Two Skew-Binary Numeral Systems and One Application*

Amr Elmasry[1], Claus Jensen[2], and Jyrki Katajainen[1]

[1] Department of Computer Science, University of Copenhagen, Denmark
[2] The Royal Library, Copenhagen, Denmark

Abstract. We introduce two numeral systems, the magical skew system and the regular skew system, and contribute to their theory development. For both systems, increments and decrements are supported using a constant number of digit changes per operation. Moreover, for the regular skew system, the operation of adding two numbers is supported efficiently. Our basic message is that some data-structural problems are better formulated at the level of a numeral system. The relationship between number representations and data representations, as well as operations on them, can be utilized for an elegant description and a clean analysis of algorithms. In many cases, a pure mathematical treatment may also be interesting in its own right. As an application of numeral systems to data structures, we consider how to implement a priority queue as a forest of pointer-based binary heaps. Some of the number-representation features that influence the efficiency of the priority-queue operations include weighting of digits, carry-propagation and borrowing mechanisms.

Keywords. Numeral systems, data structures, priority queues, binary heaps

1 Introduction

The interrelationship between numeral systems and data structures is efficacious. As far as we know, the issue was first discussed in the paper by Vuillemin on

A preliminary version of this paper entitled "The magic of a number system" [1] was presented at the 5th International Conference on Fun with Algorithms held on Ischia Island in June 2010.

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

binomial queues [2] and the seminar notes by Clancy and Knuth [3]. However, in many write-ups this connection has not been made explicit.

In a positional numeral system, a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of *digits* d_i, $i \in \{0, 1, \ldots, \ell - 1\}$, is used to represent an integer, ℓ being the length of the representation. By convention, d_0 is the least-significant digit, $d_{\ell-1}$ the most-significant digit, and $d_{\ell-1} \neq 0$. If w_i is the *weight* of d_i, the string represents the value $\sum_{i=0}^{\ell-1} d_i w_i$. In *binary systems* $w_i = 2^i$, and in *skew binary systems* $w_i = 2^{i+1} - 1$. In the *standard binary system* $d_i \in \{0, 1\}$, in a *redundant binary system* $d_i \in \{0, 1, 2\}$, and in the *zeroless variants* $d_i \neq 0$. Other numeral systems include the *regular system* [3], which is a redundant binary system where $d_i \in \{0, 1, 2\}$ conditional on that between every two 2's there is at least one 0; and the *canonical skew system* [4], which is a skew binary system where $d_i \in \{0, 1\}$ except that the first non-zero digit may be 2. These numeral systems and some others, together with their applications to data structures, are discussed in [5, Chapter 9].

The key operations used in the manipulation of number representations include a modification of a specified digit in a number, e.g. by increasing or decreasing a digit by one, and an addition of two numbers. Sometimes, it may also be relevant to support the cutting of a number in two numbers and the concatenation of two numbers. For all operations, the resulting representations must still obey the rules governing the numeral system. An important measure of efficiency is the number of digit changes made by each operation.

As an application, we consider *addressable* and *meldable priority queues*, which store one element per node, and support the following operations:

find-min(Q)**.** Return a handle (pointer) to the node with a minimum element in priority queue Q.

insert(Q, x)**.** Insert node x, already storing an element, into priority queue Q.

borrow(Q)**.** Remove an unspecified node from priority queue Q, and return a handle to that node.

delete-min(Q)**.** Remove a node from priority queue Q with a minimum element, and return a handle to that node.

delete(Q, x)**.** Remove node x from priority queue Q.

meld(Q_1, Q_2)**.** Create and return a new priority queue that contains all the nodes of priority queues Q_1 and Q_2. This operation destroys Q_1 and Q_2.

Since *find-min* is easily accomplished at $O(1)$ worst-case cost by maintaining a pointer to the node storing the current minimum, *delete-min* can be implemented at the same asymptotic worst-case cost as *delete* by calling *delete* with the handle returned by *find-min*. Hence, from now on, we shall concentrate on the operations *insert*, *borrow*, *delete*, and *meld*.

Employing the developed numeral systems, we describe two priority-queue realizations, both implemented as a forest of pointer-based binary heaps, which support *insert* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost, n denoting the number of elements stored in the data structure prior to the operation, and $\lg n$ being a shorthand for $\log_2(\max\{2, n\})$. In contrast, for the array-based implementation of a binary heap [6], $\Omega(\lg \lg n)$ is known to be a

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

Digit set	**Forest of binomial trees**	**Forest of pennants**	**Forest of perfect binary heaps**
$\{0,1\}$	$O(\lg n)$ worst case $O(1)$ amortized [2]	$O(\lg^2 n)$ worst case [8]	$O(\lg^2 n)$ worst case [folklore]
$\{0,1\}$, there may be one 2	$O(1)$ worst case [9]	$O(\lg n)$ worst case [10]	$O(\lg n)$ worst case [§6][a] $O(1)$ amortized [11, 12]
$\{0,1,2\}$	$O(1)$ worst case [13]	$O(\lg n)$ worst case [10] $O(1)$ worst case [8]	$O(1)$ worst case [§6][a]
$\{1,2,3,4\}$	$O(1)$ worst case [14][a]		
$\{0,1,2,3,4\}$			$O(1)$ worst case [§4]

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

lower bound on the worst-case complexity of *insert* [7]. Our second realization supports *borrow* at $O(1)$ worst-case cost and *meld* at $O(\lg^2 m)$ worst-case cost, m denoting the number of elements stored in the smaller priority queue. We summarize relevant results related to the present study in Table 1.

A binomial queue is a forest of heap-ordered binomial trees [2]. If the queue stores n elements and the binary representation of n contains a 1-bit at position i, $i \in \{0, 1, \ldots, \lfloor \lg n \rfloor\}$, the queue contains a tree of size 2^i. In the binary numeral system, an addition of two 1-bits at position i results in a 1-bit carry to 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* would have logarithmic worst-case cost due to the propagation of carries. Instead of relying on the binary system, some of the other specialized variants could be used to avoid cascading carries. That way, a binomial queue can support *insert* at $O(1)$ worst-case cost [9, 13, 14]. A binomial queue based on a zeroless system, where $d_i \in \{1, 2, 3, 4\}$, also supports *borrow* at $O(1)$ worst-case cost [14].

An approach similar to that used for binomial queues has also been considered for binary heaps. The components in this case are either *perfect binary heaps* [11, 12] or *pennants* [10]. A perfect binary heap is a heap-ordered complete binary tree, and accordingly is of size $2^{i+1} - 1$ where $i \geq 0$. 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 contrast to binomial trees, the worst-case cost of linking two pennants of the same size is logarithmic, not constant. To link two perfect binary heaps of the same size, we even need to have an additional node, and if this node is arbitrarily chosen, the worst-case cost per link is also logarithmic. When perfect binary heaps are used, it is natural to rely on a skew binary system. Because of the linking cost, this approach achieves $O(\lg n)$ worst-case cost [10] and $O(1)$ amortized cost per *insert* [11, 12]. By implementing the

linkings incrementally with the upcoming operations, $O(1)$ worst-case cost per *insert* is achievable for pennants [8].

The remainder of this paper is organized as follows. In Section 2, we review the canonical skew system introduced in [4]. Due to its conventional carry-propagation mechanism, in our application a digit change may involve a costly linking. We propose two ways of avoiding this computational bottleneck. First, in Section 3, we introduce the magical skew system that uses five symbols, skew weights, and an unconventional carry-propagation mechanism. It may look mysterious why this numeral system works as effectively as it does; to answer this question, we use an automaton to model the behaviour of the numeral system. We discuss the application of the magical skew system to priority queues in Section 4. Second, in Section 5, we modify the regular system discussed in [3] to get a better fit for our application. Specifically, we use skew binary weights, and support incremental digit changes (this idea was implicit in [8]) making the incremental execution of linkings possible without breaking the invariants of this numeral system. We discuss the application of the regular skew system to priority queues in Section 6. To conclude, in Section 7, we briefly summarize the results proved and the issues left open.

