Worst-Case Optimal Priority Queues via Extended Regular Counters*

Amr Elmasry and Jyrki Katajainen

Department of Computer Science, University of Copenhagen, Denmark

Abstract. We consider the classical problem of representing a collection of priority queues under the operations *find-min*, *insert*, *decrease*, *meld*, *delete*, and *delete-min*. In the comparison-based model, if the first four operations are to be supported in constant time, the last two operations must take at least logarithmic time. Brodal showed that his worst-case efficient priority queues achieve these worst-case bounds. Unfortunately, this data structure is involved and the time bounds hide large constants. We describe a new variant of the worst-case efficient priority queues that relies on extended regular counters and provides the same asymptotic time and space bounds as the original. Due to the conceptual separation of the operations on regular counters and all other operations, our data structure is simpler and easier to describe and understand. Also, the constants in the time and space bounds are smaller.

1 Introduction

A priority queue is a fundamental data structure that maintains a set of elements and supports the operations *find-min*, *insert*, *decrease*, *delete*, *delete-min*, and *meld*. In the comparison-based model, from the $\Omega(n \lg n)$ lower bound for sorting it follows that, if *insert* can be performed in $o(\lg n)$ time, *delete-min* must take $\Omega(\lg n)$ time. Also, if *meld* can be performed in $o(n)$ time, *delete-min* must take $\Omega(\lg n)$ time [1]. In addition, if *find-min* can be performed in constant time, *delete* would not be asymptotically faster than *delete-min*. Based on these observations, a priority queue is said to provide *optimal time bounds* if it can support *find-min*, *insert*, *decrease*, and *meld* in constant time; and *delete* and *delete-min* in $O(\lg n)$ time, where n denotes the number of elements stored.

After the introduction of binary heaps [14], which do not provide optimal time bounds for all priority-queue operations, an important turning point was when Fredman and Tarjan introduced Fibonacci heaps [10]. Fibonacci heaps provide optimal time bounds for all standard operations in the amortized sense. Driscoll et al. [4] introduced run-relaxed heaps, which have optimal time bounds for all operations in the worst case, except for *meld*. On the other side, Brodal [1] introduced meldable priority queues, which provide the optimal worst-case time bounds for all operations, except for *decrease*. Later, by introducing several innovative ideas, Brodal [2] was able to achieve the worst-case optimal time bounds for all operations. Though deep and involved, Brodal's data structure is complicated and should be taken as a proof of existence.

Most priority queues supporting worst-case constant *decrease* rely on the concept of *violation reductions*. A *violation* is a node that may, but not necessarily, violate the heap order by being smaller than its parent. When the value of a node is decreased, the node is declared violating. To reduce the number of violations, a violation reduction is performed in constant time whenever possible.

A *numeral system* is a notation for representing numbers in a consistent manner using symbols (*digits*). Operations on these numbers, as increments and decrements of a given digit, must obey the rules governing the numeral system. There is a connection between numeral systems and data-structural design [3, 13]. The idea is to relate the number of objects of a specific type in the data structure to the value of a digit. A representation of a number that is subject to increments and decrements of arbitrary digits can be called a *counter*. For an integer $b \geq 2$, a *regular b-ary counter* [3] uses the digits $\{0, \ldots, b\}$ in its representation and imposes the rule that between any two b's there must be a digit other than $b-1$. Such a counter supports increments of arbitrary digits with a constant number of digit changes per operation. An *extended regular b-ary counter* [3, 11] uses the digits $\{0, \ldots, b+1\}$ with the constraints that between any two $(b+1)$'s there is a digit other than b, and between any two 0's there is a digit other than 1. Such a counter supports both increments and decrements of arbitrary digits with a constant number of digit changes per operation.

Kaplan and Tarjan [12] (see also [11]) introduced fat heaps as a simplification of Brodal's worst-case optimal priority queues, but these are not meldable in $O(1)$ time. In fat heaps an extended regular ternary counter is used to maintain the trees and an extended regular binary counter to maintain the violation nodes. In a recent study [9], we have simplified fat heaps to work without regular counters.

