

The Magic of a Number System^{*}

Amr Elmasry¹, Claus Jensen², and Jyrki Katajainen³

¹ Max-Planck Institut für Informatik, Saarbrücken, Germany

² The Royal Library, Copenhagen, Denmark

³ Department of Computer Science, University of Copenhagen, Denmark

Abstract. We introduce a new number system that supports increments with a constant number of digit changes. We also give a simple method that extends any number system supporting increments to support decrements using the same number of digit changes. In the new number system the weight of the i th digit is $2^i - 1$, and hence we can implement a priority queue as a forest of heap-ordered complete binary trees. The resulting data structure guarantees $O(1)$ worst-case cost per *insert* and $O(\lg n)$ worst-case cost per *delete*, where n is the number of elements stored.

1 Introduction

The interrelationship between numerical representations and data structures is efficacious. As far as we know, the issue was first discussed in the paper by Vuillemin on binomial queues [14] and the seminar notes by Clancy and Knuth [5]. However, in many write-ups such connection has not been made explicit. In this paper, we introduce a new number system and use it to develop a priority queue that, in a sense, utilizes the structure of Williams' binary heap [15].

In the computing literature, many types of priority queues have been studied. Sometimes it is sufficient to construct priority queues that support the elementary operations *find-min*, *insert*, and *delete*. Our binary-heap variant supports *find-min* and *insert* at $O(1)$ worst-case cost, and *delete* at $O(\lg n)$ worst-case cost, n denoting the number of elements stored prior to the operation. In contrast, $\Omega(\lg \lg n)$ is known to be a lower bound on the worst-case complexity of *insert* for the standard binary heaps [9].

In a positional numeral system, a sequence of digits $\langle d_0, d_1, \dots, d_{k-1} \rangle$ is used to represent a positive integer, k being the length of the representation. By convention, d_0 is the least-significant digit and d_{k-1} the most-significant digit. If w_i is the weight of d_i , $\langle d_0, d_1, \dots, d_{k-1} \rangle$ represents the (decimal) number $\sum_{i=0}^{k-1} d_i w_i$.

^{*} © 2010 Springer-Verlag. This is the authors' version of the work. The original publication is available at www.springerlink.com with DOI 10.1007/978-3-642-13122-6_17.

The work of the authors was partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools"). A. Elmasry was supported by the Alexander von Humboldt Foundation and the VELUX Foundation.

Table 1. The number systems used in some priority queues and their effect on the complexity of *insert*. All the mentioned structures can support *find-min* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost, n is the current size of the data structure.

Digits in use	Binomial-queue variants	Binary-heap variants
$\{0, 1\}$	$O(\lg n)$ worst case & $O(1)$ amortized [14]	$O(\lg^2 n)$ worst case [folklore]
$\{0, 1\}$ & first non-zero digit may be 2	$O(1)$ worst case [3]	$O(\lg n)$ worst case & $O(1)$ amortized [2, 11] ^a
$\{0, 1, 2\}$	$O(1)$ worst case [4, 7]	$O(\lg n)$ worst case [13] ^b & $O(1)$ amortized [this paper] ^a
$\{1, 2, 3, 4\}$	$O(1)$ worst case [8] ^a	
$\{0, 1, 2, 3, 4\}$		$O(1)$ worst case [this paper]

^a *borrow* has $O(1)$ worst-case cost.

^b *meld* has $O(\lg m \cdot \lg n)$ worst-case cost, where m and n are the sizes of the data structures melded.

Different numerical representations are obtained by enforcing different invariants for the values that d_i and w_i can take for $i \in \{0, 1, \dots, k-1\}$. In accordance, the performance characteristics of some operations may vary for different number systems. Important examples of number systems include the *binary system*, where $d_i \in \{0, 1\}$ and $w_i = 2^i$; the *redundant binary system*, where $d_i \in \{0, 1, 2\}$ and $w_i = 2^i$; the *skew binary system*, where $w_i = 2^{i+1} - 1$; the *canonical skew binary system* [10], where $w_i = 2^{i+1} - 1$ and $d_i \in \{0, 1\}$ except that the first non-zero digit may also be 2; and the *zeroless variants*, where $d_i \neq 0$. These systems and some other number systems, together with some priority-queue applications, are discussed in [12, Chapter 9]. We have gathered the most relevant earlier results related to the present study in Table 1.