2 Canonical Skew System: A Warm-Up

In this section, we review the *canonical skew system* introduced in [4]. A positive integer n is represented as a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of digits, least-significant digit first, such that

- $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and $d_{\ell-1} \neq 0$,
- if $d_j = 2$, then $d_i = 0$ for all $i \in \{0, 1, \ldots, j - 1\}$,
- $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and
- the value of n is $\sum_{i=0}^{\ell-1} d_i w_i$.

In other words, every string has at most one 2; if a 2 exists, it is the first non-zero digit. In this system, every number is represented uniquely [4].

From the perspective of the related applications (see, for example, [4, 9]), it is important that such number representations efficiently support increments, decrements, and additions. Here, we shall only consider increments and decrements; additions can be realized by converting the numbers to binary form, relying on ordinary binary addition, and converting the result back to skew binary form.

In a computer realization, we would rely on a *sparse representation* [5, Section 9.1], which keeps all non-zero digits together with their positions in a singly-linked list. Only one-way linking is necessary since both the increment and decrement operations access the string of digits from the front, by adding new digits or removing old digits. The main reason for using a sparse representation is to have an immediate access to the first non-zero digit. Consequently, the overall cost of the operations will be proportional to the number of non-zero digits accessed.

In this system, increments are performed as follows:

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

```
1: let d_j be the first non-zero digit, if it exists
2: if d_j exists and d_j = 2
3:     d_j ← 0
4:     increase d_{j+1} by 1
5: else
6:     increase d_0 by 1
```

Clearly, this procedure increases the value of the represented number by one. Also, by a straightforward case analysis, it can be shown that the procedure maintains the representation canonical.

Decrements are equally simple:

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

```
1: assert ⟨d_0, d_1, ..., d_{ℓ-1}⟩ is not empty
2: let d_j be the first non-zero digit
3: decrease d_j by 1
4: if j ≠ 0
5:     d_{j-1} ← 2
```

As a result of the operation, the value of the represented number is reduced by $2^{j+1}-1$ and increased by $2(2^j-1)$, so the total change is -1 as it should be. It is again easy to check that this operation maintains the representation canonical.

The efficiency of the operations and the representation can be summarized as follows.

Theorem 1 ([4]). *The canonical skew system supports increments and decrements at $O(1)$ worst-case cost, and each operation involves at most two digit changes. The amount of space needed for representing a positive integer n while supporting these modifications is $O(\lg n)$.*

Remark 1. With respect to the number of digit changes made, the canonical skew system is very efficient. However, in an application domain, the data structure relying on this system is not necessarily efficient since each digit change may require a costly data-structural operation (cf. Section 6). □

3 Magical Skew System

In this section, we introduce a new skew-binary numeral system, which uses an unconventional carry-propagation mechanism. Because of the tricky correctness proof, we call the system *magical*. In this system, we represent a positive integer n as a string $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ of digits, least-significant digit first, such that

– $d_i \in \{0, 1, 2, 3, 4\}$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and $d_{\ell-1} \neq 0$,
– $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and
– the value of n is $\sum_{i=0}^{\ell-1} d_i w_i$.

Remark 2. In general, a skew binary system that uses five symbols is redundant, i.e. there is possibly more than one representation for the same integer. However, the operational definition of the magical skew system and the way the operations are performed guarantee a unique representation for any integer. □

We define two operations on strings of digits: An increment increases the corresponding value by one, and a decrement decreases the value by one. We define the possible representations of numbers operationally: The empty string ε represents the integer 0, and a positive integer n is represented by the string obtained by starting from ε and performing the increment operation n times. Hence, to compute the representation of n, a naive algorithm performing n increments has $O(n)$ cost since each increment involves a constant amount of work.

We say that a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of digits is *valid* if $d_i \in \{0, 1, 2, 3, 4\}$ for each $i \in \{0, 1, \ldots, \ell - 1\}$. The most interesting part of our correctness proofs is to show that increments and decrements retain the strings valid.

In a computer realization, we use a doubly-linked list to record the digits of a string. This list should grow when the string gets a new non-zero last digit and shrink when the last digit becomes 0. Such a representation is called *dense* [5, Section 9.1].

Remark 3. The number of digits in the string representing a positive integer may only change by one per operation. □

A digit is said to be *high* if it is either 3 or 4, and *low* if it is either 0 or 1. For efficiency reasons, we also maintain two singly-linked lists to record the positions of high and low digits, respectively.

Remark 4. The lists recording the positions of high and low digits should be updated after every operation. Only the first two items per list may change per operation. Accordingly, it is sufficient to implement both lists singly-linked. □

3.1 Increments

Assume that d_j is high. A *fix* for d_j is performed as follows:

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

```
1: assert d_j is high
2: decrease d_j by 3
3: increase d_{j+1} by 1
4: if j ≠ 0
5:     increase d_{j−1} by 2
```

Remark 5. Since $w_0 = 1$, $w_1 = 3$, and $3w_i = w_{i+1} + 2w_{i-1}$ for $i \geq 1$, a fix does not change the value of the represented number. □

The following pseudo-code summarizes the actions to increment a number.

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

1: increase d_0 by 1
2: let d_j be the first digit where $d_j \in \{3, 4\}$, if it exists
3: **if** d_j exists
4: $\quad$ *fix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$

Remark 6. The integers from 1 to 30 are represented by the following strings in our system: 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 7. If we do not insist on performing the fix at the first high digit, 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. □

As for the correctness, we need to show that by starting from 0 and applying any number of increments, in the produced string every digit satisfies $d_i \in \{0, 1, 2, 3, 4\}$. When fixing d_j, although we increase d_{j-1} by 2, no violation is possible 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.

To show that our algorithms are correct, we use some notions of the theory of automata and formal languages. We use d^* to denote the string that contains zero or more repetitions of the digit d. Let $\mathcal{S} = \{S_1, S_2, \ldots\}$ and $\mathcal{T} = \{T_1, T_2, \ldots\}$ be two sets of strings of digits. We use $\mathcal{S} \mid \mathcal{T}$ to denote the set containing all the strings 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 we write $\mathcal{S} = \mathcal{T}$ if $\mathcal{S} \subseteq \mathcal{T}$ and $\mathcal{T} \subseteq \mathcal{S}$.

We also write $S \xrightarrow{+} T$ indicating that the string T results by applying an increment operation to S, and we write $\mathcal{S} \xrightarrow{+} \mathcal{T}$ 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 $\overline{S}$ for a string that results from S by increasing its first digit by one without performing a fix, and $\overline{\mathcal{S}}$ for $\{\overline{S_1}, \overline{S_2}, \ldots\}$. To capture the intricate structure of allowable strings, we define the following rewriting rules, each specifying a set of strings.

$$\tau \stackrel{\text{def}}{=} 2^* \mid 2^*1 \tag{1}$$

$$\alpha \stackrel{\text{def}}{=} 2^*1\gamma \tag{2}$$

$$\beta \stackrel{\text{def}}{=} 2^* \mid 2^*1\tau \mid 2^*3\psi \tag{3}$$

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

$$\psi \stackrel{\text{def}}{=} 0\gamma \mid 1\alpha \tag{5}$$

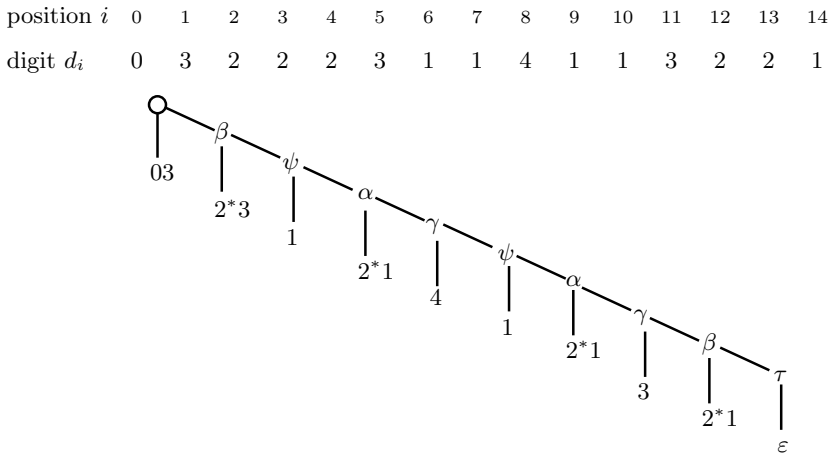

Fig. 1. The unique representation of the integer 100 000 in the magical skew system and its syntactic structure in the form of a parse tree.