Our motives for writing the current paper were the following.

1. We simplify Brodal's construction and devise a priority queue that provides optimal worst-case bounds for all operations (§2, §3, and §4). Throughout our explanation of the data structure, contrary to [2], we distinguish between the numeral-system operations and other priority-queue operations. The gap between the description complexity of worst-case optimal priority queues and binary heaps [14] is huge. One of our motivations was to narrow this gap.
2. A key ingredient in our construction is the utilization of extended regular binary counters. We describe a strikingly simple implementation of the extended regular counters (§5). In spite of their importance for many applications, the existing descriptions [3, 11] are sketchy and incomplete.
3. In this paper we are mainly interested in the theoretical performance of the structures discussed. However, some of our ideas may be of practical value.
4. With this paper, we complete our research program on the comparison complexity of priority-queue operations. All the obtained results are summarized in Table 1. Elsewhere, it has been documented that, in practice, worst-case efficient priority queues are often outperformed by simpler non-optimal priority queues. Due to the involved constant factors in the number of element comparisons, this is particularly true if one aims at developing priority queues that achieve optimal time bounds for all the standard operations.

Table 1. The best-known worst-case comparison complexity of different priority-queue operations. A minus sign indicates that the operation is not supported optimally.

Data structure	*find-min*	*insert*	*decrease*	*meld*	*delete-min*
Multipartite priority queues [5]	$O(1)$	$O(1)$	–	–	$\lg n + O(1)$
Two-tier relaxed heaps [6]	$O(1)$	$O(1)$	$O(1)$	–	$\lg n + O(\lg \lg n)$
Meldable priority queues [7]	$O(1)$	$O(1)$	–	$O(1)$	$2 \lg n + O(1)$
Optimal priority queues [this paper, 8]	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$\approx 70 \lg n$

2 Description of the Data Structure

Let us begin with a high-level description of the data structure. The overall construction is similar to that used in [2]. However, the use of extended regular counters is new. In accordance, the rank rules and hence the tree structures are different from [2]. The set of violation-reduction routines are in turn new.

Each priority queue is composed of two multi-way trees T_1 and T_2, with roots t_1 and t_2 respectively (T_2 can be empty). The atomic components of the priority queues are nodes, each storing a single element. The *rank* of a node x, denoted $rank(x)$, is an integer that is logarithmic in the size (number of nodes) of the subtree rooted at x. The children of a node are stored in a doubly-linked list in non-decreasing rank order. The rank of a subtree is the rank of its root. The trees T_1 and T_2 are heap ordered, except for some violations; but, the element at t_1 is the minimum among the elements of the priority queue. If t_2 exists, the rank of t_1 is less than that of t_2, i.e. $rank(t_1) < rank(t_2)$.

Similar to [2], in the whole data structure there can be up to $O(n)$ violations, not $O(\lg n)$ violations as in run-relaxed heaps and fat heaps. Each violation is guarded by a node that is smaller in value. Any node x, other than t_1, only guards a single list of $O(\lg n)$ violations; these are violations that took place while x was storing the minimum of its priority queue. Such violations must be tackled once the element associated with x is deleted.

For the violations guarded by t_1, a *violation structure* is maintained with each priority queue [2, 4, 11]. This structure is composed of a resizable array, called the *violation array*, in which the rth entry refers to violations of rank r, and a doubly-linked list, which links the entries of the array that have more than two violations each. The violations guarded by t_1 are divided into two groups: the so-called *active violations* are used to perform violation reductions, and the so-called *inactive violations* are violations whose ranks were larger than the size of the violation array at the time when the violation occurred. All the active violations guarded by t_1, besides being kept in a doubly-linked list, are also kept in the violation structure; all inactive violations are kept in a doubly-linked list. While violations are declared, there are two phases for handling them in accordance to whether the size of the violation array is as big as the largest rank or not. During the first phase the following actions are taken. 1) The new violation is added to the list of inactive violations. 2) The violation array is extended by a constant number of entries. During the second phase the following actions are

taken. 1) The new violation is added to the list of active violations and to the violation structure. 2) A violation reduction is performed if possible.

