Strictly-Regular Number System and Data Structures[⋆]

Amr Elmasry[1], Claus Jensen[2], and Jyrki Katajainen[3]

[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 we call the strictly-regular system, which efficiently supports the operations: digit-increment, digit-decrement, cut, concatenate, and add. Compared to other number systems, the strictly-regular system has distinguishable properties. It is superior to the regular system for its efficient support to decrements, and superior to the extended-regular system for being more compact by using three symbols instead of four. To demonstrate the applicability of the new number system, we modify Brodal's meldable priority queues making deletion require at most $2 \lg n + O(1)$ element comparisons (improving the bound from $7 \lg n + O(1)$) while maintaining the efficiency and the asymptotic time bounds for all operations.

1 Introduction

Number systems are powerful tools of the trade when designing worst-case-efficient data structures. As far as we know, their usage was first discussed in the seminar notes by Clancy and Knuth [1]. Early examples of data structures relying on number systems include finger search trees [2] and binomial queues [3]. For a survey, see [4, Chapter 9]. The problem with the normal binary number representation is that a single increment or decrement may change all the digits in the original representation. In the corresponding data structure, this may give rise to many changes that would result in weak worst-case performance.

The characteristics of a positional number system $\mathcal{N}$ are determined by the constraints imposed on the digits and the weights corresponding to them. Let $rep(d, \mathcal{N}) = \langle d_0, d_1, \ldots, d_{r-1} \rangle$ be the sequence of digits representing a positive integer d in $\mathcal{N}$. (An empty sequence can be used to represent zero.) By convention, d_0 is the least-significant digit and $d_{r-1} \neq 0$ is the most-significant digit.

[⋆] © 2010 Springer-Verlag. This is the authors' version of the work. The original publication is available at `www.springerlink.com` with DOI `10.1007/978-3-642-13731-0_4`.

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.

The value of d in $\mathcal{N}$ is $val(d, \mathcal{N}) = \sum_{i=0}^{r-1} d_i \cdot w_i$, where w_i is the weight corresponding to d_i. As a shorthand, we write $rep(d)$ for $rep(d, \mathcal{N})$ and $val(d)$ for $val(d, \mathcal{N})$. In a redundant number system, it is possible to have $val(d) = val(d')$ while $rep(d) \neq rep(d')$. In a b-ary number system, $w_i = b^i$.

A sequence of digits is said to be *valid* in $\mathcal{N}$ if all the constraints imposed by $\mathcal{N}$ are satisfied. Let d and d' be two numbers where $rep(d) = \langle d_0, d_1, \ldots, d_{r-1} \rangle$ and $rep(d') = \left\langle d'_0, d'_1, \ldots, d'_{r'-1} \right\rangle$ are valid. The following operations are defined.

$increment(d, i)$: Assert that $i \in \{0, 1, \ldots, r\}$. Perform ++$d_i$ resulting in d', i.e. $val(d') = val(d) + w_i$. Make d' valid without changing its value.

$decrement(d, i)$: Assert that $i \in \{0, 1, \ldots, r-1\}$. Perform --$d_i$ resulting in d', i.e. $val(d') = val(d) - w_i$. Make d' valid without changing its value.

$cut(d, i)$: Cut $rep(d)$ into two valid sequences having the same value as the numbers corresponding to $\langle d_0, d_1, \ldots, d_{i-1} \rangle$ and $\langle d_i, d_{i+1}, \ldots, d_{r-1} \rangle$.

$concatenate(d, d')$: Concatenate $rep(d)$ and $rep(d')$ into one valid sequence that has the same value as $\left\langle d_0, d_1, \ldots, d_{r-1}, d'_0, d'_1, \ldots, d'_{r'-1} \right\rangle$.

$add(d, d')$: Construct a valid sequence d'' such that $val(d'') = val(d) + val(d')$.

One should think that a corresponding data structure contains d_i components of rank i, where the meaning of rank is application specific. A component of rank i has size $s_i \leq w_i$. If $s_i = w_i$, we see the component as perfect. In general, the size of a structure corresponding to a sequence of digits need not be unique.

The regular system [1], called the segmented system in [4], comprises the digits $\{0, 1, 2\}$ with the constraint that every 2 is preceded by a 0 possibly having any number of 1's in between. Using the syntax for regular expressions (see, for example, [5, Section 3.3]), every regular sequence is of the form $\left(0 \mid 1 \mid 01^*2\right)^*$. The regular system allows for the increment of any digit with $O(1)$ digit changes [1, 6], a fact that can be used to modify binomial queues to accomplish *insert* at $O(1)$ worst-case cost. Brodal [7] used a zeroless variant of the regular system, comprising the digits $\{1, 2, 3\}$, to ensure that the sizes of his trees are exponential with respect to their ranks. For further examples of structures that use the regular system, see [8, 9]. To be able to perform decrements with $O(1)$ digit changes, an extension was proposed in [1, 6]. Such an extended-regular system comprises the digits $\{0, 1, 2, 3\}$ with the constraint that every 3 is preceded by a 0 or 1 possibly having any number of 2's in between, and that every 0 is preceded by a 2 or 3 possibly having any number of 1's in between. For examples of structures that use the extended-regular system, see [6, 10, 11].