Remark 8. The aforementioned rewriting rules can be viewed as production rules of an extended context-free grammar, i.e. one that allows arbitrary regular expressions in the definition of production rules. Actually, it is not difficult to convert these rules into a strictly right-regular form. As an illustration, the syntactic structure of a string in the magical skew system is given in Fig. 1. □

The definitions of the rewriting rules immediately imply that

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

Next, we show that the strings representing the positive integers in the magical skew system can be classified into a number of equivalence classes, that we call *states*. Every increment is equivalent to a transition whose current and next states are uniquely determined by the current string. Since this state space is closed under the transitions, and each contains a set of strings of digits drawn from the set $\{0, 1, 2, 3, 4\}$, the correctness of the increment operation follows.

Define the eleven states: 12α, 22β, 03β, 30γ, 11γ, 23ψ, 04ψ, 31α, 21τ, 02τ, and 12τ. We show that the following are the only possible transitions when performing increments. This implies that the numbers in our system must be represented by one of these states.

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

Remark 9. The strings representing the integers from 1 to 4 in the magical skew system are $1, 2, 01, 11$. These strings are obviously valid, though not in the form of any of the defined states. However, the string 21, which represents the integer 5, is of the form 21τ. So, we may assume that the initial string is 21 and the initial state is 21τ. □

3.2 Decrements

Our objective is to implement decrements as the reverse of increments. Given a string representing a number, we can efficiently identify the current state.

Remark 10. The first two digits are enough to distinguish between all the states except the states 12τ and 12α. To distinguish 12τ from 12α, we need to examine the first items of the lists recording the positions of high and low digits. If the string contains a high digit, then we conclude that the current state is 12α. Otherwise, we check the second item of the list recording the positions of low digits, and compare that to the position of the last digit. If only one low digit exists or if the second low digit is the last digit, the string is of the form 122^* or 122^*1; that is 12τ. On the other hand, if the second low digit is not the last digit, the string is of the form 122^*11, 122^*122^*, or 122^*122^*1; that is 12α. □

Assume that d_j is low. We define an *unfix* as the reverse of a fix:

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

assert d_j is low **and** $j \neq \ell - 1$
increase d_j by 3
decrease d_{j+1} by 1
if $j \neq 0$
    decrease d_{j-1} by 2

The following pseudo-code summarizes the actions to decrement a number:

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

assert the value of $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ is larger than 5
case the current state is in
    $\{12\alpha, 03\beta, 11\gamma, 04\psi, 02\tau\}$: *unfix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, 0)$
    $\{30\gamma, 31\alpha\}$: *unfix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, 1)$
    $\{23\psi\}$: *unfix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, 2)$
    $\{22\beta\}$: let d_j be the first digit where $d_j = 3$, **if** d_j exists
                unfix$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j + 1)$
    $\{12\tau, 21\tau\}$: do nothing
decrease d_0 by 1

Remark 11. If the string is of the form 22β, with the first high digit $d_j = 3$, then d_{j+1} is a low digit (cf. rewriting rules (3) and (5)). By unfixing this low digit, we get the exact reverse of the increment process. □

We write $T \xrightarrow{-} S$ indicating that the string S results by applying a decrement to T, and we write $\mathcal{T} \xrightarrow{-} \mathcal{S}$ if for each $T_i \in \mathcal{T}$ there exists $S_j \in \mathcal{S}$ such that $T_i \xrightarrow{-} S_j$. Furthermore, $\underline{T}$ stands for a string that results from T by decreasing its first digit by one, and $\underline{\mathcal{T}}$ for $\{\underline{T_1}, \underline{T_2}, \ldots\}$. We then have

$$\underline{\gamma} = \beta$$

$$\underline{\alpha} = \psi$$

We show that the following are the only possible transitions when performing decrements. This implies that the numbers in our system must be represented by one of our states.

1. $\underline{22\beta \xrightarrow{-} 12\tau \mid 12\alpha}$
   $22\beta = 22\big(2^* \mid 2^*1\tau \mid 2^*3\psi\big) \xrightarrow{-} 122^* \mid 122^*1\tau \mid 122^*1\big(3\underline{\gamma} \mid 4\underline{\alpha}\big)$
   $= 122^* \mid 122^*1\big(\varepsilon \mid 1 \mid 2\tau\big) \mid 122^*1\big(3\beta \mid 4\psi\big)$
   $= 12\big(2^* \mid 2^*1\big) \mid 122^*1\big(1 \mid 2\tau \mid 3\beta \mid 4\psi\big)$
   $= 12\tau \mid 122^*1\gamma = 12\tau \mid 12\alpha$

2. $\underline{12\alpha \xrightarrow{-} 31\alpha}$
   Obvious.
3. $\underline{31\alpha \xrightarrow{-} 04\psi}$
   $31\alpha \xrightarrow{-} 04\underline{\alpha} = 04\psi$
4. $\underline{04\psi \xrightarrow{-} 23\psi}$
   Obvious.
5. $\underline{23\psi \xrightarrow{-} 11\gamma}$
   $23\psi = 23\big(0\gamma \mid 1\alpha\big) \xrightarrow{-} 11\big(3\underline{\gamma} \mid 4\underline{\alpha}\big) = 11\big(3\beta \mid 4\psi\big) \subseteq 11\gamma$
6. $\underline{11\gamma \xrightarrow{-} 30\gamma}$
   Obvious.
7. $\underline{30\gamma \xrightarrow{-} 03\beta}$
   $30\gamma \xrightarrow{-} 03\underline{\gamma} = 03\beta$
8. $\underline{03\beta \xrightarrow{-} 22\beta}$
   Obvious.
9. $\underline{12\tau \xrightarrow{-} 02\tau}$
   Obvious.
10. $\underline{02\tau \xrightarrow{-} 21\tau}$
    Obvious.
11. $\underline{21\tau \setminus \{21\} \xrightarrow{-} 11\gamma}$
    $21\tau \setminus \{21\} = 21\big(\varepsilon \mid 1 \mid 2\tau\big) \setminus \{21\} = 21\big(1 \mid 2\tau\big) \xrightarrow{-} 11\big(1 \mid 2\tau\big) \subseteq 11\gamma$

Remark 12. For the above state transitions, the proof does not consider a decrement on the five strings from 21 down to 1. □

3.3 Properties

The following lemma directly follows from the state definitions.

Lemma 1. *Define a* block *to be a maximal substring where none of its digits is high, except its last digit. Define the* tail *to be the substring of digits following all the blocks in the representation of a number.*

- *The body of a block ending with* 4 *is either* 0 *or of the form* 12^*1.
- *The body of a block ending with* 3 *is either* 0 *or of the form* 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 provides a bound on the number of digits in any string.

Lemma 2. *For any positive integer* $n \neq 3$*, the number of digits in the string representing* n *in the magical skew system is at most* $\lg n$.