Handling the violations can be realized by letting each node have one pointer to its violation list, and two pointers to its predecessor and successor in the violation list where the node itself may be in. By maintaining all active violations of the same rank consecutively in the violation list of t_1, the violation array can just have a pointer to the first active violation of any particular rank.

In addition to an element, each node stores its rank and six pointers pointing to: the left sibling, the right sibling (the parent if no right sibling exists), the last child (the rightmost child), the head of the guarded violation list, and the predecessor and the successor in the violation list where the node may be in. To decide whether the right-sibling pointer of a node x points to a sibling or a parent, we locate the node y pointed to by the right-sibling pointer of x and check if the last-child pointer of y points back to x.

Next, we state the rank rules controlling the structure of the multi-way trees:

(a) The rank of a node is one more than the rank of its last child. The rank of a node that has no children is 0.

(b) The *rank sequence* of a node specifies the multiplicities of the ranks of its children. If the rank sequence has a digit d_r, the node has d_r children of rank r. The rank sequences of t_1 and t_2 are maintained in a way that allows adding and removing an arbitrary subtree of a given rank in constant time. This is done by having the rank sequences of those nodes obey the rules of an extended regular binary counter; see §5. When we add a subtree or remove a subtree from *below* t_1 or t_2, we also do the necessary actions to reestablish the constraints imposed by the numeral system.

(c) Consider a node x that is not t_1, t_2, or a child of t_1 or t_2. If the rank of x is r, there must exist at least one sibling of x whose rank is $r-1$, r or $r+1$. Note that this is a relaxation to the rules applied to the rank sequences of t_1 and t_2, for which the same rule also applies [3]. In addition, the number of siblings having the same rank is upper bounded by at most three.

Among the children of a node, there are consecutive subsequences of nodes with consecutive, and possibly equal, ranks. We call each maximal subsequence of such nodes a *group*. By our rank rules, a group has at least two *members*. The difference between the rank of a member of a group and that of another group is at least two, otherwise both constitute the same group.

Lemma 1. *The rank and the number of children of any node in our data structure is* $O(\lg n)$*, where* n *is the size of the priority queue.*

Proof. We prove by induction that the size of a subtree of rank r is at least F_r, where F_r is the rth Fibonacci number. The claim is clearly true for $r \in \{0, 1\}$. Consider a node x of rank $r \geq 2$, and assume that the claim holds for all values smaller than r. The last child of x has rank $r-1$. Our rank rules imply that there is another child of rank at least $r-2$. Using the induction hypothesis, the size of these two subtrees is at least F_{r-1} and F_{r-2}. Then, the size of the subtree

rooted at x is at least $F_{r-1} + F_{r-2} = F_r$. Hence, the maximum rank of a node is at most $1.44 \lg n$. By the rank rules, every node has at most three children of the same rank. It follows that the number of children per node is $O(\lg n)$. □

Two trees of rank r can be *joined* by making the tree whose root has the larger value the last subtree of the other. The rank of the resulting tree is $r+1$. Alternatively, a tree rooted at a node x of rank $r+1$ can be *split* by detaching its last subtree. If the last group among the children of x now has one member, the subtree rooted at this member is also detached. The rank of x becomes one more than the rank of its current last child. In accordance, two or three trees result from a *split*; among them, one has rank r and another has rank $r-1$, r, or $r+1$. The *join* and *split* operations are used to maintain the constraints imposed by the numeral system. Note that one element comparison is performed with the *join* operation, while the *split* operation involves no element comparisons.

3 Priority-Queue Operations

One complication, also encountered in [2], is that not all violations can be recorded in the violation structure. The reason is that, after a *meld*, the violation array may be too small when the old t_1 with the smaller rank becomes the new t_1 of the melded priority queue. Assume that we have a violation array of size s associated with t_1. The priority queue may contain nodes of rank $r \geq s$. Hence, violations of rank r cannot be immediately recorded in the array and remain inactive. Violation reductions are only performed on active violations whenever possible. Throughout the lifetime of t_1, the array is incrementally extended by the upcoming priority-queue operations until its size reaches the largest rank. Once the array is large enough, no new inactive violations are created. Since each priority-queue operation can only create a constant number of violations, the number of inactive violations is $O(\lg n)$.