In this paper, we introduce a number system that we call the strictly-regular system. It uses the digits $\{0, 1, 2\}$ and allows for both increments and decrements with $O(1)$ digit changes. The strictly-regular system contains less redundancy and is more compact, achieving better constant factors while supporting a larger repertoire of operations. We expect the new system to be useful in several other contexts in addition to the applications we mention here.

Utilizing the strictly-regular system, we introduce the strictly-regular trees. Such trees provide efficient support for adding a new subtree to the root, detaching an existing one, cutting and concatenating lists of children. We show that

Table 1: Known results on the worst-case comparison complexity of priority-queue operations when *decrease* is not considered and *find-min* has $O(1)$ cost. Here n and m denote the sizes of priority queues.

Source	*insert*	*delete*	*meld*
[12]	$O(1)$	$\lg n + O(1)$	$-$
[11]	$O(1)$	$\lg n + O(\lg \lg n)$	$O(\lg(\min\{n, m\}))$
[7] (see Section 3.1)	$O(1)$	$7 \lg n + O(1)$	$O(1)$
[13]	$O(1)$	$3 \lg n + O(1)$	$O(1)$
this paper	$O(1)$	$2 \lg n + O(1)$	$O(1)$

the number of children of any node in a strictly-regular tree is bounded by $\lg n$, where n is the number of descendants of such node.

A *priority queue* is a fundamental data structure which stores a dynamic collection of elements and efficiently supports the operations *find-min*, *insert*, and *delete*. A *meldable* priority queue also supports the operation *meld* efficiently. As a principal application of our number system, we implement an efficient meldable priority queue. Our best upper bound is $2 \lg n + O(1)$ element comparisons per *delete*, which is achieved by modifying the priority queue described in [7]. Table 1 summarizes the related known results.

The paper is organized as follows. We introduce the number system in Section 2, study the application to meldable priority queues in Section 3, and discuss the applicability of the number system to other data structures in Section 4.

2 The Number System

Similar to the redundant binary system, in our system any digit d_i must be 0, 1, or 2. We call 0 and 2 *extreme digits*. We say that the representation is *strictly regular* if the sequence from the least-significant to the most-significant digit is of the form $\left(1^+ \mid 01^*2\right)^* \left(\varepsilon \mid 01^+\right)$. In other words, such a sequence is a combination of zero or more interleaved 1^+ and 01^*2 blocks, which may be followed by at most one 01^+ block. We use $w_i = 2^i$, implying that the weighted value of a 2 at position i is equivalent to that of a 1 at position $i+1$.

2.1 Properties

An important property that distinguishes our number system from other systems is what we call the *compactness* property, which is defined in the next lemma.

Lemma 1. *For any strictly-regular sequence, $\sum_{i=0}^{r-1} d_i$ is either $r-1$ or r.*

Proof. The sum of the digits in a 01^*2 block or a 1^* block equals the number of digits in the block, and the sum of the digits in the possibly trailing 01^+ block is one less than the number of digits in that block. □

Note that the sum of digits $\sum_{i=0}^{r-1} d_i$ for a positive integer in the regular system is between 1 and r; in the zeroless system, where $d_i \in \{1, 2, \ldots h\}$, the sum of digits is between r and $h \cdot r$; and in the zeroless regular representation, where $d_i \in \{1, 2, 3\}$ [7], the sum of digits is between r and $2r$.

An important property, essential for designing data structures with exponential size in terms of their rank, is what we call the *exponentiality* property. Assume $s_i \geq \theta^i/c$ and $s_0 = 1$, for fixed real constants $\theta > 1$ and $c > 0$. A number system has such property if for each valid sequence $\sum_{i=0}^{r-1} d_i \cdot s_i \geq \theta^r/c - 1$ holds.

Lemma 2. *For the strictly-regular system, the exponentiality property holds by setting $\theta = c = \Phi$, where Φ is the golden ratio.*

Proof. Consider a sequence of digits in a strictly-regular representation, and think about $d_i = 2$ as two 1's at position i. It is straightforward to verify that there exists a distinct 1 whose position is at least i, for every i from 0 to $r - 2$. In other words, we have $\sum_{i=0}^{r-1} d_i \cdot s_i \geq \sum_{i=0}^{r-2} s_i$. Substituting with $s_i \geq \Phi^{i-1}$ and $s_0 = 1$, we obtain $\sum_{i=0}^{r-1} d_i \cdot s_i \geq 1 + \sum_{i=0}^{r-3} \Phi^i \geq \Phi^{r-1} - 1$. $\square$

The exponentiality property holds for any zeroless system by setting $\theta = 2$ and $c = 1$. The property also holds for any θ when $d_{r-1} \geq \theta$; this idea was used in [8], by imposing $d_{r-1} \geq 2$, to ensure that the size of a tree of rank r is at least 2^r. On the other hand, the property does not hold for the regular system.

