

Worst-Case Efficient Priority Queues*

Gerth Stølting Brodal†

Abstract

An implementation of priority queues is presented that supports the operations MAKEQUEUE, FINDMIN, INSERT, MELD and DECREASEKEY in worst case time $O(1)$ and DELETEMIN and DELETE in worst case time $O(\log n)$. The space requirement is linear. The data structure presented is the first achieving this worst case performance.

1 Introduction

We consider the problem of implementing priority queues which are efficient in the worst case sense. The operations we want to support are the following commonly needed priority queue operations [11].

MAKEQUEUE creates and returns an empty priority queue.

FINDMIN(Q) returns the minimum element contained in priority queue Q .

INSERT(Q, e) inserts an element e into priority queue Q .

MELD(Q_1, Q_2) melds priority queues Q_1 and Q_2 to a new priority queue and returns the resulting priority queue.

DECREASEKEY(Q, e, e') replaces element e by e' in priority queue Q provided $e' \leq e$ and it is known where e is stored in Q .

DELETEMIN(Q) deletes and returns the minimum element from priority queue Q .

DELETE(Q, e) deletes element e from priority queue Q provided it is known where e is stored in Q .

The construction of priority queues is a classical topic in data structures [1, 2, 3, 4, 5, 6, 7, 10, 12, 15, 16, 17]. A historical overview of implementations can be found in [11]. There are many applications of priority queues. Two of the most prominent examples are sorting problems and network optimization problems [13].

In the amortized sense, [14], the best performance for these operations is achieved by Fibonacci heaps [7]. They achieve amortized constant time for all operations except for the two delete operations which require amortized time $O(\log n)$. The data structure we present achieves matching worst case time bounds for all operations. Previously, this was only achieved for various strict subsets of the listed operations [1, 2, 3, 15]. For example the relaxed heaps of Driscoll *et al.* [3] and the priority queues in [1] achieve the above time bounds in the worst case sense except that in [3] MELD requires worst case time $\Theta(\log n)$ and in [1] DECREASEKEY requires worst case time $\Theta(\log n)$. Refer to Table 1. If we ignore the DELETE operation our results are optimal in the following sense. A lower bound for DELETEMIN in the comparison model is proved in [1] where it is proved that if MELD can be performed in time $o(n)$ then DELETEMIN *cannot* be performed in time $o(\log n)$.

The data structure presented in this paper originates from the same ideas as the relaxed heaps of Driscoll *et al.* [3]. In [3] the data structure is based on heap ordered trees where $\Theta(\log n)$ nodes may violate heap order. We extend this to allow $\Theta(n)$ heap order violations which is a necessary condition to be able to support MELD in worst case constant time and if we allow a nonconstant number of violations.

In Section 2 we describe the data structure representing a priority queue. In Section 3 we describe a special data structure needed internally in the priority queue construction. In Section 4 we show how to implement the priority queue operations. In Section 5 we summarize the required implementation details. Finally some concluding remarks on our construction are given in Section 6.

2 The Data Structure

In this section we describe the components of the data structure representing a priority queue. A lot of technical constraints are involved in the construction. Primary these are consequences of the transformations to be described in Section 3 and Section 4.3. In Section 5 we summarize the required parts of the construction described in the following sections.

The basic idea is to represent a priority queue

*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).

†BRICS (Basic Research in Computer Science), a Centre of the Danish National Research Foundation at: Computer Science Department, Aarhus University, Ny Munkegade, DK-8000 Århus C, Denmark. Email: gerth@daimi.aau.dk

	Amortized Fredman <i>et al.</i> [7]	Worst case		
		Driscoll <i>et al.</i> [3]	Brodal [1]	New result
MAKEQUEUE	$O(1)$	$O(1)$	$O(1)$	$O(1)$
FINDMIN	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(1)$	$O(1)$	$O(1)$
MELD	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$
DECREASEKEY	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$
DELETE/DELETEMIN	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 1: Time bounds for the previously best priority queue implementations.

by two trees T_1 and T_2 where all nodes contain one element and have a nonnegative integer rank assigned. Intuitively the rank of a node is the logarithm of the size of the subtree rooted at the node. The details of the rank assignment achieving this follow below.