In connection with every *decrease* or *meld*, if T_2 exists, a constant number of subtrees rooted at the children of t_2 are removed from below t_2 and added below t_1. Once $\mathit{rank}(t_1) \geq \mathit{rank}(t_2)$, the whole tree T_2 is added below t_1. To be able to move all subtrees from below t_2 and finish the job on time, we should always pick a subtree from below t_2 whose rank equals the current rank of t_1.

The priority queue operations aim at maintaining the following invariants:

1. The minimum is associated with t_1.
2. The second-smallest element is either stored at t_2, at one of the children of t_1, or at one of the violation nodes associated with t_1.
3. The number of entries in the violation list of a node is $O(\lg n)$, assuming that the priority queue that contains this node has n elements.

We are now ready to describe how the priority-queue operations are realized.

find-min(Q)**:** Following the first invariant, the minimum is at t_1.

insert(Q, x)**:** A new node x is given with the value e. If e is smaller than the value of t_1, the roles of x and t_1 are exchanged by swapping the two nodes. The node x is then added below t_1.

meld(Q, Q')**:** This operation involves at most four trees T_1, T_2, T'_1, and T'_2, two for each priority queue; their roots are named correspondingly using lowercase letters. Assume without loss of generality that $value(t_1) \leq value(t'_1)$. The tree T_1 becomes the first tree of the melded priority queue. The violation array of t'_1 is dismissed. If T_1 has the maximum rank, the other trees are added below t_1 resulting in no second tree for the melded priority queue. Otherwise, the tree with the maximum rank among T'_1, T_2, and T'_2 becomes the second tree of the melded priority queue. The remaining trees are added below the root of this tree, and the roots of the added trees are made violating. To keep the number of active violations within the threshold, two violation reductions are performed if possible. Finally, the regular counters that are no longer corresponding to roots are dismissed.

decrease(Q, x, e)**:** The element at node x is replaced by element e. If e is smaller than the element at t_1, the roles of x and t_1 are exchanged by swapping the two nodes (but not their violation lists). If x is either t_1, t_2, or a child of t_1, stop. Otherwise, x is denoted violating and added to the violation structure of t_1; if x was already violating, it is removed from the violation structure where it was in. To keep the number of active violations within the threshold, a violation reduction is performed if possible.

delete-min(Q)**:** By the first invariant, the minimum is at t_1. The node t_2 and all the subtrees rooted at its children are added below t_1. This is accompanied with extending the violation array of T_1, and dismissing the regular counter of T_2. By the second invariant, the new minimum is now stored at one of the children or violation nodes of t_1. By Lemma 1 and the third invariant, the number of minimum candidates is $O(\lg n)$. Let x be the node with the new minimum. If x is among the violation nodes of t_1, a tree that has the same rank as x is removed from below t_1, its root is made violating, and is attached in place of the subtree rooted at x. If x is among the children of t_1, the tree rooted at x is removed from below t_1. The inactive violations of t_1 are recorded in the violation array. The violations of x are also recorded in the array. The violation list of t_1 is appended to that of x. The node t_1 is then deleted and replaced by the node x. The old subtrees of x are added, one by one, below the new root. To keep the number of violations within the threshold, violation reductions are performed as many times as possible. By the third invariant, at most $O(\lg n)$ violation reductions are to be performed.

delete(Q, x)**:** The node x is swapped with t_1, which is then made violating. To remove the current root x, the same actions are performed as in *delete-min*.