2.2 Operations

It is convenient to use the following subroutines that change two digits but not the value of the underlying number.

fix-carry(d, i)**:** Assert that $d_i \geq 2$. Perform $d_i \leftarrow d_i - 2$ and $d_{i+1} \leftarrow d_{i+1} + 1$.
fix-borrow(d, i)**:** Assert that $d_i \leq 1$. Perform $d_{i+1} \leftarrow d_{i+1} - 1$ and $d_i \leftarrow d_i + 2$.

Temporarily, a digit can become a 3 due to ++d_i or *fix-borrow*, but we always eliminate such a violation before completing the operations. We demonstrate in Algorithm *increment* (*decrement*) how to implement the operation in question with at most one *fix-carry* (*fix-borrow*), which implies Theorem 1. The correctness of the algorithms follows from the case analysis of Table 2.

Theorem 1. *Given a strictly-regular representation of d, increment(d, i) and decrement(d, i) incur at most three digit changes.*

Algorithm *increment*(d, i)

```
1: ++$d_i$
2: Let $d_b$ be the first extreme digit before $d_i$, $d_b \in \{0, 2, undefined\}$
3: Let $d_a$ be the first extreme digit after $d_i$, $d_a \in \{0, 2, undefined\}$
4: if $d_i = 3$ or ($d_i = 2$ and $d_b \neq 0$)
5:     fix-carry$(d, i)$
6: else if $d_a = 2$
7:     fix-carry$(d, a)$
```

Algorithm *decrement*(d, i)

```
1: Let $d_b$ be the first extreme digit before $d_i$, $d_b \in \{0, 2, \mathit{undefined}\}$
2: Let $d_a$ be the first extreme digit after $d_i$, $d_a \in \{0, 2, \mathit{undefined}\}$
3: if $d_i = 0$ or ($d_i = 1$ and $d_b = 0$ and $i \neq r - 1$)
4:     fix-borrow$(d, i)$
5: else if $d_a = 0$
6:     fix-borrow$(d, a)$
7: --$d_i$
```

By maintaining pointers to all extreme digits in a circular doubly-linked list, the extreme digits are readily available when increments and decrements are carried out at either end of a sequence.

Corollary 1. *Let* $\langle d_0, d_1, \ldots, d_{r-1}\rangle$ *be a strictly-regular representation of* d*. If such sequence is implemented as two circular doubly-linked lists, one storing all the digits and another all extreme digits, any of the operations increment*$(d, 0)$*, increment*$(d, r-1)$*, increment*(d, r)*, decrement*$(d, 0)$*, and decrement*$(d, r-1)$ *can be executed at* $O(1)$ *worst-case cost.*

Theorem 2. *Let* $\langle d_0, d_1, \ldots, d_{r-1}\rangle$ *and* $\langle d'_0, d'_1, \ldots, d'_{r'-1}\rangle$ *be strictly-regular representations of* d *and* d'*. The operations cut*(d, i) *and concatenate*(d, d') *can be executed with* $O(1)$ *digit changes. Assuming without loss of generality that* $r \leq r'$*, add*(d, d') *can be executed at* $O(r)$ *worst-case cost including at most* r *carries.*

Proof. Consider the two sequences resulting from a cut. The first sequence is strictly regular and requires no changes. The second sequence may have a preceding 1^*2 block followed by a strictly-regular subsequence. In such case, we perform a *fix-carry* on the 2 ending such block to reestablish strict regularity. A catenation requires a fix only if *rep*(d) ends with a 01^+ block and *rep*(d') is not equal to 1^+. In such case, we perform a *fix-borrow* on the first 0 of *rep*(d'). An addition is implemented by adding the digits of one sequence to the other starting from the least-significant digit, simultaneously updating the pointers to the extreme digits in the other sequence, while maintaining strict regularity. Since each increment propagates at most one *fix-carry*, the bounds follow. $\square$

2.3 Strictly-Regular Trees

We recursively define a *strictly-regular tree* such that every subtree is as well a strictly-regular tree. For every node x in such a tree

- the rank, in brief *rank*(x), is equal to the number of the children of x;
- the *cardinality sequence*, in which entry i records the number of children of rank i, is strictly regular.

The next lemma directly follows from the definitions and Lemma 1.