A binomial queue is a forest of heap-ordered binomial trees [14]. If the queue stores n elements and the binary representation of n contains a 1-bit at position i , $i \in \{0, 1, \dots, \lfloor \lg n \rfloor\}$, the queue contains a tree of size 2^i . In the binary number system, an addition of two 1-bits at position i results in a 1-bit at position $i+1$. Correspondingly, in a binomial queue two trees of size 2^i are linked resulting in a tree of size 2^{i+1} . For binomial trees, this linking is possible at $O(1)$ worst-case cost. Since *insert* corresponds to an increment of an integer, *insert* may have logarithmic cost due to the propagation of carries. Instead of relying on the binary system, some of the aforementioned or other specialized variants could be used to avoid cascading carries. That way a binomial queue can support *insert* at $O(1)$ worst-case cost [3, 4, 7, 8]. A binomial queue based on a zeroless system where $d_i \in \{1, 2, 3, 4\}$ also supports the removal of an unspecified element—an operation that we call *borrow*—at $O(1)$ worst-case cost [8].

An approach similar to that used for binomial queues has been proposed for binary heaps too. The components in this case are either *perfect heaps* [2, 11]

or *pennants* [13]. A perfect heap is a heap-ordered complete binary tree, and accordingly is of size $2^i - 1$ where $i \geq 1$. A pennant is a heap-ordered tree whose root has one subtree that is a complete binary tree, and accordingly is of size 2^i where $i \geq 0$. In contrary to binomial trees, the worst-case cost of linking two pennants of the same size is logarithmic, not constant. To link two perfect heaps of the same size, we even need to have an extra node, and if this node is arbitrary chosen the cost per link is as well logarithmic. When perfect heaps (pennants) are used, it is natural to rely on the skew (redundant) binary system. Because of the cost of linking, this approach only guarantees $O(\lg n)$ worst-case cost [13] and $O(1)$ amortized cost per *insert* [2, 11].

Our most interesting contribution is the new number system which uses five symbols and skew weights. As the title of the paper indicates, it may look mysterious why the number system works as effectively as it does; a question that we answer in Section 2. As a by-product, we show how any number system supporting increments can be extended to support decrements with the same number of digit changes; we expect this simple technique to be helpful in the design of new number systems. The application to binary heaps is discussed in Section 3.

2 The Number System

We represent an integer n as a sequence of digits $\langle d_0, d_1, \dots, d_{k-1} \rangle$, least-significant digit first, such that

- $d_i \in \{0, 1, 2, 3, 4\}$ for all $i \in \{0, 1, \dots, k-1\}$,
- $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \dots, k-1\}$, and
- the decimal value of n is $\sum_{i=0}^{k-1} d_i w_i$.

Remark 1. In general, a skew binary number system that uses five symbols is redundant, i.e. there is possibly more than one representation for the same integer. However, for our system, the way the operations are performed guarantees a unique representation for any integer.

2.1 Operations

We define two operations on sequences of digits. An *increment* increases the value of the corresponding integer by 1, and a *decrement* decreases the value by 1. Each operation involves at most four digit changes. We say that a sequence of digits is *valid* if it can be procured by repeatedly performing the increment operation starting from zero. It follows from the correctness proof of Section 2.2 that every valid sequence in our number system has $d_i \in \{0, 1, 2, 3, 4\}$.

Increment. Assume that d_j is equal to 3 or 4. We define how to perform a *fix* for d_j as follows:

1. Decrease d_j by 3.

2. Increase d_{j+1} by 1.
3. If $j \neq 0$, increase d_{j-1} by 2.

Remark 2. Since $w_0 = 1$, $w_1 = 3$, and $3w_i = w_{i+1} + 2w_{i-1}$ for $i \geq 1$, the fix does not change the value of the number.

To increment a number, we perform the following steps:

1. Increase d_0 by 1.
2. Find the smallest j where $d_j \in \{3, 4\}$. If no such digit exists, set j to -1 .
3. If $j \neq -1$, perform a fix for d_j .
4. Push j onto an undo stack.

Remark 3. When defining all valid sequences by means of increments, we make the representation of every integer unique. For example, decimal numbers from 1 to 30 are represented by the following sequences: 1, 2, 01, 11, 21, 02, 12, 22, 03, 301, 111, 211, 021, 121, 221, 031, 302, 112, 212, 022, 122, 222, 032, 303, 113, 2301, 0401, 3111, 1211, 2211.

