In-Place Binary Counters*

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

[1] Department of Computer Engineering and Systems, Alexandria University
Alexandria 21544, Egypt
[2] Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

Abstract. We introduce a binary counter that supports increments and decrements in $O(1)$ worst-case time per operation. (We assume that arithmetic operations on an index variable that is stored in one computer word can be performed in $O(1)$ time each.) To represent any integer in the range from 0 to 2^n-1, our counter uses an array of at most n bits plus few words of $\lceil \lg(1+n) \rceil$ bits each. Extended-regular and strictly-regular counters are known to also support increments and decrements in $O(1)$ worst-case time per operation, but the implementation of these counters would require $O(n)$ words of extra space, whereas our counter only needs $O(1)$ words of extra space. Compared to other space-efficient counters, which rely on Gray codes, our counter utilizes codes with binary weights allowing for its usage in the construction of efficient data structures.

1 Introduction

A *numeral system* provides a specification for how to represent integers. In a positional numeral system, a string $\boldsymbol{d} = \langle d_{n-1}, \ldots, d_1, d_0 \rangle$ of *digits* is used to represent an integer, n being the length of the representation. As in the decimal system, d_0 denotes the least-significant digit, d_{n-1} the most-significant digit, and $d_{n-1} \neq 0$. If w_i is the *weight* of d_i, the string represents the decimal number $\mathit{value}(\boldsymbol{d}) = \sum_{i=0}^{n-1} d_i w_i$. (An empty string can be used to represent zero.)

A numeral system comprises four components:

1. The *digit set* specifies the values that the digits can take. For example, in a *redundant binary system* the digit set is $\{0, 1, 2\}$.
2. The *weight set* specifies the weights that the digits represent when determining the decimal value of the underlying integer. For example, a binary system uses the *binary weights* $w_i = 2^i$ where $i \in \{0, 1, \ldots, n-1\}$.
3. The *rule set* specifies the rules that the representation of each integer must obey. For example, the *regular system* [3] is a redundant binary system where every two 2's have at least one 0 in between.
4. The *operation set* specifies the operations that are to be supported. In this paper we only consider in details *increment* (increase the value by one) and *decrement* (decrease the value by one) operations. However, it is also relevant to support other arithmetic operations like additions and subtractions.

A representation of an integer that is subject to increments and decrements is called a *counter*. To represent an integer in the range $[0\,..\,2^n-1]$, the ordinary binary counter is space-efficient requiring n bits, but an *increment* or a *decrement* requires $\Theta(n)$ bit flips in the worst case. A *regular counter*—a counter using the regular system [3]—supports increments (of arbitrary digits) with a constant number of digit changes per operation. An *extended-regular counter* [3, 10, 13] uses the digit set $\{0, 1, 2, 3\}$ imposing a rule set that between any two 3's there is a digit other than 2 and between any two 0's there is a digit other than 1. Such a counter supports both increments and decrements (of arbitrary digits) with a constant number of digit changes per operation. A *strictly-regular counter* [9] provides the same guarantee, but it uses the digit set $\{0, 1, 2\}$. Unfortunately, the implementation of the aforementioned regular counters would require up to n indices in addition to the space needed by the digits themselves.

Recently, efficient and more space-economical counters were proposed by Bose et al. [1], Brodal et al. [2], and Rahman and Munro [16]. In these papers the complexity of the operations was analysed in the bit-probe model. In [2], using a representation of $n+1$ bits, each of the *increment* and *decrement* operations could be accomplished by reading $\lg n + O(1)$ bits and writing $O(1)$ bits.

We use the word-RAM model [12] as our model of computation. If a counter requires n bits, these bits are kept compactly in an array of $\lceil n/w \rceil$ words, where w is the size of the machine word in bits. In addition to these words, we only allow the usage of $O(1)$ other words. Also we assume that, for a problem of size n, $w \geq \lceil \lg(1+n) \rceil$, i.e. a variable counting the number of bits and a variable referring to a position in the array of bits can each fit in one computer word.

In this paper we introduce an in-place binary counter; it uses $n + O(\lg n)$ bits, and supports *increment* and *decrement* operations in $O(1)$ worst-case time. This solves an open problem stated, for example, in Demaine's lecture notes [4]. In the bit-probe model, both our *increment* and *decrement* operations involve $O(\lg n)$ bit accesses and modifications. However, the bits accessed and modified are stored in a constant number of words, and we only perform $O(1)$ word operations per *increment* and *decrement*. We can also test whether the value of the counter is zero in $O(1)$ worst-case time. Conceptually, our counter is a modification of a regular counter; instead of giving preference to handling the carries (2's) at the least-significant end of the representation, we handle the carries at the most-significant end first. Although increments are easy and direct, incorporating decrements is more involved and tricky. A simple consequence of our construction is a new representation of positive integers, using binary weights, in which a positive integer with value K is coded using $\lg K + O(\lg \lg K)$ bits and the encoding differs from that of $K+1$ only in $O(\lg \lg K)$ bits.