Table 2: d_i is displayed in bold. d_a is the first extreme digit after d_i, k is a positive integer, α denotes any combination of 1^+ and 01^*2 blocks, and ω any combination of 1^+ and 01^*2 blocks followed by at most one 01^+ block.

(a) Case analysis for *increment*(d, i).

Initial configuration	Action	Final configuration
$\alpha01^*\mathbf{2}$	$d_i \leftarrow 3$; *fix-carry*(d,i)	$\alpha01^*\mathbf{1}1$
$\alpha01^*\mathbf{2}1^k\omega$	$d_i \leftarrow 3$; *fix-carry*(d,i)	$\alpha01^*\mathbf{1}21^{k-1}\omega$
$\alpha01^*\mathbf{2}01^*2\omega$	$d_i \leftarrow 3$; *fix-carry*(d,i)	$\alpha01^*\mathbf{1}11^*2\omega$
$\alpha01^*\mathbf{2}01^k$	$d_i \leftarrow 3$; *fix-carry*(d,i)	$\alpha01^*\mathbf{1}11^k$
$\alpha\mathbf{1}$	$d_i \leftarrow 2$; *fix-carry*(d,i)	$\alpha\mathbf{0}1$
$\alpha\mathbf{1}1^k\omega$	$d_i \leftarrow 2$; *fix-carry*(d,i)	$\alpha\mathbf{0}21^{k-1}\omega$
$\alpha\mathbf{1}01^*2\omega$	$d_i \leftarrow 2$; *fix-carry*(d,i)	$\alpha\mathbf{0}11^*2\omega$
$\alpha\mathbf{1}01^k$	$d_i \leftarrow 2$; *fix-carry*(d,i)	$\alpha\mathbf{0}11^k$
$\alpha01^*\mathbf{1}1^*2$	$d_i \leftarrow 2$; *fix-carry*(d,a)	$\alpha01^*\mathbf{2}1^*01$
$\alpha01^*\mathbf{1}1^*21^k\omega$	$d_i \leftarrow 2$; *fix-carry*(d,a)	$\alpha01^*\mathbf{2}1^*021^{k-1}\omega$
$\alpha01^*\mathbf{1}1^*201^*2\omega$	$d_i \leftarrow 2$; *fix-carry*(d,a)	$\alpha01^*\mathbf{2}1^*011^*2\omega$
$\alpha01^*\mathbf{1}1^*201^k$	$d_i \leftarrow 2$; *fix-carry*(d,a)	$\alpha01^*\mathbf{2}1^*011^k$
$\alpha\mathbf{0}1^*2$	$d_i \leftarrow 1$; *fix-carry*(d,a)	$\alpha\mathbf{1}1^*01$
$\alpha\mathbf{0}1^*21^k\omega$	$d_i \leftarrow 1$; *fix-carry*(d,a)	$\alpha\mathbf{1}1^*021^{k-1}\omega$
$\alpha\mathbf{0}1^*201^*2\omega$	$d_i \leftarrow 1$; *fix-carry*(d,a)	$\alpha\mathbf{1}1^*011^*2\omega$
$\alpha\mathbf{0}1^*201^k$	$d_i \leftarrow 1$; *fix-carry*(d,a)	$\alpha\mathbf{1}1^*011^k$
$\alpha01^*\mathbf{1}1^*$	$d_i \leftarrow 2$	$\alpha01^*\mathbf{2}1^*$
$\omega\mathbf{0}$	$d_i \leftarrow 1$	$\omega\mathbf{1}$
$\alpha\mathbf{0}1^k$	$d_i \leftarrow 1$	$\alpha\mathbf{1}1^k$

(b) Case analysis for *decrement*(d, i).