Proof. Inspecting all the state definitions and strings for small integers, the sum of the digits for any of our strings is at least 2, except for the strings 1 and 01. We also note that $d_{\ell-1} \neq 0$. It follows that, for all other strings, either $d_{\ell-1} > 1$ or $d_{\ell-1} = 1$ and $d_j \neq 0$ for some $j \neq \ell - 1$. Accordingly, we have $n \geq 2^\ell$ implying that $\ell \leq \lg n$. □

The following lemma bounds the average of the digits in any of our strings to be at most 2.

Lemma 3. *If $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ is a representation of a positive integer in the magical skew system, then $\sum_{i=0}^{\ell-1} d_i \leq 2\ell$. If ℓ' denotes the number of the digits constituting the blocks of the representation, then $2\ell' - 1 \leq \sum_{i=0}^{\ell'-1} d_i \leq 2\ell'$.*

Proof. We prove the second part of the lemma, which implies the first part using the fact that any digit in the tail is at most 2. First, we show by induction on the length of the strings that the sum of the digits of a substring 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 substrings when ignoring the trailing digits that are not in a block. The base case is for the substring solely consisting of the digit 3, which is a type-γ substring with $\ell_\gamma = 1$ and $\sum_\gamma = 3$. From rewriting rule (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 rewriting rule (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 rewriting rule (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 rewriting rule (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 substring that constitutes the blocks of the representation. Let ℓ' be the length of that substring. Since any sequence of blocks can be represented in one of the forms: 12α, 22β, 03β, 30γ, 11γ, 23ψ, 04ψ, 31α (excluding the tail). It follows that $\ell_\alpha, \ell_\beta, \ell_\gamma, \ell_\psi = \ell' - 2$. A case analysis implies that $\sum_{i=0}^{\ell'-1} d_i$ either equals $2\ell' - 1$ or $2\ell'$ for all cases. □

As a consequence to the previous two lemmas, we get the following corollary.

Corollary 1. *The sum of the digits in the string representing a positive integer n in the magical skew system is at most $2 \lg n$.*

The efficiency of the operations and the representation can be summarized as follows.

Theorem 2. *The magical skew system supports increments and decrements at $O(1)$ worst-case cost, and each operation involves at most four digit changes. The amount of space needed for representing a positive integer n while supporting these modifications is $O(\lg n)$.*

4 Application: A Worst-Case-Efficient Priority Queue

A binary heap [6] is a *heap-ordered* binary tree where the element associated with a node is not greater than that associated with its children. A *perfect binary heap* of height h is a complete binary tree storing $2^h - 1$ elements for an integer $h \geq 1$. Our heaps are pointer-based; each node has 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 version, we maintain an ordered collection of perfect binary heaps. A similar approach has been used in several earlier publications [10–12]. The key difference between our approach and the earlier approaches is the numeral system in use; here we rely on our magical skew system. Assuming that the number of elements being stored is n and that $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ is the representation of n in the numeral system, we maintain the invariant that the number of perfect binary heaps of size $2^{i+1} - 1$ is d_i.

To keep track of the perfect binary heaps, we maintain an auxiliary structure resembling the magical skew system. As before, we maintain a doubly-linked list of digits and two singly-linked lists recording the positions of the high digits and the low digits, respectively. In addition to these lists, we associate each digit with a list of items where each item stores a pointer to the root of a perfect binary heap of that particular size. To facilitate fast *find-min*, we maintain a pointer to a root that is associated with a minimum element.

The basic toolbox for manipulating binary heaps is given in most textbooks on algorithms and data structures (see, for example, [15, Chapter 6]). We need the function *siftdown* to reestablish the heap order when the element associated with a node is made larger, and the function *siftup* to reestablish the heap order when the element associated with a node is made smaller. Both operations are known to have logarithmic cost in the worst case; *siftdown* performs at most $2 \lg n$ and *siftup* at most $\lg n$ element comparisons. Note that in *siftdown* and *siftup* we move whole nodes not elements. In effect, the handles to nodes will always remain valid, and *delete* operations can be executed without problems.

A fix is emulated by involving three perfect binary heaps of the same height h, determining which root is associated with the smallest element, making this node the new root of a perfect binary heap of height $h+1$, and making the roots of the other two perfect binary heaps the children of this new root. The old subtrees of the detached root become perfect binary heaps of height $h-1$. That is, starting with three heaps of height h, one heap of height $h+1$ and two heaps of height $h-1$ are created; this corresponds to the digit changes resulting from a fix in the numeral system. After performing the fix on the heaps, the respective changes have to be made in the auxiliary structure (lists of roots, digits, and positions of high and low digits). The emulation of an unfix is a reverse of these actions. Compared to a fix, the only new ingredient is that, when the root of a perfect binary heap of height $h+1$ is made the root of the two perfect binary heaps of height $h-1$, *siftdown* is necessary. Otherwise, it cannot be guaranteed that the heap order is valid for 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.

In *insert*, a node that is a perfect binary heap of size one is first added to the collection. Thereafter, the other actions specified for an increment in the numeral system are emulated. In effect, if a high or low digit is created, the corresponding list is updated. The location of the desired fix can be easily determined by accessing the first item in the list recording the positions of high

digits. If the element in the inserted node is smaller than the current minimum, the minimum pointer is updated to point to the new node. The worst-case cost of *insert* is $O(1)$ and it involves at most three element comparisons (one to compare the new element with the minimum and two when performing a fix).

When deleting a node, it is important that we avoid any disturbance to the numeral system and do not change the sizes of the heaps. Hence, we implement *delete* by borrowing a node and using it to replace the deleted node in the associated perfect binary heap. This approach guarantees that the numeral system only has to support decrements of the least-significant digit. First, *borrow* performs an unfix, and thereafter it removes a perfect binary heap of size one from the data structure. Due to the cost of the unfix, the worst-case cost of *borrow* is $O(\lg n)$ and it involves at most $2 \lg n$ element comparisons.

By the aid of *borrow*, it is straightforward to implement *delete*. Assuming that the borrowed node is different from the node to be deleted, the replacement is done, and *siftdown* or *siftup* is executed depending on the element associated with the replacement node. Because of this process, the root of the underlying perfect binary heap may change. If this happens, we have to scan through the roots of the heaps and update the pointer, in the item referring to this root, to point to the new root instead. A deletion may also invalidate the minimum pointer. If this happens, 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 at most $6 \lg n$; *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 13. For perfect binary heaps, as for binomial trees [16], the parent-child relationships can be represented using two pointers per node instead of three. In accordance, the amount of extra space can be reduced from $3n + O(\lg n)$ words to $2n + O(\lg n)$ words. □

The above discussion can be summarized as follows.

Theorem 3. *A priority queue that maintains an ordered collection of perfect binary heaps 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. This data structure requires $2n + O(\lg n)$ words of memory, in addition to the elements themselves.*

Remark 14. Our focus has been on good worst-case performance. Alternatively, more efficient amortized performance is achievable through lazy execution [17]. The idea is to maintain the roots of the perfect binary heaps in a doubly-linked list, keep a pointer to the root associated with the current minimum, and let *delete* do most of the work. In *insert*, a new node is added to the list of roots and the minimum pointer is updated if necessary. In *meld*, the two root lists are concatenated and the minimum pointer is set to point to the smaller of the two minima. In *borrow*, a root not pointed to by the minimum pointer is borrowed and its children (if any) are added to the root list. If there is only one tree, its

root is borrowed, its two children are moved to the root list, and the minimum pointer is updated to point to the child whose associated element is smaller.

Clearly, the worst-case cost of *insert*, *borrow*, and *meld* is $O(1)$. In *delete*, a *consolidation* is done: the perfect binary heaps of the same size are gathered together using a temporary array indexed by height and thereafter a fix—as described in Section 3—is performed as many times as possible. It is not difficult to show that the cost of *delete* is $\Theta(n)$ in the worst case.