Compared to other space-efficient counters, our counter has a significant advantage: It can be applied in the construction of efficient data structures [3, 17]. For a survey on numeral systems and their applications to data structures, we refer to [14, Chapter 9]. The idea is to relate the number of objects of a specific type in a data structure to the value of a digit. Often two objects of the same size can be combined efficiently, and one object can be split into two objects

of the same size. These operations are the exact counterparts for the carry and borrow operations that are employed by binary counters. On the other hand, the known space-economical counters [1, 2, 16] rely on some variant of a Gray code [11], and more involved operations, like bit flips, required by a Gray code can seldom be simulated at the level of the data structure. Because the upper bound for the sum of the digits of our counter is optimized (the sum of the digits representing a positive integer K is at most $\lceil \lg(1+K) \rceil$), the number of objects in the corresponding data structure is bounded from above as well.

The drawback of the ordinary binary counter is that *increment* costs $O(1)$ only in the amortized (not wost-case) sense, and that it cannot support both *increment* and *decrement* operations efficiently at the same time. Since our in-place counter supports both *increment* and *decrement* operations efficiently, it can be used as a replacement for the ordinary binary counter even in applications where space-saving is not the main goal. As for the data structure, the cost of the *insert* operation, which resembles *increment* and appends a new element to the data structure, is $O(1)$ in the worst case. Additionally, the cost of the *borrow* operation, which resembles *decrement* and removes an arbitrary element from the data structure, is $O(1)$ in the worst case. The importance of fast *borrow* has been demonstrated in several earlier papers; see for example [5–7, 13].

2 The Data Structure

Our objective is to implement a counter that represents an integer in the range $[0\,..\,2^n - 1]$ with at most $n + O(\lg n)$ bits. Assuming $\lceil \lg(1+n) \rceil$ bits fit in one computer word, our counter uses a constant number of words in addition to the n bits. To represent a positive integer K, the counter has the following characteristics:

C$_1$. The sum of the digits is at most $\lceil \lg(1+K) \rceil$.
C$_2$. Other than the least-significant digit, at most one digit has value 2, and all the other digits are 0's and 1's.
C$_3$. The most-significant digit is always non-zero. Due to the binary weights, we have $K \geq 2^{n-1}$ implying $n \leq \lceil \lg(1+K) \rceil$.

Let ℓ denote the current length of the number representation and assume that the string of digits in the representation is $\boldsymbol{d} = \langle d_{\ell-1}, d_{n-2}, \ldots, d_0 \rangle$. A straightforward implementation of our in-place counter is to store the value of d_0, which is at most ℓ, in one word that we call x. In addition, we store the index of the digit with value 2 (if any such digit other than the least-significant one exists); this also consumes one word that we call α. Each of the other digits is either 0 or 1. To store these bits, we assume the availability of an (infinite) array $\boldsymbol{b}$, the first ℓ bits (or $\lceil \ell/w \rceil$ words) of which are in use. To be able to efficiently incorporate the *decrement* operations, we shall also use two more variables: β and ζ. The meaning of β will be explained later; ζ counts the number of 0 bits in $\boldsymbol{d}$. The representation of the integer 50 is given in Fig. 1.

$$\begin{array}{cc} & \ell = 6 \\ \langle\ 1,\ 0,\ 1,\ 1,\ 0,\ -\ \rangle & x = 2 \\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0 & carry = 1 \text{ (indicates that there is a 2)} \\ & \alpha = 2 \\ & \zeta = 2 \end{array}$$

Fig. 1. Representation of the integer 50 when only *increment* is supported

To distinguish whether there is a 2 in the representation or not, while still using as few bits as indicated, we use an extra bit called *carry* (which could actually be the unused bit b_0) and adopt the following convention:

- If $carry = 1$, then the actual value of d_α is 2. Otherwise, d_α equals b_α.

We still maintain the stored bit in b_α to be 1 when $carry = 1$. Accordingly,

$$value(\boldsymbol{d}) = \begin{cases} x + \sum_{i=1}^{\ell-1} b_i \cdot 2^i & \text{if } carry = 0, \\ x + 2^\alpha + \sum_{i=1}^{\ell-1} b_i \cdot 2^i & \text{if } carry = 1. \end{cases}$$

The following procedure is used to initialize our counter to zero.