The sons of a node are stored in a doubly linked list in increasing rank order from right to left. Each node has also a pointer to its leftmost son and a pointer to its parent.

The notation we use is the following. We make no distinction between a node and the element it contains. We let x, y, \dots denote nodes, $p(x)$ the parent of x , $r(x)$ the rank of x , $n_i(x)$ the number of sons of rank i that x has and t_i the root of T_i . Nodes which are larger than their parents are called *good* nodes — good because they satisfy heap order. Nodes which are not good are called *violating* nodes.

The idea is to let t_1 be the minimum element and to lazy merge the two trees T_1 and T_2 such that T_2 becomes empty. Since t_1 is the minimum element we can support FINDMIN in worst case constant time and the lazy merging of the two trees corresponds intuitively to performing MELD incrementally over the next sequence of operations. The merging of the two trees is done by incrementally increasing the rank of t_1 by moving the sons of t_2 to t_1 such that T_2 becomes empty and t_1 becomes the node of maximum rank. The actual details of implementing MELD follow in Section 4.5.

As mentioned before we have some restrictions on the trees forcing the rank of a node to be related to the size of the subtree rooted at the node. For this purpose we maintain the invariants S1–S5 below for any node x .

S1 : If x is a leaf, then $r(x) = 0$,

S2 : $r(x) < r(p(x))$,

S3 : if $r(x) > 0$, then $n_{r(x)-1}(x) \geq 2$,

S4 : $n_i(x) \in \{0, 2, 3, \dots, 7\}$,

S5 : $T_2 = \emptyset$ or $r(t_1) \leq r(t_2)$.

The first two invariants just say that leaves have rank zero and that the ranks of the nodes strictly increase towards the root. Invariant S3 says that a node of rank k has at least two sons of rank $k - 1$.

This guarantees that the size of the subtree rooted at a node is at least exponential in the rank of the node (by induction it follows from S1 and S3 that the subtree rooted at node x has size at least $2^{r(x)+1} - 1$). Invariant S4 bounds the number of sons of a node that have the same rank within a constant. This implies the crucial fact that all nodes have rank and degree $O(\log n)$. Finally S5 says that either T_2 is empty or its root has rank larger than or equal to the rank of the root of T_1 .

Notice that in S4 we do not allow a node to have only a single son of a given rank. This is because this allows us to cut off the leftmost sons of a node such that the node can get a new rank assigned where S3 is still satisfied. This property is essential to the transformations to be described in Section 4.3. The requirement $n_i(x) \leq 7$ in S4 is a consequence of the construction described in Section 3.

After having described the conditions of how nodes are assigned ranks and how this forces structure on the trees we now turn to consider how to handle the violating nodes — which together with the two roots could be potential minimum elements. To keep track of the violating nodes we associate to each node x two subsets $V(x)$ and $W(x)$ of nodes larger than x from the trees T_1 and T_2 . That is the nodes in $V(x)$ and $W(x)$ are good with respect to x . We do not require that if $y \in V(x) \cup W(x)$ that x and y belong to the same T_i tree. But we require that a node y belongs to at most one V or one W set. Also we do not require that if $y \in V(x) \cup W(x)$ then $r(y) \leq r(x)$.

The V sets and the W sets are all stored as doubly linked lists. Violations added to a V set are always added to the front of the list. Violations added to a W set are always added in such a way that violations of the same rank are adjacent. So if we have to add a violation to $W(x)$ and there is already a node in $W(x)$ of the same rank, then we insert the new node adjacent to this node. Otherwise we just insert the new node at the front of $W(x)$.