Nevertheless, the amortized costs are: $O(1)$ per *insert* and *meld*, and $O(\lg n)$ per *borrow* and *delete*, where n is the size of the considered priority queue. 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* increases it by $O(\lg n)$, *meld* does not change the total potential, and *delete* involves $O(\lg n)$ work that is not compensated by a potential reduction. □

5 Regular Skew System

In this section, we reconsider the regular system discussed in [3]. The advantage of the regular system is that it efficiently supports increments, decrements, and additions. We modify this system to handle expensive digit changes. When this modified regular system is used in our application, the performance of the priority queue developed matches, or even outperforms, that of the related priority queues presented in [8, 10].

Recall that in the regular system a positive integer n is represented as a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of digits, where

- $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and $d_{\ell-1} \neq 0$, and
- if $d_i = 2$ and $d_k = 2$ and $i < k$, then $d_j = 0$ for some $j \in \{i + 1, \ldots, k - 1\}$ *(regularity condition)*.

Traditionally, the regular system employs perfect weights, i.e. the weight of digit d_i is 2^i. We, however, adapt the system to skew weights. Therefore, as in the other skew binary systems, we have

- $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and
- the value of n is $\sum_{i=0}^{\ell-1} d_i w_i$.

Moreover, we associate a variable b_i with every digit d_i. This variable expresses the amount of work left until the change at that digit is completed.

In a computer realization, we rely on a sparse representation that keeps all non-zero digits together with their positions and variables in a doubly-linked list in sorted order according to the positions. In addition, we maintain a doubly-linked list for the positions of the positive b_i's and another for the positions of the 2's.

5.1 Increments

To perform an increment, it would be natural to use the surplus unit supplied by the increment for transferring the first $d_j = 2$ (if it exists) to d_{j+1}, by setting $d_j \leftarrow 0$ and increasing d_{j+1} by one. Such a *transfer* is exactly what is done in the canonical skew system. One complication that arises in our application is that the transfer of d_j may have $O(j)$ worst-case cost. To handle expensive digit changes, the key idea is to delay the work to be done in an expensive digit change and leave the delayed work for the upcoming operations (increments, decrements, and additions). Hence, we perform digit changes incrementally and keep track of the digits that are under an incremental change. Because of these incremental changes, we have to allow more than one 2 in the representation.

We divide the work to be done in an expensive digit change into discrete units, called *bricks*. When a digit d_k is under an incremental change, i.e. when $b_k > 0$, d_k is said to form a *wall* of b_k bricks. Decreasing b_k by one means that we perform a constant amount of work in order to bring the incremental change at d_k forward, which can be viewed as removing a brick from that wall.

The increment algorithm works as follows. Let d_j be the first 2 for which $b_j = 0$. If j is smaller than the position of the first wall, we transfer d_j to d_{j+1}; a value of $j + 1$ bricks is associated with b_{j+1}, and one brick is immediately removed from d_{j+1} leaving j bricks to be removed by the upcoming operations. Otherwise, d_0 is increased by one, and one brick is removed from the first wall if any. For this case, since $d_0 \in \{0, 1\}$ before the increment, $d_0 \in \{1, 2\}$ after the increment. For both cases, the value of the number is increased by one.

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

```
1: assert d_j = 2
2: d_j ← 0
3: increase d_{j+1} by 1
4: b_{j+1} ← j + 1
```

Remark 15. Since $w_{i+1} = 2w_i + 1$, a transfer increases the value of the represented number by one. □

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

```
1: let d_k be the first digit for which b_k > 0, if it exists
2: let d_j be the first 2 for which b_j = 0, if it exists
3: if d_j exists and (d_k does not exist or j < k)
4:    transfer(⟨d_0, d_1, ..., d_{ℓ−1}⟩, j)
5:    reduce b_{j+1} by 1
6: else
7:    increase d_0 by 1
8:    if d_k exists
9:       reduce b_k by 1
```

position i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
critical 0					d_1		d_3		d_5			d_7				
digit d_i	2	0	1	0	$\boxed{1}$	0	$\boxed{1}$	0	$\boxed{1}$	1	0	$\boxed{2}$	1	1	0	1
variable b_i					1		4		6			8				

Fig. 2. The representation obtained by performing 100 000 increments in the regular skew system starting from 0. All the walls are drawn in boxes.

Remark 16. Starting from 0 and performing 30 increments, the strings generated are: 1, 2, 01, 11, 21, 02, 00$\boxed{1}$, 101, 201, 011, 111, 211, 021, 00$\boxed{2}$, 102, 100$\boxed{1}$, 200$\boxed{1}$, 010$\boxed{1}$, 1101, 2101, 0201, 00$\boxed{1}$1, 1011, 2011, 0111, 1111, 2111, 0211, 00$\boxed{2}$1, and 1021. All walls are drawn in boxes. □

Remark 17. In the algorithm described in [8], several units of leftover work are to be performed per increment. Interestingly, in our case, it is enough to perform only one unit of leftover work per operation. This stems from the fact that we use skew weights instead of perfect weights. □

To prove the correctness of our algorithms, we have to rely on stronger regularity conditions than those of the standard regular system. Our first invariant states that every 2 is immediately preceded by a 0, while it is possible to have $d_0 = 2$. The second invariant indicates that between every two 2's there are at least two 0's. The third invariant states that every wall is immediately preceded by a 0. The fourth invariant indicates that if a 2 is followed by a wall, there are at least two 0's between them. The fifth invariant indicates that every wall has at least two preceding 0's. Assume that d_k is a wall. Let d_j be a 0, such that $j < k-1$ and $d_{j'} \neq 0$ for all $j' \in \{j+1, \ldots, k-2\}$. In other words, d_j is the 0 with the highest position that precedes the 0 at d_{k-1}. By the fifth invariant, such a 0 always exists. We call this 0 the *critical* 0 for the wall d_k. The sixth invariant states that the number of leftover bricks at a wall is not larger than $j+1$, where j is the position of the critical 0 for that wall, and even not larger than j if this critical 0 is immediately preceded by a substring of the form 21^*.

More formally, the invariants maintained are:

(1) If $d_j = 2$ and $j > 0$, then $d_{j-1} = 0$.
(2) If $d_j = 2$, $d_k = 2$, and $j < k$, then $d_{j'} = 0$ for some $j' \in \{j+1, \ldots, k-2\}$.
(3) If d_k is a wall, then $d_{k-1} = 0$.
(4) If $d_j = 2$, d_k is a wall, and $j < k$, then $d_{j'} = 0$ for some $j' \in \{j+1, \ldots, k-2\}$.
(5) If d_k is a wall, then $d_i = 0$ for some $i \in \{0, \ldots, k-2\}$.
(6) Let d_j be the critical 0 for a wall d_k. Then
    (i) $b_k \leq j+1$; moreover
    (ii) $b_k \leq j$, if $d_i = 2$ and $d_{i'} = 1$ for all $i' \in \{i+1, \ldots, j-1\}$.

Remark 18. Our intuition about the wall-breaking process is the following (for a snapshot of this dynamic process, see Fig. 2):

- The first wall is *visible* for an increment if all the digits preceding that wall are 0's and 1's. When a wall is visible, this particular increment will reduce the number of bricks at that wall by 1.
- The configuration of the digits preceding a wall affects the number of bricks remaining at that wall. In particular, the position of the critical 0, and whether the critical 0 is immediately preceded by a substring of the form 21^* or not, is used to bound the height of the wall. In accordance, this prevents that two walls become adjacent.
- During the counting process, there will be enough configurations for every wall being visible for increments. Specifically, when—or even before—the critical 0 for the first wall disappears (all the digits preceding the first wall, except its immediately preceding 0, are 1's or 2's), there are no bricks left at this wall, i.e. the wall has been dismantled. □

The aforementioned stronger regularity conditions imply that in any configuration a digit is either $0, 1$, or 2, and that no walls become adjacent. In addition, our algorithms will never transfer a digit that is a 2 if it also forms a wall, i.e. no walls are involved in creating other walls. Next, we show that the invariants are retained after every increment if they were fulfilled before the operation; the correctness of *increment* accordingly follows.