Initial configuration	Action	Final configuration
$\alpha\mathbf{0}2\omega$	*fix-borrow*(d,i); $d_i \leftarrow 1$	$\alpha\mathbf{1}1\omega$
$\alpha\mathbf{0}1^k2\omega$	*fix-borrow*(d,i); $d_i \leftarrow 1$	$\alpha\mathbf{1}01^{k-1}2\omega$
$\alpha\mathbf{0}1^k$	*fix-borrow*(d,i); $d_i \leftarrow 1$	$\alpha\mathbf{1}01^{k-1}$
$\alpha01^*\mathbf{1}2\omega$	*fix-borrow*(d,i); $d_i \leftarrow 2$	$\alpha01^*\mathbf{2}1\omega$
$\alpha01^*\mathbf{1}1^k2\omega$	*fix-borrow*(d,i); $d_i \leftarrow 2$	$\alpha01^*\mathbf{2}01^{k-1}2\omega$
$\alpha01^*\mathbf{1}1^k$	*fix-borrow*(d,i); $d_i \leftarrow 2$	$\alpha01^*\mathbf{2}01^{k-1}$
$\alpha\mathbf{1}1^*02\omega$	*fix-borrow*(d,a); $d_i \leftarrow 0$	$\alpha\mathbf{0}1^*21\omega$
$\alpha\mathbf{1}1^*01^k2\omega$	*fix-borrow*(d,a); $d_i \leftarrow 0$	$\alpha\mathbf{0}1^*201^{k-1}2\omega$
$\alpha\mathbf{1}1^*01^k$	*fix-borrow*(d,a); $d_i \leftarrow 0$	$\alpha\mathbf{0}1^*201^{k-1}$
$\alpha01^*\mathbf{2}1^*02\omega$	*fix-borrow*(d,a); $d_i \leftarrow 1$	$\alpha01^*\mathbf{1}1^*21\omega$
$\alpha01^*\mathbf{2}1^*01^k2\omega$	*fix-borrow*(d,a); $d_i \leftarrow 1$	$\alpha01^*\mathbf{1}1^*201^{k-1}2\omega$
$\alpha01^*\mathbf{2}1^*01^k$	*fix-borrow*(d,a) ; $d_i \leftarrow 1$	$\alpha01^*\mathbf{1}1^*201^{k-1}$
$\alpha\mathbf{1}1^*$	$d_i \leftarrow 0$	$\alpha\mathbf{0}1^*$
$\alpha01^*\mathbf{1}$	$d_i \leftarrow 0$	$\alpha01^*$
$\alpha01^*\mathbf{2}1^*$	$d_i \leftarrow 1$	$\alpha01^*\mathbf{1}1^*$

Lemma 3. *Let $\langle d_0, d_1, \ldots d_{r-1}\rangle$ be the cardinality sequence of a node x in a strictly-regular tree. If the last block of this sequence is a 01^+ block, then $rank(x) = r - 1$; otherwise, $rank(x) = r$.*

The next lemma illustrates the exponentiality property for such trees.

Lemma 4. *A strictly-regular tree of rank r has at least 2^r nodes.*

Proof. The proof is by induction. The claim is clearly true for nodes of rank 0. Assume the hypothesis is true for all the subtrees of a node x with rank r. Let y be the child of x with the largest rank. From Lemma 3, if the last block of the cardinality sequence of x is a 01^+ block, then $rank(x) = rank(y)$. Using induction, the number of nodes of y's subtree is at least 2^r, and the lemma follows. Otherwise, the cardinality sequence of x only contains 01^*2 and 1^+ blocks. We conclude that there exists a distinct subtree of x whose rank is at least i, for every i from 0 to $r-1$. Again using induction, the size of the tree rooted at x must be at least $1 + \sum_{i=0}^{r-1} 2^i = 2^r$. □

The operations that we would like to efficiently support include: adding a subtree whose root has rank at most r to the children of x; detaching a subtree from the children of x; splitting the sequence of the children of x, those having the highest ranks and the others; and concatenating a strictly-regular subsequence of trees, whose smallest rank equals r, to the children of x.

In accordance, we need to support implementations corresponding to the subroutines *fix-carry* and *fix-borrow*. For these, we use *link* and *unlink*.

link(T_1, T_2)**:** Assert that the roots of T_1 and T_2 have the same rank. Make one root the child of the other, and increase the rank of the surviving root by 1.

unlink(T)**:** Detach a child with the largest rank from the root of tree T. If T has rank r, the resulting two trees will have ranks either $r-1$, $r-1$ or $r-1$, r.

Subroutine *fix-carry*(d, i), which converts two consecutive digits $d_i = 2$ and $d_{i+1} = q$ to $0, q+1$ is realizable by subroutine *link*. Subroutine *fix-borrow*(d, i), which converts two consecutive digits $d_i = 0$ and $d_{i+1} = q$ to $2, q-1$ is realizable by subroutine *unlink* that results in two trees of equal rank. However, unlinking a tree of rank r may result in one tree of rank $r-1$ and another of rank r. In such case, a *fix-borrow* corresponds to converting the two digits $0, q$ to $1, q$. For this scenario, as for Table 2(b), it is also easy to show that all the cases following a decrement lead to a strictly-regular sequence. We leave the details for the reader to verify.

3 Application: Meldable Priority Queues

Our motivation is to investigate the worst-case bound for the number of element comparisons performed by *delete* under the assumption that *find-min*, *insert*, and *meld* have $O(1)$ worst-case cost. From the comparison-based lower bound for sorting, we know that if *find-min* and *insert* only involve $O(1)$ element comparisons, *delete* has to perform at least $\lg n - O(1)$ element comparisons, where n is the number of elements stored prior to the operation.