In our description, we assume that it is possible to dismiss an array in constant time. We also assume that the doubly-linked list indicating the ranks where a reduction is possible is realized inside the violation array, and that a regular counter is compactly represented within an array. Hence, the only garbage created by freeing a violation structure or a regular counter is an array of pointers. If it is not possible to dismiss such an array in constant time, we rely on incremental garbage collection. In that case, to dismiss a violation structure or a regular counter, we add it to the garbage pile, and release a constant amount of garbage in connection with every priority-queue operation. It is not hard to

prove by induction that the sum of the sizes of the violation structures, the regular counters, and the garbage pile remains linear in the size of the priority queue.

4 Violation Reductions

Each time when new violations are introduced, we perform equally many violation reductions whenever possible. A violation reduction is possible if there exists a rank recording at least three active violations. This will fix the maximum number of active violations at $O(\lg n)$. Our violation reductions diminish the number of violations by either getting rid of one violation, getting rid of two and introducing one new violation, or getting rid of three and introducing two new violations. We use the powerful tool that the rank sequence of t_1 obey the rules of a numeral system, which allows adding a new subtree and removing a subtree of a given rank from below t_1 in worst-case constant time. When a subtree with a violating root is added below t_1, its root is no longer violating.

One consequence of allowing $O(n)$ violations is that we cannot use the violation reductions exactly in the form described for run-relaxed heaps or fat heaps. When applying the cleaning transformation to a node of rank r (see [4] for the details), we cannot any more be sure that its sibling of rank $r+1$ is not violating, simply because there can be violations that are guarded by other nodes. We then have to avoid executing the cleaning transformation by the violation reductions.

Let x_1, x_2, and x_3 be three violations of the same rank r. We distinguish several cases to be considered when applying our reductions:

Case 1. If, for some $i \in \{1, 2, 3\}$, x_i is neither the last nor the second-last child, detach the subtree rooted at x_i from its parent and add it below t_1. The node x_i will not be anymore violating. The detachment of x_i may leave one or two groups with one member (but not the last group). If this happens, the subtree rooted at each of these singleton members is then detached and added below t_1. (We can still detach the subtree of x_i even when x_i is one of the last two children of its parent, conditioned that such detachment leaves this last group with at least two members and retains the rank of the parent.)

For the remaining cases, after checking Case 1, we assume that each of x_1, x_2, and x_3 is either the last or the second-last child of its parent. Let s_1, s_2, and s_3 be the other member of the last two members of the groups of x_1, x_2, and x_3, respectively. Let p_1, p_2, and p_3 be the parents of x_1, x_2, and x_3, respectively. Assume without loss of generality that $\mathit{rank}(s_1) \geq \mathit{rank}(s_2) \geq \mathit{rank}(s_3)$.

Case 2. $\mathit{rank}(s_1) = \mathit{rank}(s_2) = r+1$, or $\mathit{rank}(s_1) = \mathit{rank}(s_2) = r$, or $\mathit{rank}(s_1) = r$ and $\mathit{rank}(s_2) = r - 1$:

(a) $\mathit{value}(p_1) \leq \mathit{value}(p_2)$: Detach the subtrees rooted at x_1 and x_2, and add them below t_1; this reduces the number of violations by two. Detach the subtree rooted at s_2 and attach it below p_1 (retain rank order); this does not introduce any new violations. Detach the subtree rooted at the last child of p_2 if it is a singleton member, detach the remainder of the

subtree rooted at p_2, change the rank of p_2 to one more than that of its current last child, and add the resulting subtrees below t_1. Remove a subtree with the old rank of p_2 from below t_1, make its root violating, and attach it in the old place of the subtree rooted at p_2.

(b) $value(p_2) < value(p_1)$: Change the roles of x_1, s_1, p_1 and x_2, s_2, p_2, and apply the same actions as in Case 2(a).

Case 3. $rank(s_1) = r + 1$ and $rank(s_2) = r$:

(a) $value(p_1) \leq value(p_2)$: Apply the same actions as in Case 2(a).