Remark 4. If we do not insist on doing the fix at the smallest index j where $d_j \in \{3, 4\}$, the representation may become invalid. For example, starting from 22222, which is valid, two increments will subsequently give 03222 and 30322. If we now repeatedly fix the second 3 in connection with the forthcoming increments, after three more increments we will end up at 622201. Actually, one can show that d_0 can get as high as $\Theta(k^2)$ for a k -digit representation.

Decrement. We define the *unfix*, as the reverse of the fix, as follows:

1. Increase d_j by 3.
2. Decrease d_{j+1} by 1.
3. If $j \neq 0$, decrease d_{j-1} by 2.

As a result of the increments, we maintain an undo stack containing the positions where the fixes have been performed. To decrement a number, we perform the following steps:

1. Pop the index at the top of the stack; let it be j .
2. If $j \neq -1$, perform an unfix for d_j .
3. Decrease d_0 by 1.

Remark 5. Since a decrement is the reverse of an increment, the correctness of a decrement operation (that it creates a valid sequence) directly follows from the correctness of the increment.

Remark 6. After any sequence of increments and decrements, the stack size will be equal to the difference between the number of increments and decrements, i.e. the value of the number.

2.2 Correctness

To prove that the operations work correctly, we only need to show that by applying any number of increments starting from zero every digit satisfies $d_i \in \{0, 1, 2, 3, 4\}$. In a fix, although we increase d_{j-1} by 2, no violations could happen as d_{j-1} was at most 2 before the increment. So, a violation would only be possible if, before the increment, d_0 or d_{j+1} was 4.

Define a *block* to be a maximal sequence, none of its digits is 3 or 4 except the last digit. Hence, any sequence representing a number consists of a sequence of blocks, if any, followed by a sequence of digits not in a block, if any, called the *tail*. Since every digit in the tail is less than 3, increasing any of its digits by 1 keeps the sequence valid.

To characterize valid sequences, we borrow some notions from the theory of automata and formal languages. We use d^* to denote the set containing zero or more repetitions of the digit d . Let $\mathcal{S} = \{S_1, S_2, \dots\}$ and $\mathcal{T} = \{T_1, T_2, \dots\}$ be two sets of sequences of digits. We use $\mathcal{S} \mid \mathcal{T}$ to denote the set containing all sequences in \mathcal{S} and \mathcal{T} . We write $\mathcal{S} \subseteq \mathcal{T}$ if for every $S_i \in \mathcal{S}$ there exists $T_j \in \mathcal{T}$ such that $S_i = T_j$, and $\mathcal{S} = \mathcal{T}$ if $\mathcal{S} \subseteq \mathcal{T}$ and $\mathcal{T} \subseteq \mathcal{S}$. We also write $\mathcal{S} \xrightarrow{+} \mathcal{T}$ indicating that the sequence T results by applying an increment to S , and $\mathcal{S} \xrightarrow{+} \mathcal{T}$ if for each $S_i \in \mathcal{S}$ there exists $T_j \in \mathcal{T}$ such that $S_i \xrightarrow{+} T_j$. Furthermore, we write $\overline{\mathcal{S}}$ for a sequence that results from \mathcal{S} by increasing its least-significant digit by 1 without performing a fix, and $\overline{\mathcal{S}}$ for $\{\overline{S_1}, \overline{S_2}, \dots\}$. To capture the intricate structure of valid sequences, we recursively define the following sets of sequences.

$$\tau \stackrel{\text{def}}{=} (1^*2^*)^* \quad (1)$$

$$\alpha \stackrel{\text{def}}{=} 2^*1\gamma \quad (2)$$

$$\beta \stackrel{\text{def}}{=} \tau \mid 2^*3\psi \quad (3)$$

$$\gamma \stackrel{\text{def}}{=} 1 \mid 2\tau \mid 3\beta \mid 4\psi \quad (4)$$

$$\psi \stackrel{\text{def}}{=} 0\gamma \mid 1\alpha \quad (5)$$

$$\phi \stackrel{\text{def}}{=} (21 \mid 02 \mid 12)\tau \quad (6)$$

Remark 7. The intersections among the defined sets are non-empty.