3.1 Brodal's Meldable Priority Queues

Our development is based on the priority queue presented in [7]. In this section, we describe this data structure. We also analyse the constant factor in the bound on the number of element comparisons performed by *delete*, since the original analysis was only asymptotic.

The construction in [7] is based on two key ideas. First, *insert* is supported at $O(1)$ worst-case cost. Second, *meld* is reduced to *insert* by allowing a priority queue to store other priority queues inside it. To make this possible, the whole data structure is a tree having two types of nodes: $\boxdot$-nodes (read: *square* or *type-I nodes*) and $\odot$-nodes (read: *circle* or *type-II nodes*). Each node stores a locator to an element, which is a representative of the descendants of the node; the representative has the smallest element among those of its descendants.

Each node has a non-negative integer *rank*. A node of rank 0 has no $\odot$-children. For an integer $r > 0$, the $\odot$-children of a node of rank r have ranks from 0 to $r - 1$. Each node can have at most one $\boxdot$-child and that child can be of arbitrary rank. The number of $\odot$-children is restricted to be at least one and at most three per rank. More precisely, the *regularity constraint* posed is that the cardinality sequence is of the form $\left(1 \mid 2 \mid 12^*3\right)^*$. This regular number system allows for increasing the least significant digit at $O(1)$ worst-case cost. In addition, because of the zeroless property, the size of a subtree of rank r is at least 2^r and the number of children of its root is at most $2r$. The rank of the root is required to be zero. So, if the tree holds more than one element, the other elements are held in the subtree rooted at the $\boxdot$-child of the root.

To represent such multi-way tree, the standard child-sibling representation can be used. Each node stores its rank as an integer, its type as a Boolean, a pointer to its parent, a pointer to its sibling, and a pointer to its $\odot$-child having the highest rank. The children of a node are kept in a circular singly-linked list containing the $\odot$-children in rank order and the $\boxdot$-child after the $\odot$-child of the highest rank; the $\boxdot$-child is further connected to the $\odot$-child of rank 0. Additionally, each node stores a pointer to a linked list, which holds pointers to the first $\odot$-node in every group of three consecutive nodes of the same rank corresponding to a 3 in the cardinality sequence.

A basic subroutine used in the manipulation of these trees is *link*. For node u, let $element(u)$ denote the element associated with u. Let u and v be two nodes of the same rank such that $element(u) \leq element(v)$. Now, *link* makes v a $\odot$-child of u. This increases the rank of u by one. Note that *link* has $O(1)$ worst-case cost and performs one element comparison.

The minimum element is readily found by accessing the root of the tree, so *find-min* is easily accomplished at $O(1)$ worst-case cost.

When inserting a new element, a node is created. The new element and those associated with the root and its $\boxdot$-child are compared; the two smallest among the three are associated with the root and its $\boxdot$-child, and the largest is associated with the created node. Hereafter, the new node is added as a $\odot$-child of rank 0 to the $\boxdot$-child of the root. Since the cardinality sequence of that node was regular before the insertion, only $O(1)$ structural changes are necessary to restore the regularity constraint. That is, *insert* has $O(1)$ worst-case cost.

To meld two trees, the elements associated with the root and its $\boxdot$-child are taken from both trees and these four elements are sorted. The largest element is associated with a $\boxdot$-child of the root of one tree. Let T be that tree, and let S be the other tree. The two smallest elements are then associated with the root of S and its $\boxdot$-child. Accordingly, the other two elements are associated with the root of T and its $\boxdot$-child. Subsequently, T is added as a rank-0 $\odot$-child to the $\boxdot$-child of the root of S. So, also *meld* has $O(1)$ worst-case cost.

When deleting an element, the corresponding node is located and made the current node. If the current node is the root, the element associated with the $\boxdot$-child of the root is swapped with that associated with the root, and the $\boxdot$-child of the root is made the current node. On the other hand, if the current node is a $\odot$-node, the elements associated with the current node and its parent are swapped until a $\boxdot$-node is reached. Therefore, both cases reduce to a situation where a $\boxdot$-node is to be removed.

Assume that we are removing a $\boxdot$-node z. The actual removal involves finding a node that holds the smallest element among the elements associated with the children of z (call this node x), and finding a node that has the highest rank among the children of x and z (call this node y). To reestablish the regularity constraint, z is removed, x is promoted into its place, y is detached from its children, and all the children previously under x and y, plus y itself, are moved under x. This is done by performing repeated linkings until the number of nodes of the same rank is one or two. The rank of x is updated accordingly.

In the whole deletion process $O(\lg n)$ nodes are handled and $O(1)$ work is done per node, so the total cost of *delete* is $O(\lg n)$. To analyse the number of element comparisons performed, we point out that a node with rank r can have up to $2r$ $\odot$-children (not $3r$ as stated in [7]). Hence, finding the smallest element associated with a node requires up to $2\lg n + O(1)$ element comparisons, and reducing the number of children from $6\lg n + O(1)$ to $\lg n + O(1)$ involves $5\lg n + O(1)$ element comparisons (each *link* requires one). To see that this bound is possible, consider the addition of four numbers 1, 1232^k, 2222^k, and 1232^k (where the least significant digits are listed first), which gives $1211^{k+1}2$.