(b) $value(p_2) < value(p_1)$: Detach the subtrees rooted at x_1 and x_2, and add them below t_1; this reduces the number of violations by two. Detach the subtree rooted at s_2 if it becomes a singleton member of its group, detach the remainder of the subtree rooted at p_2, change the rank of p_2 to one more than that of its current last child, and add the resulting subtrees below t_1. Detach the subtree rooted at s_1, and attach it in the old place of the subtree rooted at p_2; this does not introduce any new violations. Detach the subtree rooted at the current last child of p_1 if such child becomes a singleton member of its group, detach the remainder of the subtree rooted at p_1, change the rank of p_1 to one more than that of its current last child, and add the resulting subtrees below t_1. Remove a subtree of rank $r + 2$ from below t_1, make its root violating, and attach it in the old place of the subtree rooted at p_1.

Case 4. $rank(s_1) = rank(s_2) = rank(s_3) = r - 1$:

Assume without loss of generality that $value(p_1) \leq \min\{value(p_2), value(p_3)\}$. Detach the subtrees of x_1, x_2, and x_3, and add them below t_1; this reduces the number of violations by three. Detach the subtrees of s_2 and s_3, and join them to form a subtree of rank r. Attach the resulting subtree in place of x_1; this does not introduce any new violations. Detach the subtree rooted at the current last child of each of p_2 and p_3 if such child becomes a singleton member of its group, detach the remainder of the subtrees rooted at p_2 and p_3, change the rank of each of p_2 and p_3 to one more than that of its current last child, and add the resulting subtrees below t_1. Remove two subtrees of rank $r + 1$ from below t_1, make the roots of each of them violating, and attach them in the old places of the subtrees rooted at p_2 and p_3.

Case 5. $rank(s_1) = r + 1$ and $rank(s_2) = rank(s_3) = r - 1$:

(a) $value(p_1) \leq \min\{value(p_2), value(p_3)\}$: Apply same actions as Case 4.

(b) $value(p_2) < value(p_1)$: Apply the same actions as in Case 3(b).

(c) $value(p_3) < value(p_1)$: Change the roles of x_2, s_2, p_2 to x_3, s_3, p_3, and apply the same actions as in Case 3(b).

The following properties are the keys for the success of our violation-reduction routines. 1) Since there is no tree T_2 when a violation reduction takes place, the rank of t_1 will be the maximum rank among all other nodes. In accordance, we can remove a subtree of any specified rank from below t_1. 2) Since t_1 has the minimum value of the priority queue, its children are not violating. In accordance, we can add a subtree below t_1 and ensure that its root is not violating.

5 Extended Regular Binary Counters

An extended regular binary counter represents a non-negative integer n as a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of digits, least-significant digit first, such that $d_i \in \{0, 1, 2, 3\}$, $d_{\ell-1} \neq 0$, and $n = \sum_i d_i \cdot 2^i$. The main constraint is that every 3 is preceded by a 0 or 1 possibly having any number of 2's in between, and that every 0 is preceded by a 2 or 3 possibly having any number of 1's in between. This constraint is stricter than the standard one, which allows the first 3 and the first 0 to come after any (or even no) digits. An extended regular counter [3, 11] supports the increments and decrements of arbitrary digits with a constant number of digit changes per operation.

Brodal [2] showed how, what he called, a *guide* can realize a regular binary counter (the digit set has three symbols and the counter supports *increments* in constant time). To support *decrements* in constant time as well, he suggested to couple two such guides back to back, and accordingly used six symbols altogether. We show how to implement, in a simpler way, an extended regular binary counter more efficiently. The construction described here was sketched in [11]; our description is more detailed and more precise.

We partition any sequence into *blocks* of consecutive digits, and digits that are in no blocks. We have two categories of blocks: blocks that end with a 3 are of the forms $12^*3, 02^*3$, and blocks that end with a 0 are of the forms $21^*0, 31^*0$ (we assume that least-significant digits come first, and * means zero or more repetitions). We call the last digit of a block the *distinguishing digit*, and the other digits of the block the *block members*. Note that the distinguishing digit of a block may be the first member of a block from the other category.