The above definitions imply that

$$\begin{aligned} \overline{\beta} &= 1 \mid 2\tau \mid 3\tau \mid 32^*3\psi \mid 4\psi \\ &= 1 \mid 2\tau \mid 3(\tau \mid 2^*3\psi) \mid 4\psi \\ &= 1 \mid 2\tau \mid 3\beta \mid 4\psi \\ &= \gamma \\ \overline{\psi} &= 1\gamma \mid 2\alpha \\ &= 1\gamma \mid 22^*1\gamma \\ &= \alpha \end{aligned}$$

Next, we show that any sequence representing an integer in our number system can be fully characterized. More precisely, such sequences can be classified into a fixed number of sets, that we call *states*, where every increment is equivalent to a transition whose current and resulting states are uniquely determined from the sequence. Since this state space is closed under the transitions, and each is characterized by a set of sequences of digits with $d_i \in \{0, 1, 2, 3, 4\}$, the correctness of the increment operation follows.

Define the following nine states: 12α , 22β , 03β , 30γ , 11γ , 23ψ , 04ψ , 31α , and ϕ . Next, we show that the following are the only possible transitions.

1. $\frac{12\alpha \xrightarrow{+} 22\beta}{12\alpha = 122^*1\gamma = 122^*1(1 \mid 2\tau \mid 3\beta \mid 4\psi)}$
 $\xrightarrow{+} 22\tau \mid 222^*3(0\bar{\beta} \mid 1\bar{\psi}) = 22\tau \mid 222^*3(0\gamma \mid 1\alpha) = 22(\tau \mid 2^*3\psi) = 22\beta$
2. $\frac{22\beta \xrightarrow{+} 03\beta}{\text{Obvious.}}$
3. $\frac{03\beta \xrightarrow{+} 30\gamma}{03\beta \xrightarrow{+} 30\bar{\beta} = 30\gamma}$
4. $\frac{30\gamma \xrightarrow{+} 11\gamma}{\text{Obvious.}}$
5. $\frac{11\gamma \xrightarrow{+} \phi \mid 23\psi}{11\gamma = 11(1 \mid 2\tau \mid 3\beta \mid 4\psi)}$
 $\xrightarrow{+} \phi \mid 23(0\bar{\beta} \mid 1\bar{\psi}) = \phi \mid 23(0\gamma \mid 1\alpha) = \phi \mid 23\psi$
6. $\frac{23\psi \xrightarrow{+} 04\psi}{\text{Obvious.}}$
7. $\frac{04\psi \xrightarrow{+} 31\alpha}{04\psi \xrightarrow{+} 31\bar{\psi} = 31\alpha}$
8. $\frac{31\alpha \xrightarrow{+} 12\alpha}{\text{Obvious.}}$
9. $\frac{\phi \xrightarrow{+} \phi \mid 22\beta}{\phi = (21 \mid 02 \mid 12)\tau \xrightarrow{+} \phi \mid 22\beta}$

Remark 8. By Remark 3, the numbers from 1 up to 21 (whose decimal equivalent is 5) are valid, so we may assume that ϕ is the initial state.

2.3 Properties

The following lemma directly follows from the definition of the sequence families.

Lemma 1.

- The body of a block ending with 4 constitutes either 0 or 12^*1 .
- The body of a block ending with 3 constitutes either 0, 12^*1 , or 2^* .
- Each 4, 23 and 33 is followed by either 0 or 1.

– There can be at most one 0 in the tail, which must then be its first digit.

The next lemma bounds the average of the digits of any valid sequence to be at most 2.

Lemma 2. *If $\langle d_0, d_1, \dots, d_{k-1} \rangle$ is a representation of a number in our number system, then $\sum_{i=0}^{k-1} d_i \leq 2k$. If k' denotes the number of the digits constituting the blocks of a number, then $2k' - 1 \leq \sum_{i=0}^{k'-1} d_i \leq 2k'$.*

