMin is performed.

- Delete(Q, e) can be implemented similar to DeleteMin. If e is the root we just perform DeleteMin. Otherwise we start by bubbling e upwards in the tree. We replace e with its father until the father of e has rank less than or equal to the rank of e. Now, e is the arbitrarily ranked son of its father. This allows us to replace e with an arbitrary ranked node, provided that the heap order is still satisfied. Because the rank of e increases for each bubble step, and the rank of a node is bounded by $\lfloor \log(n-1) \rfloor$, this can be performed in time $O(\log n)$.

  We can now replace e with the meld of the sons of e as described in the implementation of DeleteMin. This again can be performed in worst case time $O(\log n)$.

To summarise, we have the theorem:

Theorem 1 *There exists an implementation of priority queues that supports* MakeQueue, FindMin, Insert *and* Meld *in worst case time* $O(1)$ *and* DeleteMin *and* Delete *in worst case time* $O(\log n)$*. The implementation requires linear space and can be implemented on the pointer machine.*

3 Optimality

The following theorem shows that if Meld is required to be nontrivial, i.e. to take worst case sublinear time, then DeleteMin must take worst case logarithmic time. This shows that the construction described in the previous sections is optimal among all implementations where Meld takes sublinear time.

If Meld is allowed to take linear time it is possible to support DeleteMin in worst case constant time by using the finger search trees of Dietz and Raman [3]. By using their data structure MakeQueue, FindMin, DeleteMin, Delete can be supported in worst case time $O(1)$, Insert in worst case time $O(\log n)$ and Meld in worst case time $O(n)$.

Theorem 2 *If* Meld *can be performed in worst case time* $o(n)$ *then* DeleteMin *cannot be performed in worst case time* $o(\log n)$*.*

Proof: The proof is by contradiction. Assume Meld takes worst case time $o(n)$ and DeleteMin takes worst cast time $o(\log n)$. We show that this implies a contradiction with the $\Omega(n \log n)$ lower bound on comparison based sorting.

Assume we have n elements that we want to sort. Assume w.l.o.g. that n is a power of 2, $n = 2^k$. We can sort the elements by the following list of priority queue operations. First, create n priority queues each containing one of the n elements (each creation takes worst case time $O(1)$). Then join the n priority queues to one priority queue by $n-1$ Meld operations. The Meld operations are done bottom-up by always melding two priority queues of smallest size. Finally, perform n DeleteMin operations. The elements are now sorted.

The total time for this sequence of operations is:

$$n\mathrm{T}_{\mathrm{MakeQueue}} + \sum_{i=0}^{k-1} 2^{k-1-i}\mathrm{T}_{\mathrm{Meld}}(2^i) + \sum_{i=1}^{n} \mathrm{T}_{\mathrm{DeleteMin}}(i) = \mathrm{o}(n \log n).$$

This contradicts the lower bound on comparison based sorting. □

4 Conclusion

We have presented an implementation of meldable priority queues where Meld takes worst case time O(1) and DeleteMin worst case time O($\log n$).

Another interesting operation to consider is DecreaseKey. Our data structure supports DecreaseKey in worst case time O($\log n$), because DecreaseKey can be implemented in terms of a Delete operation followed by an Insert operation. Relaxed heaps [5] support DecreaseKey in worst case time O(1) but do not support Meld. But it is easy to see that relaxed heaps can be extended to support Meld in worst case time O($\log n$). The problem to consider is if it is possible to support both DecreaseKey and Meld simultaneously in worst case constant time.

As a simple consequence of our construction we get a new implementation of meldable double ended priority queues, which is a data type that allows both FindMin/FindMax and DeleteMin/DeleteMax [1, 4]. For each queue we just have to maintain two heap ordered trees as described in section 1. One tree ordered with respect to minimum and the other with respect to maximum. If we let both trees contain all elements and the elements know their positions in both trees we get the following corollary.

Corollary 2 *An implementation of meldable double ended priority queues exists that supports* MakeQueue*,* FindMin*,* FindMax*,* Insert *and* Meld *in worst case time* O(1) *and* DeleteMin*,* DeleteMax*,* Delete*,* DecreaseKey *and* IncreaseKey *in worst case time* O($\log n$)*.*

References

[1] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.

[2] Svante Carlsson and J. Ian Munro. An implicit binomial queue with constant insertion time. In *Proc. 1st Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13. Springer Verlag, Berlin, 1988.

[3] Paul F. Dietz and Rajeev Raman. A constant update time finger search tree. In *Advances in Computing and Information - ICCI '90*, volume 468 of *Lecture Notes in Computer Science*, pages 100–109. Springer Verlag, Berlin, 1990.

[4] Yuzheng Ding and Mark Allen Weiss. The relaxed min-max heap. *ACTA Informatica*, 30:215–231, 1993.

[5] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[6] Michael J. Fischer and Michael S. Paterson. Fishspear: A priority queue algorithm. *Journal of the ACM*, 41(1):3–30, 1994.

[7] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self–adjusting heap. *Algorithmica*, 1:111–129, 1986.

[8] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. 25rd Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 338–346, 1984.

[9] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proc. 9thAnn. ACM Symp. on Theory of Computing (STOC)*, pages 49–60, 1977.

[10] Peter Høyer. A general technique for implementation of efficient priority queues. Technical Report IMADA-94-33, Odense University, 1994.

[11] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[12] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[13] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.