Consider the case where a digit d_j is the first digit that equals 2, is not a wall, precedes any wall, and $j \neq 0$. In such a case, a transfer is initiated at d_j. Using invariant (1), $d_{j-1} = 0$. After the operation, $d_j = 0$ and d_{j+1} becomes a wall with $b_{j+1} = j$. At this point, d_{j-1} is the critical 0 for the wall d_{j+1}; this fulfils all the invariants for this wall. A special case is when $j = 0$. For such a case, the created wall d_1 is immediately dismantled. Consider a digit $d_k = 2$ that follows d_j. Using invariants (1) and (2), before the increment, there are two 0's, one at d_{k-1} and the other has a position in $\{j + 1, \ldots, k - 2\}$. Whether d_{j+1} was 0 or 1 before the operation, it is straightforward to verify that all the invariants are fulfilled for d_k after the operation. The interesting case is when a digit $d_k \in \{1, 2\}$ is a wall and d_{j+1} was the critical 0 for this wall before the operation. For such a case, using invariant (6.ii) as $d_j = 2$, we have $b_k \leq j + 1$. After the operation, the critical 0 for the wall d_k becomes d_j. However, the validity of the invariants still hold as the weaker bound of invariant (6.i) applies and is fulfilled. It is straightforward to verify that all the other invariants are also retained.

Consider the case where d_k is the first wall and that it precedes any 2. In such a case, one brick is removed from the wall d_k and d_0 is increased by 1. Using invariants (3) and (5), $d_{k-1} = 0$ and $d_j = 0$ for some $j \in \{0, \ldots, k - 2\}$. An interesting case is when, before the increment, d_i was 1 for all $i \in \{0, \ldots, j - 1\}$, and d_j was the critical 0 for the wall d_k. For such a case, using invariant (6.i), $b_k \leq j+1$. After the operation, as d_0 becomes 2, the stronger bound of invariant (6.ii) applies and is fulfilled following the brick removal from d_k, i.e. $b_k \leq j$. Another interesting case is when, before the increment, d_0 was the critical 0 for the wall d_k. For such a case, using invariant (6.i), $b_k \leq 1$. After the operation,

d_0 becomes 1, and as a result only one 0 precedes d_k. Fortunately, one brick must have been removed from the wall d_k, dismantling the wall and saving our invariants just on time. It is straightforward to verify that, even in the trivial cases omitted, the invariants are also retained.

5.2 Decrements

In the regular skew system, we carry out decrements in the same way as in the canonical skew system. To ensure that the amount of work left at the walls satisfies our invariants, if the first non-zero digit was a wall, the wall is moved to the preceding position and a brick is removed from it. The details are specified in the following pseudo-code:

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

```
1: assert ⟨d_0, d_1, ..., d_{ℓ-1}⟩ is not empty
2: let d_j be the first non-zero digit
3: if b_j > 0
4:    reduce b_j by 1
5: decrease d_j by 1
6: if j ≠ 0
7:    d_{j-1} ← 2;   b_{j-1} ← b_j;   b_j ← 0
```

Let d_j be the first non-zero digit before a decrement. First, consider the case where $j = 0$, i.e. d_0 was 1 or 2 before the operation. After the operation, d_0 is decreased by one. It is straightforward to verify that all the invariants are retained after the operation. Next, assume that $j \neq 0$. Consider the case where d_j was a wall before the operation. By invariant (6), $b_j \leq j - 1$ was valid before the operation. Due to the brick removal, $b_{j-1} \leq j - 2$ after the operation, and invariant (6) is retained. Consider the case where d_j was equal to 1 before the operation, and d_k was the following wall whose critical 0 was d_{j-1}. (The case where the critical 0 for d_k comes after d_j is trivial.) Using invariant (6.i), $b_k \leq j$ before the operation. After the operation, d_j becomes the critical 0 for the wall d_k, and d_{j-1} becomes 2. In accordance, invariant (6.ii) applies for b_k and is fulfilled. Alternatively, if d_j was equal to 2 before the operation, the critical 0 for the following wall d_k must come after d_j, and invariant (6) trivially holds after the operation. It is also clear that, whether d_j was a wall or not, and whether it was equal to 1 or 2, all the other invariants are retained.

5.3 Increments at Arbitrary Positions

Given a string $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$, our objective is to efficiently increase an arbitrary digit d_i by one. Compared to a normal increment, this general increment involves more new cases which make the algorithm a bit more complicated. Also,

the worst-case complexity of the operation becomes higher, namely $O(i)$, even though we still solve the special case $i = 0$ at $O(1)$ worst-case cost.

A crucial difference between *increment* and *arbitrary-increment* is that, if $i > 0$, we have no surplus unit to transfer a digit forward. Accordingly, we always have to add one to digit d_i. To preserve regularity, we may need to perform up to two digit transfers. To create the surplus needed for digit transfers, we perform two decrement operations, and use the resulting surplus for such transfers. Since we do not know in advance the number of the transfers that we perform, at the end, we perform one *increment* for each surplus we have saved but not used. The following corrective actions are necessary to reestablish our invariants.

If d_i is a wall, we dismantle this wall by removing all (at most $i - 1$) of its bricks. If $d_i = 2$, we transfer d_i one position forward. Hereafter, we can assume that $d_i \in \{0, 1\}$. At this point, let d_j be the first 2 for which $j > i$ and $b_j = 0$. It is time now that we increase d_i by one. If after increasing d_i its value becomes 2, we transfer the created 2 one position forward. Note that at most one of these aforementioned two transfers is executed. If d_{i+1} is a wall, we dismantle this wall by removing all (at most i) of its bricks. This wall is either an original wall even before the operation or has just been created by transferring a 2 from d_i forward. At this point, let d_k be the first wall for which $k > i$. Afterwards, we perform the following steps that are quite similar to the increment algorithm of Section 5.1. If j is smaller than k, we transfer the 2 at d_j one position forward, and remove the minimum of $i + 2$ and all the bricks from d_{j+1}. If k is smaller than j, we remove the minimum of $i + 1$ and all the bricks from d_k.

To sum up, at most a constant number of digits are changed in each increment. To find the walls and 2's operated on, it may be necessary to scan the linked lists, starting from the beginning, to the desired positions. To get there, at most $O(i)$ list items have to be visited. Also, the number of bricks removed at each step is $O(i)$. We conclude that the complexity of the algorithm is $O(i)$ in the worst case. This completes the description of the algorithm. All actions are summarized in the enclosed pseudo-code.

To prove the correctness of the algorithm, we shall show that all the invariants hold for every wall and for every 2 when considering the preceding digits for each. For the purpose of our analyses, we split the pseudo-code into *phase I* constituting lines 5 to 16, and *phase II* constituting lines 17 to 23. Consider the effect of executing phase I: only d_i and d_{i+1} may change, d_i will neither be a 2 nor a wall, and d_{i+1} will not be a wall. There are three cases which result in d_{i+1} being a 2 after phase I.

- d_i was 1 and d_{i+1} was 1: After phase I, d_i becomes a 0.
- d_i was 2 and d_{i+1} was 1: After phase I, d_i becomes a 1. In such case, $j = i+1$.
- d_i was 0 and d_{i+1} was 2: After phase I, d_i becomes a 1. In such case, $j = i+1$.

For the first case, the invariants are directly fulfilled for d_{i+1} after phase I. For the last two cases, we have to wait until phase II, when the 2 at d_{i+1} is transferred to d_{i+2}, for the invariants to hold.