Proof. We prove the second part of the lemma, which implies the first part following the fact that any digit in the tail is at most 2. First, we show by induction that the sum of the digits of a subsequence of the form $\alpha, \beta, \gamma, \psi$ is respectively $\sum_\alpha = 2\ell_\alpha, \sum_\beta = 2\ell_\beta, \sum_\gamma = 2\ell_\gamma + 1, \sum_\psi = 2\ell_\psi - 1$, where $\ell_\alpha, \ell_\beta, \ell_\gamma, \ell_\psi$ are the lengths of the corresponding subsequences when ignoring the trailing digits that are not in a block. The base case is for the subsequence solely consisting of the digit 3, which is a type- γ subsequence with $\ell_\gamma = 1$ and $\sum_\gamma = 3$. From definition (2), $\sum_\alpha = 2(\ell_\alpha - \ell_\gamma - 1) + 1 + \sum_\gamma = 2(\ell_\alpha - \ell_\gamma - 1) + 1 + 2\ell_\gamma + 1 = 2\ell_\alpha$. From definition (3), $\sum_\beta = 2(\ell_\beta - \ell_\psi - 1) + 3 + \sum_\psi = 2(\ell_\beta - \ell_\psi - 1) + 3 + 2\ell_\psi - 1 = 2\ell_\beta$. From definition (4), $\sum_\gamma = 3 + \sum_\beta = 3 + 2\ell_\beta = 3 + 2(\ell_\gamma - 1) = 2\ell_\gamma + 1$. Alternatively, $\sum_\gamma = 4 + \sum_\psi = 4 + 2\ell_\psi - 1 = 4 + 2(\ell_\gamma - 1) - 1 = 2\ell_\gamma + 1$. From definition (5), $\sum_\psi = \sum_\gamma = 2\ell_\gamma + 1 = 2(\ell_\psi - 1) + 1 = 2\ell_\psi - 1$. Alternatively, $\sum_\psi = 1 + \sum_\alpha = 1 + 2\ell_\alpha = 1 + 2(\ell_\psi - 1) = 2\ell_\psi - 1$. The induction step is accordingly complete, and the above bounds follow.

Consider the subsequence that constitutes the blocks of a number. Let k' be the length of such subsequence. Since any sequence of blocks can be represented in one of the forms: $12\alpha, 22\beta, 03\beta, 30\gamma, 11\gamma, 23\psi, 04\psi, 31\alpha$ (excluding the tail). It follows that $\ell_\alpha, \ell_\beta, \ell_\gamma, \ell_\psi = k' - 2$. A case analysis implies that $\sum_{i=0}^{k'-1} d_i$ either equals $2k' - 1$ or $2k'$ for all cases. \square

3 Application: A Worst-Case Efficient Priority Queue

Let us now use the number system for developing a worst-case efficient priority queue. Recall that a binary heap [15] is a *heap-ordered* binary tree where the element stored at a node is no greater than that stored at its children. We rely on *perfect heaps* that are complete binary trees storing $2^h - 1$ elements for some integer $h \geq 1$. Moreover, our heaps are pointer-based; each node keeps pointers to its parent and children.

As in a binomial queue, which is an ordered collection of heap-ordered binomial trees, in our binary-heap variant we maintain an ordered collection of perfect heaps. A similar approach has been used in several earlier publications [2, 11, 13]. The key difference between our approach and the earlier approaches is the number system in use; we rely on our new number system. Assuming that the number of elements being stored is n and that $\langle d_0, d_1, \dots, d_{\lfloor \lg n \rfloor} \rangle$ is the representation of n in this number system, we maintain the invariant that the number of perfect heaps of size $2^{i+1} - 1$ is exactly d_i .

To keep track of the perfect heaps, we maintain a resizable array whose i th entry points to the roots of the perfect heaps of size $2^i - 1$. Since it is important to access the *big* digits 3 and 4 quickly, we maintain the big digits in a singly-linked list by having an additional *jump pointer* at each array entry. In addition, to support borrowing, we also need an *undo stack* holding the indexes corresponding to the positions where fixes were made. To facilitate fast *find-min*, we maintain a pointer to a root that stores the minimum among all elements.

The basic toolbox for manipulating perfect heaps is described in most textbooks on algorithms and data structures (see, for example, [6, Chapter 6]). We need the function *sift-down* to reestablish the heap order when an element at a node is made larger, and the function *sift-up* to reestablish the heap order when an element at a node is made smaller. Both operations are known to have logarithmic cost in the worst case; *sift-down* performs at most $2 \lg n$ and *sift-up* at most $\lg n$ element comparisons. Note that in *sift-down* and *sift-up* we never move elements but whole nodes. This way the handles to nodes will always remain valid and *delete* operations can be executed without any problems.