To efficiently implement *increment* and *decrement* operations, a *fix* is performed at most twice per operation. A fix does not change the value of a number. When a digit that is a member of a block is increased or decreased by one, we may need to perform a fix on the distinguishing digit of its block. We associate a forward pointer f_i with every digit d_i, and maintain the invariant that all the members of the same block point to the distinguishing digit of that block. The forward pointers of the digits that are not members of a block point to an arbitrary digit. Starting from any block member, we can access the distinguishing digit of its block, and hence perform the required fix, in constant time.

As a result of an *increment* or a *decrement*, the following properties make such a construction possible.

- A block may only extend from the beginning and by only one digit. In accordance, the forward pointer of this new block member inherits the same value as the forward pointer of the following digit.
- A newly created block will have only two digits. In accordance, the forward pointer of the first digit is made to point to the other digit.
- A fix that is performed unnecessarily is not harmful (keeps the representation regular). In accordance, if a block is destroyed when fixing its distinguishing digit, no changes are done with the forward pointers.

A string of length zero represents the number 0. In our pseudo-code, the change in the length of the representation is implicit; the key observation is that the

length can only increase by at most one digit with an *increment* and decrease by at most two digits with a *decrement*.

For $d_j = 3$, a *fix-carry* for d_j is performed as follows:

Algorithm *fix-carry*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$

assert $0 \leq j \leq \ell - 1$ **and** $d_j = 3$
$d_j \leftarrow 1$
increase d_{j+1} by 1
if $d_{j+1} = 3$
    $f_j \leftarrow j + 1$ // a new block of two digits
else
    $f_j \leftarrow f_{j+1}$ // extending a *possible* block from the beginning

As a result of a *fix-carry* the value of a number does not change. Accompanying a *fix-carry* two digits are changed. In the corresponding data structure, this results in performing a *join*, which involves one element comparison.

The following pseudo-code summarizes the actions needed to increase the ith digit of a number by one.

Algorithm *increment*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$

assert $0 \leq i \leq \ell$
if $d_i = 3$
    fix-carry$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$ // either this fix is executed
$j \leftarrow f_i$
if $j \leq \ell - 1$ **and** $d_j = 3$
    fix-carry$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$
increase d_i by 1
if $d_i = 3$
    fix-carry$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$ // or this fix

Using case analysis, it is not hard to verify that this operation maintains the regularity of the representation.

For $d_j = 0$, a *fix-borrow* for d_j is performed as follows:

Algorithm *fix-borrow*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$

assert $0 \leq j < \ell - 1$ **and** $d_j = 0$
$d_j \leftarrow 2$
decrease d_{j+1} by 1
if $d_{j+1} = 0$
    $f_j \leftarrow j + 1$ // a new block of two digits
else
    $f_j \leftarrow f_{j+1}$ // extending a *possible* block from the beginning

As a result of a *fix-borrow* the value of a number does not change. Accompanying a *fix-borrow* two digits are changed. In the corresponding data structure, this results in performing a *split*, which involves no element comparisons.

The following pseudo-code summarizes the actions needed to decrease the ith digit of a number by one.

Algorithm *decrement*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$

```
1:  assert 0 ≤ i ≤ ℓ − 1
2:  if d_i = 0
3:     fix-borrow(⟨d_0, d_1, ..., d_{ℓ−1}⟩, i)   // either this fix is executed
6:  j ← f_i
7:  if j < ℓ − 1 and d_j = 0
8:     fix-borrow(⟨d_0, d_1, ..., d_{ℓ−1}⟩, j)
8:  decrease d_i by 1
10: if i < ℓ − 1 and d_i = 0
11:    fix-borrow(⟨d_0, d_1, ..., d_{ℓ−1}⟩, i)   // or this fix
```

Using case analysis, it is not hard to verify that this operation maintains the regularity of the representation.