We implement the $V(x)$ and $W(x)$ sets by letting each tree node x have four additional pointers: One to

the first node in $V(x)$, one to the first node in $W(x)$, and two to the next and previous node in the violation list that x belongs to — provided x is contained in a violation list. Each time we add a node to a violation set we always first remove the node from the set it possibly belonged to.

Intuitively $V(x)$'s purpose is to contain violating nodes of large rank. Whereas $W(x)$'s purpose is to contain violating nodes of small rank. If a new violating node is created which has large rank, i.e. $r(x) \geq r(t_1)$, we add the violation to $V(t_1)$, otherwise we add the violation to $W(t_1)$. To be able to add a node to $W(t_1)$ at the correct position we need to know if a node already exists in $W(t_1)$ of the same rank. In case there is we need to know such an element. For this purpose we maintain an extendible array¹ of size $r(t_1)$ of pointers to nodes in $W(t_1)$ of each possible rank. If no node exists of a given rank in $W(t_1)$ the corresponding entry in the array is NULL.

The structure on the V and W sets is enforced by the following invariants O1–O5. We let $w_i(x)$ denote the number of nodes in $W(x)$ of rank i .

$$\text{O1 : } t_1 = \min T_1 \cup T_2,$$

$$\text{O2 : if } y \in V(x) \cup W(x), \text{ then } y \geq x,$$

$$\text{O3 : if } y < p(y), \text{ then an } x \neq y \text{ exists such that } y \in V(x) \cup W(x),$$

$$\text{O4 : } w_i(x) \leq 6,$$

$$\text{O5 : if } V(x) = (y_{|V(x)|}, \dots, y_2, y_1), \text{ then}$$

$$r(y_i) \geq \lfloor (i-1)/\alpha \rfloor \text{ for } i = 1, 2, \dots, |V(x)|$$

where α is a constant.

O1 guarantees that the minimum element contained in a priority queue always is the root of T_1 . O2 says that the elements are heap ordered with respect to membership of the V and W sets. O3 says that all violating nodes belong to a V or W set. Because all nodes have rank $O(\log n)$ invariants O4 and O5 imply that the sizes of all V and W sets are $O(\log n)$. Notice that if we remove an element from a V or W set, then the invariants O4 and O5 cannot become violated.

That invariants O4 and O5 are stated quite differently is because the V and W sets have very different roles in the construction. Recall that the V sets take care of large violations, i.e. violations that have rank larger than $r(t_1)$ when they are created. The constant

¹An extendible array is an array of which the length can be increased by one in worst case constant time. It is folklore that extendible arrays can be obtained from ordinary arrays by array doubling and incremental copying. In the rest of this paper all arrays are extendible arrays.

α is the number of large violations that can be created between two increases in the rank of t_1 .

For the roots t_1 and t_2 we strengthen the invariants such that R1–R3 also should be satisfied.

$$\text{R1 : } n_i(t_j) \in \{2, 3, \dots, 7\} \text{ for } i = 0, 1, \dots, r(t_j) - 1,$$

$$\text{R2 : } |V(t_1)| \leq \alpha r(t_1),$$

$$\text{R3 : if } y \in W(t_1) \text{ then } r(y) < r(t_1).$$

Invariant R1 guarantees that there are at least two sons of each rank at both roots. This property is important for the transformations to be described in Section 4.2 and Section 4.3. Invariant R2 together with invariant O5 guarantee that if we can increase the rank of t_1 by one we can create α new large violations and add them to $V(t_1)$ without violating invariant O5. Invariant R3 says that all violations in $W(t_1)$ have to be small.

The maintenance of R1 and O4 turns out to be nontrivial but they can all be maintained by applying the same idea. To unify this idea we introduce the concept of a *guide* to be described in Section 3.