In our data structure a fix is emulated by taking three perfect heaps of the same height h , determining which root stores the minimum element (breaking ties arbitrarily), making this node the new root of a perfect heap of height $h + 1$, and making the roots of the other two perfect heaps the children of this new root. The old subtrees of the selected root become perfect heaps of height $h - 1$. That is, starting from three heaps of height h , one new heap of height $h + 1$ and two new heaps of height $h - 1$ are created; this is exactly corresponding to the digit changes resulting from a fix in the number system. After performing the fix on the heaps, the respective changes have to be made in the auxiliary structures (resizable array, jump pointers, and undo stack). The emulation of an unfix is a reverse of these actions. The necessary information indicating the position of the unfix is available at the undo stack. Compared to a fix, the only new ingredient is that, when the root of a perfect heap of height $h + 1$ is made the root of the two perfect heaps of height $h - 1$, *sift-down* is necessary due to the possible changes made in the perfect heaps between the fix and the corresponding unfix; otherwise, it cannot be guaranteed that the heap order is satisfied in the composed tree. Hence, a fix can be emulated at $O(1)$ worst-case cost, whereas an unfix has $O(\lg n)$ worst-case cost involving at most $2 \lg n$ element comparisons.

Because of the minimum pointer, *find-min* has $O(1)$ worst-case cost. In *insert*, a node that is a perfect heap of size 1 is first added to the collection. If the element in that node is smaller than the current minimum, the minimum pointer is updated to point to the new node. Additionally, if the added node creates a big digit, the list of big digits is updated accordingly. Thereafter, the other actions specified for an increment in the number system are emulated. The location of the desired fix can be easily determined by accessing the first in the list of big digits. The worst-case cost of *insert* is $O(1)$ and it may involve at most three element comparisons (one to compare the new element with the minimum and two when performing a fix).

When removing a node it is important that we avoid any interference with the number system and do not change the sizes of the heaps retained by the number system. Hence, we implement *delete* by borrowing a node and then using it to replace the deleted node in the associated perfect heap. This approach guarantees that the number system should only support decrements and unfixes. Now, *borrow* is performed by doing an unfix using the information available at the undo stack, and thereafter removing a perfect heap of size 1 from the data structure. Such a heap must always exist since a fix recorded in the undo stack was preceded by an increment. Due to the cost of the unfix, the worst-case cost of *borrow* is $O(\lg n)$ and it may involve at most $2 \lg n$ element comparisons.

By the aid of *borrow*, it is quite straightforward to implement *delete*. Assuming that the replacement node is different from the node to be deleted, the replacement is done, and *siftdown* or *siftup* is executed depending on the value stored at the replacement node. Because of this process, the root of the underlying perfect heap may change. If this happens, we have to go through the resizable array pointing to the roots of the heaps and update the pointer in one of the entries to point to the new root instead of the old root. A deletion may also invalidate the minimum pointer, so we have to scan all roots to determine the current overall minimum and update the minimum pointer to point to this root. The worst-case cost of all these operations is $O(\lg n)$. In total, the number of element comparisons performed is never larger than $6 \lg n$; *borrow* requires at most $2 \lg n$, *siftdown* (as well as *siftup*) requires at most $2 \lg n$, and the scan over all roots requires at most $2 \lg n$ element comparisons.

Remark 9. As a consequence of Lemma 2, the number of perfect heaps in the priority queue is at most $2 \lg n$.

Remark 10. We can incorporate $O(\lg^2 m + \lg n)$ worst-case *meld*, where m and n are the number of elements in the two melded priority queues and $m \leq n$. In such case, *insert* will have $O(\lg n)$ worst-case cost. Following each *insert* or *meld*, the idea is to perform a fix on every three heaps having the same height until there are at most two heaps per height. Fixes can be performed in arbitrary order; neither the number of fixes nor the resulting representation are affected by the order in which the fixes are done.