In general, if $j > i + 1$, phase II of the algorithm is pretty similar to the increment algorithm of Section 5.1. The only differences are the positions of the

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

assert the value of $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ is larger than 2
repeat 2 times
    decrement$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$
surplus $\leftarrow 2$
if $b_i > 0$
    reduce b_i to 0
if $d_i = 2$
    transfer$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$
    decrease *surplus* by 1
let d_j be the first 2 for which $j > i$ and $b_j = 0$, if it exists
increase d_i by 1
if $d_i = 2$
    transfer$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$
    decrease *surplus* by 1
if $b_{i+1} > 0$
    reduce b_{i+1} to 0
let d_k be the first digit for which $k > i$ and $b_k > 0$, if it exists
if d_j exists **and** (d_k does not exist **or** $j < k$)
    transfer$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$
    decrease *surplus* by 1
    reduce b_{j+1} by $\min\{b_{j+1}, i+2\}$
else if d_k exists
    reduce b_k by $\min\{b_k, i+1\}$
repeat *surplus* times
    increment$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

critical 0's, and the number of bricks we remove from the walls. The same proof of Section 5.1 implies the validity of the invariants for all the 2's. What is left to be shown is that invariant (6) is also satisfied for all the walls.

Consider the first wall d_k for which $i < k < j$. If the critical 0 for d_k was following d_{i+1}, then it does not change after phase I, and hence all the invariants hold for d_k. If the critical 0 for d_k was preceding d_{i+1}, then $b_k \leq i+1$. This means that by removing $i+1$ bricks this wall is dismantled. If the critical 0 for d_k was at d_{i+1} and d_i was a 2, then $b_k \leq i+1$, and the wall is also dismantled. If the critical 0 for d_k was at d_{i+1} and d_i was not a 2, then the critical 0 for d_k becomes at d_i or is still at d_{i+1}. Either way, invariant (6) still holds.

Back to the case where $j = i+1$. When the 2 at d_{i+1} is transferred in phase II, it results in a wall at d_{i+2}. However, we instantaneously dismantle this wall by removing its $i+2$ bricks. Alternatively, if $i+1 < j < k$, then d_{j-1} or d_{j-2} is the critical 0 for the resulting wall at d_{j+1}. In this case, b_{j+1} is initiated with $j+1$, which is followed by removing $i+2$ bricks (we only need to remove one or two bricks in this case). Accordingly, invariant (6) still holds.

5.4 Additions

When an increment at any position can be carried out efficiently, a simple way of implementing *addition* is to apply *arbitrary-increment* repeatedly. For every digit d_i of the shorter string, starting from the least-significant digit, increment the ith digit of the longer string d_i times. The correctness of this approach directly follows from the correctness of *arbitrary-increment*.

Remark 19. An addition will destroy the longer of the two input strings. □

Algorithm *addition*($\langle d_0, d_1, \ldots, d_{k-1}\rangle$, $\langle e_0, e_1, \ldots, e_{\ell-1}\rangle$)

```
1: assert k ≤ ℓ
2: for i ∈ {0, 1, ..., k − 1}
3:    repeat d_i times
4:       if b_i > 0
5:          reduce b_i to 0
6:       arbitrary-increment(⟨e_0, e_1, ..., e_{ℓ−1}⟩, i)
7: return ⟨e_0, e_1, ..., e_{ℓ−1}⟩
```

Assume that the representations of the given numbers are of length k and ℓ, and that $k \leq \ell$. By Lemma 4 (see Section 5.5), the sum of the digits in the shorter representation is at most $k + 1$. So, an addition of the two numbers involves at most $k+1$ arbitrary increments. Each such increment only changes a constant number of digits. Hence, the total number of digit changes performed is proportional to the length of the shorter representation. At position i, both the wall dismantling and the arbitrary increment have $O(i)$ worst-case cost. Thus, the worst-case cost of *addition* is $O(k^2)$.

5.5 Properties

The following lemma directly follows from the invariants maintained by the operations.

Lemma 4. *Let $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ be a string of digits in the regular skew system. The sum of the digits is at most $\ell + 1$, i.e. $\sum_{i=0}^{\ell-1} d_i \leq \ell + 1$.*

We can summarize the results proved for the regular skew system as follows.

Theorem 4. *The regular skew system supports increments and decrements at $O(1)$ worst-case cost, and additions at $O(k^2)$ worst-case cost, where k denotes the length of the shorter of the two strings added. The number of digit changes involved is at most two for increments and decrements, and $O(k)$ for additions. The amount of space needed for representing a positive integer n while supporting these modifications is $O(\lg n)$.*

6 Application Revisited

As earlier, we implement a priority queue as a forest of perfect binary heaps, and use a numeral system to control the number and size of the heaps in the structure. When the canonical skew system is in use, *insert* has logarithmic worst-case cost, whereas *borrow* has $O(1)$ worst-case cost. For the magical skew system, the opposite is true: *insert* has $O(1)$ worst-case cost and *borrow* has logarithmic worst-case cost. In both cases, the reason for the logarithmic behaviour is that, even though only a constant number of digit changes is made per operation, it may be necessary to perform a *siftdown* operation for a large heap, and such operation can be costly. In order to avoid this problem, we rely on the regular skew system which allows incremental execution of *siftdown* operations.

In our implementation the basic components are: perfect binary heaps, an auxiliary structure resembling the regular skew system keeping track of the heaps, and a minimum pointer identifying the location of the current minimum. The auxiliary structure comprises the data recording the digits, the positions of the walls and 2's, and the heights of the walls. The digit at position i denotes the number of perfect binary heaps of size $2^{i+1} - 1$. Each digit should be associated with a list of items referring to heaps of that particular size; every such list has at most two items, and each item stores a pointer to the root of a heap. In a brick removal, a *siftdown* process is advanced one level down a heap, or nothing is done if the heap order has already been reestablished. To facilitate incremental execution of *siftdown* operations, each of the items associated with a digit also stores a pointer to the node up to which the corresponding *siftdown* has proceeded; this pointer may be null indicating that there is no ongoing *siftdown* process. When this pointer is available, the process can be easily advanced, level by level, until the bottom level is reached or the heap order is reestablished. In addition, the path traversed by every ongoing *siftdown* is memorized by storing the path traversed so far in binary form; we call this a *bit trace*. A 0-bit (1-bit) at the index i of a bit trace reflects that the process continued in the direction of the left (right) child at level i.

Remark 20. A suitable realization of a bit trace would be a single computer word. Using logical bitwise operations and bit shifts, the bit at a specified index can be easily fetched and updated at $O(1)$ worst-case cost. This accounts for $O(\lg n)$ words to be stored, at most one per heap. □

In *insert*, the actions specified for an increment in the numeral system are emulated. There are two possibilities to consider. The first possibility is that the given node is used to combine two heaps, by making it the root of the combined heap and making the roots of the two heaps the children of that root. A *siftdown* process is initiated at the new root and advanced one level downwards. This maintains the invariant that the root of any heap contains a minimum element among those in that heap. The second possibility is that the given node is inserted into the structure as a heap of size one. For such a case, the *siftdown* process at the first wall (if any) is advanced one level downwards.

When a *siftdown* is advanced, the direction to which the process proceeds is recorded in the bit trace, by updating the bit corresponding to the current level. Finally, the minimum pointer is updated if necessary. The worst-case cost of *insert* is $O(1)$, and at most three element comparisons are performed (two in *siftdown* and one to check whether the minimum pointer is up to date or not).

In *borrow*, the actions specified for a decrement in the numeral system are emulated. The numeral system dictates that the smallest heap is selected for splitting. If the first non-zero digit is a 2 that forms a wall, of the two candidates, the heap that has an ongoing *siftdown* process is selected. First, the ongoing *siftdown* (if any) in the selected heap is advanced one level downwards. Normally, the selected heap is split by detaching and returning its root, and moving the heaps rooted at the children of the root (if any) to the preceding position. A special case is when the root of the selected heap is associated with the minimum of the priority queue. In such a case, this root is swapped either with another root or with one of its two children, which is then detached and returned. A removal of the root, or a change in one of its children, will not affect the *siftdown* process possibly being in progress further down in the heap. Lastly, if the selected heap had an ongoing *siftdown*, it should be possible to identify which of the two heaps owns that process after the split. Here we need the bit trace. The existing bit trace is updated to correspond to the root removal and is assigned to the new owner. Clearly, the worst-case cost of *borrow* is $O(1)$, and at most two element comparisons are performed.