In our application, the outcome of splitting a tree of rank $r+1$ is not always two trees of rank r as assumed above. The outcome can also be one tree of rank r and another of rank $r+1$; two trees of rank r and a third tree of a smaller rank; or one tree of rank r, one of rank $r-1$, and a third tree of a smaller rank. We can handle the third tree, if any, by adding it below t_1 (executing an *increment* just after the *decrement*). The remaining two cases, where the *split* results in one tree of rank r and another of rank either $r+1$ or $r-1$, are pretty similar to the case where we have two trees of rank r. In the first case, we have detached a tree of rank $r+1$ and added a tree of rank r and another of rank $r+1$; this case maintains the representation regular. In the second case, we have detached a tree of rank $r+1$ and added a tree of rank r and another of rank $r-1$; after that, there may be three or four trees of rank $r-1$, and one *join* (executing a *fix-carry*) at rank $r-1$ may be necessary to make the representation regular. In the worst case, a *fix-borrow* may require three element comparisons: two because of the extra addition (*increment*) and one because of the extra *join*. A *decrement*, which involves two fixes, may then require up to six element comparisons.

6 Conclusions

We showed that a simpler data structure achieving the same asymptotic bounds as Brodal's data structure [2] exists. We were careful not to introduce any artificial complications when presenting our data structure. Our construction reduces the constant factor hidden behind the big-Oh notation for the worst-case number of element comparisons performed by *delete-min*. Theoretically, it would be interesting to know how low such factor can get.

Our storage requirements are as follows. Every node stores an element, a rank, two sibling pointers, a last-child pointer, a pointer to its violation list, and two pointers for the violation list it may be in. The violation structure and the regular counters require logarithmic space. The good news is that we save four pointers per node in comparison with [2]. The bad news is that, for example, binomial queues [13] can be implemented with only two pointers per node.

We summarize the main differences between our construction and that in [2].

+ We use a standard numeral system with fewer digits (four instead of six). Besides improving the constant factors, this allows for the logical distinction between the operations of the numeral system and other operations.
+ We use normal joins instead of three-way joins, each involving one element comparison instead of two.
+ We do not use parent pointers, except for the last children. This saves one pointer per node and allows swapping of nodes in constant time. Since node swapping is possible, elements can be stored directly inside nodes; this saves two more pointers per node.
+ We gathered the violations associated with every node in one violation list, instead of two; this saves one more pointer per node.
+ We bound the number of violations by restricting their total number to be within a threshold, whereas the treatment in [2] imposes an involved numeral system that constrains the number of violations per rank.
− We check more cases within our violation reductions.
− We have to deal with larger ranks; the maximum rank may go up to $1.44 \lg n$, instead of $\lg n$.

References

1. Brodal, G.S.: Fast meldable priority queues. In: 4th International Workshop on Algorithms and Data Structures. LNCS, vol. 955, pp. 282–290. Springer (1995)
2. Brodal, G.S.: Worst-case efficient priority queues. In: 7th ACM-SIAM Symposium on Discrete Algorithms. pp. 52–58. ACM/SIAM (1996)
3. Clancy, M.J., Knuth, D.E.: A programming and problem-solving seminar. Report STAN-CS-77-606. Computer Science Department, Stanford University (1977)
4. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Communications of the ACM 31(11), 1343–1354 (1988)
5. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms 5(1), Article 14 (2008)
6. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Informatica 45(3), 193–210 (2008)
7. Elmasry, A., Jensen, C., Katajainen, J.: Strictly-regular number system and data structures. In: 12th Scandinavian Symposium and Workshops on Algorithm Theory. LNCS, vol. 6139, pp. 26–37. Springer (2010)
8. Elmasry, A., Katajainen, J.: Worst-case optimal priority queues via extended regular counters, E-print arXiv:1112.0993. arXiv.org (2011)
9. Elmasry, A., Katajainen, J.: Fat heaps without regular counters. In: 6th Workshop on Algorithms and Computation. LNCS, vol. 7157, pp. 173–185. Springer (2012)
10. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM 34(3), 596–615 (1987)
11. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and Boolean union-find. In: 34th Annual ACM Symposium of Theory of Computing. pp. 573–582. ACM (2002)
12. Kaplan, H., Tarjan, R.E.: New heap data structures. Report TR-597-99. Department of Computer Science, Princeton University (1999)
13. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM 21(4), 309–315 (1978)
14. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM 7(6), 347–348 (1964)