To establish these worst-case bounds, we note that a fix on the highest index j , where $d_j > 2$, may propagate to the higher indexes at most $\lg n$ times. The worst-case bound on the cost of *insert* follows. To analyse *meld*, we distinguish between the fixes performed on the lower $\lfloor \lg m \rfloor$ indexes and those performed on the higher indexes. Since there are at most two heaps per height in each queue prior to each *meld*, the number of possible fixes on $d_{\lfloor \lg m \rfloor}$ is at most four (we leave the verification of this fact for the reader); each of these fixes may propagate forward resulting in at most $\lg n$ fixes on the higher indexes. Since the sum of the heights of the heaps corresponding to the lower $\lfloor \lg m \rfloor$ indexes is $O(\lg^2 m)$ and each fix on the lower indexes decreases this quantity by 1, there are $O(\lg^2 m)$ such fixes. The worst-case bound on the cost of *meld* follows.

By extracting the root of the smallest perfect heap and adding the subtrees of that node, if any, to the collection of perfect heaps, this data structure can

support *borrow* at $O(1)$ worst-case cost. In accordance, *delete* can use this kind of borrowing instead.

For this implementation, the amortized costs are: $O(1)$ per *insert*, $O(\lg m)$ per *meld*, and $O(\lg n)$ per *borrow* and *delete*. To establish these bounds, we use a potential function that is the sum of the heights of the heaps currently in the priority queue. The key observation is that a *fix* decreases the potential by 1, *insert* increases it by 1, *borrow* and *delete* increase it by $O(\lg n)$, and *meld* does not change the total potential. Caveat, *meld* involves $O(\lg m)$ work since the perfect heaps have to be maintained in height order.

4 Conclusions

We gave a number system that efficiently supports increments. A disturbance in any of the digits may push the increments out of the orbit, resulting in an invalid representation. In order not to disturb this sensitive system, we implemented decrements as the reverse of increments. This undo-logging technique can be applied to other number systems as well. On the other hand, one may still ask whether there is another way of extending our system to incorporate efficient decrements without using an undo stack.

When applying the number system to implement a variant of Williams' binary heap, we obtained a priority queue that performs *insert* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost. This improves all earlier approaches that implement binary-heap variants. However, there are other ways of achieving the same bounds by using specialized data structures like binomial queues [3, 4, 7, 8], or via general data-structural transformations [1]. Also, the price we pay for $O(1)$ worst-case *insert* is relatively high; we cannot support polylogarithmic *meld*, and the bound on the number of element comparisons performed by *delete* is increased from $2 \lg n$ to $6 \lg n$. It would be natural to ask whether our approach can be improved in any of these aspects.

References

1. Alstrup, S., Husfeld, T., Rauhe, T., Thorup, M.: Black box for constant-time insertion in priority queues. *ACM Transactions on Algorithms* 1(1), 102–106 (2005)
2. Bansal, S., Sreekanth, S., Gupta, P.: M-heap: A modified heap data structure. *International Journal of Foundations of Computer Science* 14(3), 491–502 (2003)
3. Brodal, G. S., Okasaki, C.: Optimal purely functional priority queues. *Journal of Functional Programming* 6(6), 839–857 (1996)
4. Carlsson, S., Munro, J. I., Poblete, P. V.: An implicit binomial queue with constant insertion time. *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory. Lecture Notes in Computer Science*, vol. 318, 1–13 (1988)
5. Clancy, M. J., Knuth, D. E.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Stanford University (1977)
6. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: *Introduction to Algorithms*. 3rd edition. The MIT Press (2009)

7. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. *ACM Transactions on Algorithms* 5(1), Article 14 (2008)
8. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. *Acta Informatica* 45(3), 193–210 (2008)
9. Gonnet, G. H., Munro, J. I.: Heaps on heaps. *SIAM Journal on Computing* 15(4), 964–971 (1986)
10. Myers, E.: An applicative random-access stack. *Information Processing Letters* 17, 241–248 (1983)
11. Harvey, N. J. A., Zatloukal, K.: The post-order heap. *Proceedings of the 3rd International Conference on Fun with Algorithms* (2004)
12. Okasaki, C.: *Purely Functional Data Structures*. Cambridge University Press (1998)
13. Sack, J. -R., Strothotte, T.: A characterization of heaps and its applications. *Information and Computation* 86(1), 69–86 (1990)
14. Vuillemin, J.: A data structure for manipulating priority queues. *Communications of the ACM* 21(4), 309–315 (1978)
15. Williams, J. W. J.: Algorithm 232: Heapsort. *Communications of the ACM* 7(6), 347–348 (1964)