The main idea behind the construction is the following captured by the DECREASEKEY operation. The details follow in Section 4. Each time we perform a DECREASEKEY operation we just add the new violating node to one of the sets $V(t_1)$ or $W(t_1)$. To avoid having too many violations stored at the root of T_1 we incrementally do two different kinds of transformations. The first transformation moves the sons of t_2 to t_1 such that the rank of t_1 increases. The second transformation reduces the number of violations in $W(t_1)$ by replacing two violations of rank k by at most one violation of rank $k+1$. These transformations are performed to reestablish invariants R2 and O4.

3 Guides

In this section we describe the *guide* data structure that helps us maintaining the invariants R1 and O4 on $n_i(t_1), n_i(t_2)$ and $w_i(t_1)$. The relationship between the abstract sequences of variables in this section and the sons and the violations stored at the roots are explained in Section 4.

The problem can informally be described as follows. Assume we have to maintain a sequence of integer variables x_k, x_{k-1}, \dots, x_1 (all sequences in this section goes from right to left) and we want to satisfy the invariant that all $x_i \leq T$ for some threshold T . On the sequence we can only perform REDUCE(i) operations which decrease x_i by at least two and increase x_{i+1} by at most one. The x_i s can be forced to increase and decrease by one, but for each change in an x_i we are allowed to do $O(1)$ REDUCE operations to prevent any x_i from exceeding T . The guide's job is to tell us which operations to perform.

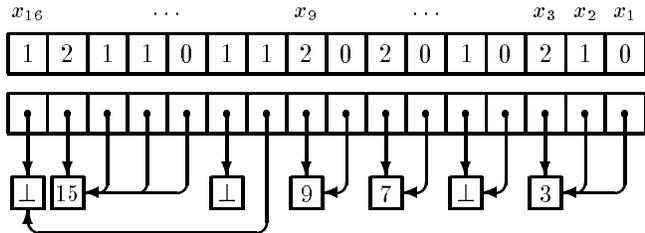


Figure 1: The guide data structure.

This problem also arises implicitly in [1, 2, 8, 9]. But the solution presented in [8] requires time $\Theta(k)$ to find which REDUCE operations to perform whereas the problems in the other papers are simpler because only x_1 can be forced to increase and decrease. The data structure we present can find which operations to perform in worst case time $O(1)$ for the general problem.

To make the guide's knowledge about the x_i s as small as possible we reveal to the guide another sequence x'_k, \dots, x'_1 such that $x_i \leq x'_i \in \{T-2, T-1, T\}$ (this choice is a consequence of the construction we describe below). As long as all $x_i \leq x'_i$ we do not require help from the guide. First when an $x_i = x'_i$ is forced to become $x_i + 1$ we require help from the guide. In the following we assume w.l.o.g. that the threshold T is two such that $x'_i \in \{0, 1, 2\}$ and that $\text{REDUCE}(i)$ decreases x'_i by two and increases x'_{i+1} by one.

The data structure maintained by the guide partitions the sequence x'_k, \dots, x'_1 into *blocks* of consecutive x'_i s of the form $2, 1, 1, \dots, 1, 0$ where the number of ones is allowed to be zero. The guide maintains the invariant that all x'_i s not belonging to a block of the above type have value either zero or one. An example of a sequence satisfying this is the following where blocks are shown by underlining the subsequences.

$$1, \underline{2, 1, 1, 0}, 1, 1, \underline{2, 0}, \underline{2, 0}, 1, 0, \underline{2, 1, 0}.$$

The guide stores the values of the variables x'_i in one array and uses another array to handle the blocks. The second array contains pointers to memory cells which contain the index of an x_i or the value $-$. All variables in the same block point to the same cell and this cell contains the index of the leftmost variable in the block. Variables not belonging to a block point to a cell containing $-$. A data structure for the previous example is illustrated in Figure 1. Notice that several variables can share a memory cell containing $-$. This data structure has two very important properties:

1. Given a variable we can in worst case time $O(1)$ find the leftmost variable in the block, and
2. we can in worst case time $O(1)$ destroy a given

block, i.e. let all nodes in the block belong to no block, by simply assigning $-$ to the block's memory cell.