In accordance, *delete* can use this kind of borrowing and reestablish the heap order by sifting the replacement node up or down. Naturally, the minimum pointer has to be updated if the current minimum is being deleted. Compared to our earlier solution, the only difference is that the affected heap may have an incremental *siftdown* process in progress. We let *delete* finish this process before handling the actual deletion. As before, the worst-case cost of *delete* is $O(\lg n)$. By Lemma 4, the number of perfect binary heaps is bounded by $\lg n + O(1)$. Therefore, in the worst case, *delete* involves at most $5 \lg n + O(1)$ element comparisons as a result of performing at most two *siftdown* operations and one possible scan over the roots, or one *siftup* and one possible *siftdown*.

Remark 21. It is possible to reduce the number of element comparisons performed by *delete* to at most $3 \lg n + O(1)$. The worst-case scenario occurs when a *siftdown* operation is to be performed, while another *siftdown* is in progress. In this case, the ongoing *siftdown* is first completed, but the second *siftdown* is performed only partially, leaving the wall with the same height as before the operation. Then, the number of element comparisons performed is at most $3 \lg n + O(1)$; the two *siftdown* operations combined account for $2 \lg n + O(1)$, and the scan over the roots accounts for $\lg n + O(1)$. □

In *meld*, the actions specified for an addition in the numeral system are emulated. A transfer of two perfect binary heaps forward, using a surplus node borrowed at the beginning of the addition, is done in the same manner as in *insert*. Similarly, *siftdown* processes can be advanced as in *insert*. The only difference

is that now several bricks are processed, instead of just one. The performance of *meld* directly follows from the performance of the addition operation. Given two priority queues of sizes m and n, $m \leq n$, the worst-case cost of *meld* is $O(\lg^2 m)$.

Remark 22. The space optimization mentioned in Remark 13 is also applicable here. Hence, the amount of extra space can be reduced to $2n+O(\lg n)$ words. □

The efficiency of the operations and the representation can be summarized as follows.

Theorem 5. *A meldable priority queue that maintains an ordered collection of perfect binary heaps guarantees $O(1)$ worst-case cost per insert and borrow, $O(\lg n)$ worst-case cost per delete, and $O(\lg^2 m)$ worst-case cost per meld, where n is the number of elements in the priority queue manipulated by delete and m is the number of elements in the smaller of the two priority queues manipulated by meld. A priority queue storing n elements requires $2n + O(\lg n)$ words, in addition to the space used by the elements themselves.*

7 Conclusions

All the numeral systems discussed in this paper—canonical skew, magical skew, and regular skew—support increments and decrements at $O(1)$ worst-case cost, and involve at most a constant number of digit changes per operation. In spite of this, the efficiency of the data structures using the numeral systems varied depending on the carry-propagation and borrowing mechanisms used. Hence, to understand the actual behaviour in our application, a deeper investigation into the properties of the numeral systems was necessary.

For the magical skew system, when proving the correctness of our algorithms, we used an automaton. The state space of the modelling automaton was small, and the states can be seen as forming an orbit. Increments were proved to move along the states around this orbit. The system turned out to be extremely sensitive. Therefore, we decided to implement decrements as the reverse of increments. In the conference version of this paper [1], we relied on a general undo-logging technique to implement decrements as the reverse of increments. This, however, has the drawback that the amount of space used is proportional to the value of the underlying number, which is exponential in the length of the representation. In this version of the paper, we showed that the reversal approach works even if the amount of space available is proportional to the length of the representation.

One drawback of the magical skew system is that it does not support additions efficiently. Simply, it is not known how to efficiently get from an arbitrary configuration back to one of the configurations on the orbit. The invariants enforced by the canonical skew and regular skew systems are more relaxed, so additions can be efficiently realized. The regular skew system supports additions even more efficiently than the canonical skew system ($O(k)$ versus $O(\ell)$ digit changes, where k and ℓ are the lengths of the added strings and $k \leq \ell$).

By implementing a priority queue as a forest of perfect binary heaps, we provided two data structures that perform *insert* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost. There are several other ways of achieving the same bounds; for example, one could use specialized data structures like a forest of pennants [8] or a binomial queue [9, 13, 14]. The priority queue obtained using the regular skew system supports *borrow* at $O(1)$ worst-case cost and *meld* at $O(\lg^2 m)$ worst-case cost, where m is the number of elements stored in the smaller of the two melded priority queues.

We conclude the paper with some open problems.

1. The array-based implementation of a binary heap supports *delete* with at most $2 \lg n$ element comparisons (or even less [7]). To perform *insert* at $O(1)$ worst-case cost, we increased the number of element comparisons performed by *delete* to $6 \lg n$ (Section 4) or $3 \lg n$ (Section 6). How to reduce the number of element comparisons performed by *delete* without making other operations asymptotically slower?
2. We showed that all the following operations: *insert*, *meld*, and *delete* could be performed at optimal asymptotic cost in the amortized sense. However, our best data structure only supports *meld* at $O(\lg^2 m)$ worst-case cost, where m is the number of elements stored in the smaller of the two melded priority queues. In [9], the technique of data-structural bootstrapping was used to speed up *meld* for (skew) binomial queues. In our case, the worst-case cost of *meld* is still too large to apply this technique successfully. Can the worst-case performance of *meld* be improved?
3. All known approaches to implement fast *decrease*, which decreases the value stored at a given node, are based on heap-ordered binomial-tree-like structures. Is it really so that binary heaps are not competitive in this respect?
4. So far, we have not attempted to provide any evidence for the practical relevance of the findings reported in this paper. How efficient are the described data structures in practice?
5. We hope that our approach for designing numeral systems and related data structures is relevant in other contexts. Are there other applications that could be tackled with the developed numeral systems?

Acknowledgment

We thank the anonymous referee who motivated us to extend this version of the paper beyond the frontiers of the conference version.

References

1. Elmasry, A., Jensen, C., Katajainen, J.: The magic of a number system. In: Proceedings of the 5th International Conference on Fun with Algorithms. Volume 6099 of Lecture Notes in Computer Science, Springer-Verlag (2010) 156–165
2. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM **21**(4) (1978) 309–315

3. Clancy, M., Knuth, D.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University (1977)
4. Myers, E.W.: An applicative random-access stack. Information Processing Letters **17**(5) (1983) 241–248
5. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
6. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM **7**(6) (1964) 347–348
7. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM Journal on Computing **15**(4) (1986) 964–971
8. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Proceedings of the 1st Scandinavian Workshop on Algorithm Theory. Volume 318 of Lecture Notes in Computer Science, Springer-Verlag (1988) 1–13
9. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. Journal of Functional Programming **6**(6) (1996) 839–857
10. Sack, J.R., Strothotte, T.: A characterization of heaps and its applications. Information and Computation **86**(1) (1990) 69–86
11. Bansal, S., Sreekanth, S., Gupta, P.: M-heap: A modified heap data structure. International Journal of Foundations of Computer Science **14**(3) (2003) 491–502
12. Harvey, N.J.A., Zatloukal, K.: The post-order heap. In: Proceedings of the 3rd International Conference on Fun with Algorithms. (2004)
13. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms **5**(1) (2008) 14:1–14:19
14. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Informatica **45**(3) (2008) 193–210
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 3nd edn. The MIT Press, Cambridge (2009)
16. Brown, M.R.: Implementation and analysis of binomial queue algorithms. SIAM Journal on Computing **7**(3) (1978) 298–319
17. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3) (1987) 596–615