Our discussion so far can be summarized as follows.

Theorem 3. *Brodal's meldable priority queue, as described in [7], supports find-min, insert, and meld at* $O(1)$ *worst-case cost, and delete at* $O(\lg n)$ *worst-case cost including at most* $7\lg n + O(1)$ *element comparisons.*

3.2 Our Improvement

Consider a simple mixed scheme, in which the number system used for the children of $\odot$-nodes is perfect, following the pattern 1^*, and that used for the children of $\boxdot$-nodes is regular. This implies that the $\odot$-nodes form binomial trees [3]. After this modification, the bounds for *insert* and *meld* remain the same if we rely on the delayed melding strategy. However, since each node has at most

$\lg n + O(1)$ children, the bound for *delete* would be better than that reported in Theorem 3. Such an implementation of *delete* has three bottlenecks: finding the minimum, executing a delayed *meld*, and adding the $\odot$-children of a $\boxdot$-node to another node. In this mixed system, each of these three procedures requires at most $\lg n + O(1)$ element comparisons. Accordingly, *delete* involves at most $3\lg n + O(1)$ element comparisons. Still, the question is how to do better!

The major change we make is to use the strictly-regular system instead of the zeroless regular system. We carry out *find-min*, *insert*, and *meld* similar to [7]. We use subroutine *merge* to combine two trees. Let y and y' be the roots of these trees, and let r and r' be their respective ranks where $r \leq r'$. We show how to *merge* the two trees at $O(r)$ worst-case cost using $O(1)$ element comparisons. For this, we have to locate the nodes representing the extreme digits closest to r in the cardinality sequence of y'. Consequently, by Theorems 1 and 2, a cut or an increment at that rank is done at $O(1)$ worst-case cost. If $\mathit{element}(y') \leq \mathit{element}(y)$, add y as a $\odot$-child of y', update the rank of y' and stop. Otherwise, cut the $\odot$-children of y' at r. Let the two resulting sublists be C and D, C containing the nodes of lower rank. Then, concatenate the lists representing the sequence of the $\odot$-children of y and the sequence D. We regard y' together with the $\odot$-children in C and y''s earlier $\boxdot$-child as one tree whose root y' is a $\odot$-node. Finally, place this tree under y and update the rank of y.

Now we show how to improve *delete*. If the node to be deleted is the root, we swap the elements associated with the root and its $\boxdot$-child, and let that $\boxdot$-node be the node z to be deleted. If the node to be deleted is a $\odot$-node, we repeatedly swap the elements associated with this node and its parent until the current node is a $\boxdot$-node (Case 1) or the rank of the current node is the same as that of its parent (Case 2). When the process stops, the current node z is to be deleted.

Case 1: z is a $\boxdot$-node. Let x denote the node that contains the smallest element among the children of z (if any). We remove z, lift x into its place, and make x into a $\boxdot$-node. Next, we move all the other $\odot$-children of z under x by performing an addition operation, and update the rank of x. Since z and x may each have had a $\boxdot$-child, there may be two $\boxdot$-children around. In such case, *merge* such two subtrees and make the root of the resulting tree the $\boxdot$-child of x.

Case 2: z is a $\odot$-node. Let p be the parent of z. We remove z and move its $\odot$-children to p by performing an addition operation. As $\mathit{rank}(p) = \mathit{rank}(z)$ before the addition, $\mathit{rank}(p) = \mathit{rank}(z)$ or $\mathit{rank}(z) + 1$ after the addition. If $\mathit{rank}(p) = \mathit{rank}(z) + 1$, to ensure that $\mathit{rank}(p)$ remains the same as before the operation, we detach the child of p that has the highest rank and *merge* the subtree rooted at it with the subtrees rooted at the $\boxdot$-children of p and z (there could be up to two such subtrees), and make the root of the resulting tree the $\boxdot$-child of p.

Let r be the maximum rank of a node in the tree under consideration. Climbing up the tree to locate a node z has $O(r)$ cost, since after every step the new current node has a larger rank. In Case 1, a $\boxdot$-node is deleted at $O(r)$ cost involving at most r element comparisons when finding its smallest child. In Cases

1 and 2, the addition of the $\odot$-children of two nodes has $O(r)$ cost and requires at most r element comparisons. Additionally, applying the *merge* operation on two trees (Case 1) or three trees (Case 2) has $O(r)$ cost and requires $O(1)$ element comparisons. Thus, the total cost is $O(r)$ and at most $2r + O(1)$ element comparisons are performed. Using Lemma 4, $r \leq \lg n$, and the claim follows.