When an x'_i is forced to increase the guide can in worst case time $O(1)$ decide which REDUCE operations to perform. We only show how to handle one nontrivial case, all other cases are similar. Assume that there are two blocks of variables adjacent to each other and that the leftmost $x'_i = 1$ in the rightmost block has to be increased. Then the following transformations have to be performed:

$$\begin{array}{ll} \underline{2, 1, 1, 0}, \underline{2, 1, 1, 1, 0} & \\ \triangleright \underline{2, 1, 1, 0}, \underline{2, 2, 1, 1, 0} & \text{increment } x'_i, \\ \triangleright \underline{2, 1, 1, 1, 0}, \underline{2, 1, 1, 0} & \text{REDUCE,} \\ \triangleright \underline{2, 1, 1, 1, 1, 0}, \underline{1, 0, 1, 1, 0} & \text{REDUCE,} \\ \triangleright \underline{2, 1, 1, 1, 1, 0}, 1, 1, 0 & \text{reestablish blocks.} \end{array}$$

To reestablish the blocks the two pointers of the new variables in the leftmost block are set to point to the leftmost block's memory cell and the rightmost block's memory cell is assigned the value $-$.

In the case described above only two REDUCE operations were required and these were performed on x'_j s where $j \geq i$. This is true for all cases.

We conclude this section with two remarks on the construction. By using extendible arrays the sequence of variables can be extended by a new x_{k+1} equal to zero or one in worst case time $O(1)$. If we add a reference counter to each memory cell we can reuse the memory cells such that the total number of needed memory cells is at most k .

4 Operations

In this section we describe how to implement the different priority queue operations. We begin by describing some transformations on the trees which are essential to the operations to be implemented.

4.1 Linking and delinking trees. The fundamental operation on the trees is the linking of trees. Assume that we have three nodes x_1, x_2 and x_3 of equal rank and none of them is a root t_i . By doing two comparisons we can find the minimum. Assume w.l.o.g. that x_1 is the minimum. We can now make x_2 and x_3 the leftmost sons of x_1 and increase the rank of x_1 by one. Neither x_2 or x_3 become violating nodes and x_1 still satisfies all the invariants S1-S5 and O1-O5.

The delinking of a tree rooted at node x is a little bit more tricky. If x has exactly two or three sons of rank $r(x) - 1$, then these two or three sons can be cut off and x gets the rank of the largest ranked son plus one. From S4 it follows that x still satisfies S3 and it

follows that S1–S5 and O1–O5 are still satisfied. In the case where x has at least four sons of rank $r(x) - 1$ two of these sons are simply cut off. Because x still has at least two sons of rank $r(x) - 1$ the invariants are still satisfied.

It follows that the delinking of a tree of rank k always results in two or three trees of rank $k - 1$ and one additional tree of rank at most k (the tree can be of any rank between zero and k).

4.2 Maintaining the sons of a root. We now describe how to add sons below a root and how to cut off sons at a root while keeping R1 satisfied. For this purpose we require four guides, two for each of the roots t_1 and t_2 . We only sketch the situation at t_1 because the construction for t_2 is analogous.

To have constant time access to the sons of t_1 we maintain an extendible array of pointers that for each rank $i = 0, \dots, r(t_1) - 1$ has a pointer to a son of t_1 of rank i . Because of R1 such sons are guaranteed to exist. This enables us to link and delink sons of rank i in worst case time $O(1)$ for an arbitrary i . One guide takes care of that $n_i(t_1) \leq 7$ and the other of that $n_i(t_1) \geq 2$ for $i = 0, \dots, r(t_1) - 3$ (to maintain a lower bound on a sequence of variables is equivalent to maintaining an upper bound on the negated sequence). The sons of t_1 of rank $r(t_1) - 1$ and $r(t_1) - 2$ are treated separately in a straight forward way such that there always are between 2 and 7 sons of these ranks. This is necessary because of the dependency between the guide maintaining the upper bound on $n_i(t_1)$ and the guide maintaining the lower bound on $n_i(t_1)$. The “marked” variables that we reveal to the guide that maintains the upper bound on $n_i(t_1)$ have values $\{5, 6, 7\}$ and to the guide that maintains the lower bound have values $\{4, 3, 2\}$.