In summary, our data structure improves the original data structure in two ways. First, by Lemma 4, the new system reduces the maximum number of children a node can have from $2 \lg n$ to $\lg n$. Second, the new system breaks the bottleneck resulting from delayed melding, since two subtrees can be merged with $O(1)$ element comparisons. The above discussion implies the following theorem.

Theorem 4. *Let n denote the number of elements stored in the data structure prior to a deletion. There exists a priority queue that supports find-min, insert, and meld at $O(1)$ worst-case cost, and delete at $O(\lg n)$ worst-case cost including at most $2 \lg n + O(1)$ element comparisons.*

4 Other Applications

Historically, it is interesting to note that in early papers a number system supporting increments and decrements of an arbitrary digit was constructed by putting two regular systems back to back, i.e. $d_i \in \{0, 1, 2, 3, 4, 5\}$. It is relatively easy to prove the correctness of this system. This approach was used in [14] for constructing catenable deques, in [9] for constructing catenable finger search trees, and in [8] for constructing meldable priority queues. (In [8], $d_i \in \{2, 3, 4, 5, 6, 7\}$ is imposed, since an extra constraint that $d_i \geq 2$ was required to facilitate the violation reductions and to guarantee the exponentiality property.) Later on, it was realized that the extended-regular system, $d_i \in \{0, 1, 2, 3\}$, could be utilized for the same purpose (see, for example, [6]). The strictly-regular system may be employed in applications where these more extensive number systems have been used earlier. This replacement, when possible, would have two important consequences:

1. The underlying data structures become simpler.
2. The operations supported may become a constant factor faster.

While surveying papers that presented potential applications to the new number system, we found that, even though our number system may be applied, there were situations where other approaches would be more favourable. For example, the relaxed heap described in [11] relies on the zeroless extended-regular system to support increments and decrements. Naturally, the strictly-regular system could be used instead, and this would reduce the number of trees that have to be maintained. However, the approach of using a two-tier structure as described in [11] makes the reduction in the number of trees insignificant since the amount of work done is proportional to the logarithm of the number of trees. Also, a fat heap [6] uses the extended-regular binary system for keeping track of the potential violation nodes and the extended-regular ternary system for keeping track of the trees in the structure. However, we discovered that a priority queue with the same functionality and efficiency can be implemented with simpler tools without using number systems at all. The reader is warned: number systems are powerful tools but they should not be applied haphazardly.

Up till now we have ignored the cost of accessing the extreme digits in the vicinity of a given digit. When dealing with the regular or the extended-regular systems this can be done at $O(1)$ cost by using the guides described in [8]. In contrary, for our number system, accessing the extreme digits in the vicinity of any digit does not seem to be doable at $O(1)$ cost. However, the special case of accessing the first and last extreme digits is soluble at $O(1)$ cost.

In some applications, like fat heaps [6] and the priority queues described in [8], the underlying number system is ternary. We have not found a satisfactory solution to extend the strictly-regular system to handle ternary numbers efficiently; it is an open question whether such an extension exists.

References

1. Clancy, M., Knuth, D.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Dept. of Computer Science, Stanford University (1977)
2. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A new representation for linear lists. In: Proceedings of the 9th Annual ACM Symposium on Theory of Computing, ACM (1977) 49–60
3. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM **21**(4) (1978) 309–315
4. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
5. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. 2nd edn. Pearson Education, Inc. (2007)
6. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and Boolean union-find. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, ACM (2002) 573–582
7. Brodal, G.S.: Fast meldable priority queues. In: Proceedings of the 4th International Workshop on Algorithms and Data Structures. Volume 955 of Lecture Notes in Computer Science., Springer-Verlag (1995) 282–290
8. Brodal, G.S.: Worst-case efficient priority queues. In: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM/SIAM (1996) 52–58
9. Kaplan, H., Tarjan, R.E.: Purely functional representations of catenable sorted lists. In: Proceedings of the 28th Annual ACM Symposium on Theory of Computing, ACM (1996) 202–211
10. Elmasry, A.: A priority queue with the working-set property. International Journal of Foundations of Computer Science **17**(6) (2006) 1455–1465
11. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Informatica **45**(3) (2008) 193–210
12. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms **5**(1) (2008) Article 14
13. Jensen, C.: A note on meldable heaps relying on data-structural bootstrapping. CPH STL Report 2009-2, Department of Computer Science, University of Copenhagen (2009) Available at http://cphstl.dk.

14. Kaplan, H., Tarjan, R.E.: Persistent lists with catenation via recursive slow-down. In: Proceedings of the 27th Annual ACM Symposium on Theory of Computing, ACM (1995) 93–102