If we add a new son at t_1 of rank i we tell the guide maintaining the upper bound that $n_i(t_1)$ is forced to increase by one (this assumes $i < r(t_1) - 2$). Then the guide then tells us where to do at most two REDUCE operations. The REDUCE(i) operation in this context corresponds to the linking of three trees of rank i . This decreases $n_i(t_1)$ by three and increases $n_{i+1}(t_1)$ by one. We only do the linking when $n_i(t_1) = 7$ so that the guide maintaining the lower bound on $n_i(t_1)$ will be unaffected (this implies a minor change in the guide). If this results in too many sons of rank $r(t_1) - 2$ or $r(t_1) - 1$ we link some of these sons and possibly increase the rank of t_1 . If the rank of t_1 increases we also have to increase the domain of the two guides.

To cut off a son is similar, but now the REDUCE operation corresponds to the delinking of a tree. The additional tree from the delinking transformation that can have various ranks is treated separately after the

delinking. We just add it below t_1 as described above.

At t_2 the situation is nearly the same. The major difference is that because we knew that t_1 was the smallest element the linking and delinking of sons of t_1 would not create new violations. This is not true at t_2 . The linking of sons never creates new violations but the delinking of sons at t_2 can create three new violations. We will see in Section 4.4 that it turns out that we only cut off sons of t_2 which have rank larger than $r(t_1)$. The tree “left over” by a delinking is made a son of t_1 if it has rank less than $r(t_1)$. Otherwise it is made a son of t_2 . The new violations which have rank larger than $r(t_1)$ are added to $V(t_1)$. To satisfy O5 and R2 we just have to guarantee that the rank of t_1 will be increased and that α in R2 and O5 is chosen large enough.

4.3 Violation reducing transformations. We now describe the most essential transformation on the trees. The transformation reduces the number of *potential violations* $\bigcup_{y \in T_1 \cup T_2} V(y) \cup W(y)$ in the tree by at least one.

Assume we have two potential violations x_1 and x_2 of equal rank $k < r(t_1)$ which are not roots or sons of a root. First we check that both x_1 and x_2 are violating nodes. If one of the nodes already is a good node we remove it from the corresponding violation set. Otherwise we proceed as described below.

Because of S4 we know that both x_1 and x_2 have at least one brother. If x_1 and x_2 are not brothers assume w.l.o.g. that $p(x_1) \leq p(x_2)$ and swap the subtrees rooted at x_1 and at a brother of x_2 . The number of violations can only decrease by doing this swap. We can now w.l.o.g. assume that x_1 and x_2 are brothers and both sons of node y .

If x_1 has more than one brother of rank k we just cut off x_1 and make it a good son of t_1 as described in Section 4.2. Because x_1 had at least two brothers of rank k , S4 is still satisfied at y .

In case x_1 and x_2 are the only brothers of rank k and $r(y) > k + 1$ we just cut off both x_1 and x_2 and make them new good sons of t_1 as described in Section 4.2. Because of invariant S4 we are forced to cut off both sons.

The only case that remains to be considered is when x_1 and x_2 are the only sons of rank k and that $r(y) = k + 1$. In this case we cut off x_1 , x_2 and y . The new rank of y is uniquely given by one plus the rank of its new leftmost son. We replace y by a son of t_1 of rank $k + 1$ which can be cut off as described in Section 4.2. If y was a son of t_1 we only cut off y . If the replacement for y becomes a violating node of rank $k + 1$ we add it to $W(t_1)$. Finally, x_1 , x_2 and y are made good sons of t_1 as described in Section 4.2.

Above it is important that the node y is replaced by is not an ancestor of y , because if y was replaced by such a node a cycle among the parent pointers would be created. Invariant S2 guarantees that this cannot happen.

4.4 Avoiding too many violations. We now describe how to avoid too many violations. The only violation sets we add violations to are $V(t_1)$ and $W(t_1)$. Violations of rank larger than $r(t_1)$ are added to $V(t_1)$ and otherwise they are added to $W(t_1)$. The violations in $W(t_1)$ are controlled by a guide. This guide guarantees that $w_i(t_1) \leq 6$. We maintained a single array so we could access the violating nodes in $W(t_1)$ by their rank.

If we add a violation to $W(t_1)$ the guide tells us for which ranks we should do violation reducing transformations as described in the previous section. We only do the transformation if there are exactly six violations of the given rank and that there is at least two violating nodes which are not sons of t_2 . If there are more than four violations that are sons of t_2 we cut the additional violations off and links them below t_1 . This makes these nodes good and does not affect the guides maintaining the sons at t_2 .

For each priority queue operation that is performed we increase the rank of t_1 by at least one by moving a constant number of sons from t_2 to t_1 — provided $T_2 \neq \emptyset$. By increasing the rank of t_1 by one we can afford creating α new violations of rank larger than $r(t_1)$ by invariant O5 and we can just add the violations to the list $V(t_1)$. If $T_2 \neq \emptyset$ and $r(t_2) \leq r(t_1) + 2$ we just cut off the largest sons of t_2 and link them below t_1 and finally add t_2 below t_1 . This will satisfy the invariants. Otherwise we cut off a son of t_2 of rank $r(t_1) + 2$ and delink this son and add the resulting trees below t_1 such that the rank of t_1 increases by at least one. By choosing α large enough the invariants will become reestablished.

If T_2 is empty we cannot increase the rank t_1 , but this also implies that t_1 is the node of maximum rank so no large violations can be created and R2 cannot become violated.

4.5 Priority queue operations. In the following we describe how to implement the different priority queue operations such that the invariants from Section 2 are maintained.

- MAKEQUEUE is trivial. We return a pair of empty trees.
- FINDMIN(Q) returns t_1 .
- INSERT(Q, e) is a special case of MELD where Q_2 is a priority queue only containing one element.

- MELD(Q_1, Q_2) involves at most four trees; two for each queue. The tree having the new minimum element as root becomes the new T_1 tree. This tree was either the T_1 tree of Q_1 or of Q_2 . If this tree is the tree of maximum rank we just add the other trees below this tree as described previously. In this case no violating node is created so no transformation is done on the violating nodes.

Otherwise the tree of maximum rank becomes the new T_2 tree and the remaining trees are added below this node as described in Section 4.2, possibly delinking the new sons once if they have the same rank as t_2 . The violations created by this are treated as described in Section 4.4. The guides and arrays used at the old roots that now are linked below the new t_2 node we just discard.

- DECREASEKEY(Q, e, e') replaces the element of e by e' ($e' \leq e$). If e' is less than t_1 we swap the elements in the two nodes. If e' is a good node we stop, otherwise we proceed as described in Section 4.4 to avoid having too many violations stored at t_1 .
- DELETEMIN(Q) is allowed to take worst case time $O(\log n)$. First T_2 is made empty by moving all sons of T_2 to T_1 and making the root t_2 a rank zero son of t_1 . Then t_1 is deleted. This gives us at most $O(\log n)$ independent trees. The minimum element is then found by looking at the sets V and W of the old root of T_1 and all the roots of the independent trees. If the minimum element is not a root we swap it with one of the independent trees of equal rank. This at most creates one new violation. By making the independent trees sons of the new minimum element and performing $O(\log n)$ linking and delinking operations on these sons we can reestablish S1–S5 and R1 and R3. By merging the V and W sets at the root to one set and merging the old minimum element's V and W sets with the set we get one new set of violations of size $O(\log n)$. Possibly we also have to add the single violation created by the swapping. By doing at most $O(\log n)$ violation reducing transformations as described previously we can reduce the set to contain at most one violation of each rank. We make the resulting set the new W set of the new root and let the corresponding V set be empty. This implies that O1–O5 and R2 are being reestablished. The guides involved are initiated according to the new situation at the root of T_1 .
- DELETE(Q, e). If we let $-\infty$ denote the smallest possible element, then DELETE can be implemented as DECREASEKEY($Q, e, -\infty$) followed by DELETEMIN(Q).

5 Implementation details

In this section we summarize the required details of our new data structure.

Each node we represent by a record having the following fields.

- The element associated with the node,
- the rank of the node,
- pointers to the node's left and right brothers,
- a pointer to the father node,
- a pointer to the leftmost son,
- pointers to the first node in the node's V and W sets, and
- pointers to the next and the previous node in the violation list that the node belongs to. The first node in a violation list $V(x)$ or $W(x)$ has its previous violation pointer pointing to x .

In addition to the above nodes we maintain the following three extendible arrays:

- An array of pointers to sons of t_1 of rank $i = 0, \dots, r(t_1) - 1$,
- a similar array for t_2 , and
- an array of pointers to nodes in $W(t_1)$ of rank $i = 0, \dots, r(t_1) - 1$ (if no node in $W(t_1)$ exist of a given rank we let the corresponding pointer be NULL).

Finally we have five guides: Three to maintain the upper bounds on $n_i(t_1), n_i(t_2)$ and $w_i(t_1)$ and two to maintain the lower bounds on $n_i(t_1)$ and $n_i(t_2)$.

6 Conclusion

From the construction presented in the previous sections we conclude that:

THEOREM 6.1. *An implementation of priority queues exists that supports the operations MAKEQUEUE, FINDMIN, INSERT, MELD and DECREASEKEY in worst case time $O(1)$ and DELETEMIN and DELETE in worst case time $O(\log n)$. The space required is linear in the number of elements contained in the priority queues.*

The data structure presented is quite complicated. An important issue for further work is to simplify the construction to make it applicable in practice. It would also be interesting to see if it is possible to remove the requirement for arrays from the construction.

Acknowledgement

The author thanks Rolf Fagerberg for the long discussions that lead to the results presented in the paper.

References

- [1] Gerth Stølting Brodal. Fast meldable priority queues. In *Proc. 4th Workshop on Algorithms and Data Structures (WADS)*, volume 955 of *Lecture Notes in Computer Science*, pages 282–290. Springer Verlag, Berlin, 1995.
- [2] Svante Carlsson, Patricio V. Poblete, 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] 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.
- [4] Rolf Fagerberg. A generalization of binomial queues. Technical Report IMADA-94-35, Odense University, 1994. To appear in *Information Processing Letters*.
- [5] Robert W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.
- [6] 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.
- [7] 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.
- [8] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 49–60, 1977.
- [9] Haim Kaplan and Robert Tarjan. Persistent lists with catenation via recursive slow-down. In *Proc. 27th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 93–102, 1995.
- [10] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.
- [11] Kurt Mehlhorn and Athanasios K. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. MIT Press/Elsevier, 1990.
- [12] Jörg-R. Sack and Thomas Strothotte. An algorithm for merging heaps. *ACTA Informatica*, 22:171–186, 1985.
- [13] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.
- [14] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.
- [15] Jan van Leeuwen. The composition of fast priority queues. Technical Report RUU-CS-78-5, Department of Computer Science, University of Utrecht, 1978.
- [16] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.
- [17] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.