Algorithm *initialize*($\boldsymbol{b}$, ℓ, x, $carry$, α, β, ζ)

1: $\ell \leftarrow 1$
2: $x \leftarrow 0$
3: $carry \leftarrow 0$
4: $\alpha \leftarrow 1$
5: $\beta \leftarrow 1$
6: $\zeta \leftarrow 0$

2.1 Increments

To maintain $\mathbf{C}_1$ we need a mechanism to reduce the sum of the digits within the *increment* operations. Instead of monitoring this sum until it reaches the threshold, we reduce the sum of the digits by one with every *increment* operation whenever possible. We use a procedure called *fix-carry* that works as follows: If there exists an index $\alpha \neq 0$ where d_α is a 2 (i.e. $carry = 1$), we set d_α to 0 and increase $d_{\alpha+1}$ by one. Otherwise, if $d_0 \geq 2$, we decrease d_0 by two and increase d_1 by one. Note that a *fix-carry* does not change the value of the number, and in addition it maintains all the aforementioned characteristics.

To increment a number, we perform a *fix-carry* operation and add one to x.

Algorithm *increment*($\boldsymbol{b}$, ℓ, x, $carry$, α, ζ)

1: *fix-carry*($\boldsymbol{b}$, ℓ, x, $carry$, α, ζ)
2: $x \leftarrow x + 1$

Algorithm *fix-carry*($\boldsymbol{b}$, ℓ, x, *carry*, α, ζ)

```
1:  if carry = 1
2:     b_α ← 0
3:     ζ ← ζ + 1
4:     α ← α + 1
5:  else if x ≥ 2
6:     x ← x − 2
7:     α ← 1
8:  else
9:     return
10: if α = ℓ
11:    ℓ ← ℓ + 1
12:    b_{ℓ−1} ← 1
13:    carry ← 0
14: else if b_α = 0
15:    b_α ← 1
16:    ζ ← ζ − 1
17:    carry ← 0
18: else
19:    carry ← 1
```

For more illustration, here are the first 50 positive integers obtained by applying *increment* repeatedly: 1, 2, 11, 12, 21, 102, 111, 112, 121, 202, 1003, 1012, 1021, 1102, 1111, 1112, 1121, 1202, 2003, 10004, 10013, 10022, 10103, 10112, 10121, 10202, 11003, 11012, 11021, 11102, 11111, 11112, 11121, 11202, 12003, 20004, 100005, 100014, 100023, 100104, 100113, 100122, 100203, 101004, 101013, 101022, 101103, 101112, 101121, 101202.

We would further characterize any representation of our counter, resulting from a sequence of *increment* operations, with the following properties:

P$_1$. The value of the least-significant digit is greater than 0.

P$_2$. If there is a digit with value 2 other than the least-significant digit, all the digits between this 2 and the least-significant digit are 0's.

In consequence, the number $\boldsymbol{d}$ is kept as a string of the form $(1(0|1)^*)^*x$ or $(1(0|1)^*)^*20^*x$, where x is a positive integer and * denotes zero or more repetitions of the preceding digit or string of digits.

P$_3$. If $\boldsymbol{d}$ is of the form 1^*2, there is obviously no 0 digits in $\boldsymbol{d}$. Otherwise, if $\boldsymbol{d}$ is of the form 1^*20^*x or if there is no digit in $\boldsymbol{d}$ with value 2 other than (possibly) the least-significant digit, the number of 0 digits in $\boldsymbol{d}$ equals $x-1$. Otherwise, the number of 0 digits in $\boldsymbol{d}$ equals x.

We can thus express this property using the following trichotomy:

$$\begin{cases} \zeta = 0 & \text{if } \boldsymbol{d} \in 1^*2, \\ \zeta = x - 1 & \text{if } \boldsymbol{d} \in 1^*20^*x \cup (1(0|1)^*)^*x, \\ \zeta = x & \text{otherwise.} \end{cases} \tag{1}$$

As a consequence of **P$_3$**, the following property also holds:

$\mathbf{P}_4$**.** If no digit is larger than 1, then all the digits are 1's.

This property results from $\mathbf{P}_3$ because $carry = 0$ and $x = 1$ imply $\zeta = 0$.

Lemma 1. *The increment operation sustains the characteristics and properties.*

Proof. Trivially the characteristics are valid for a counter whose value is one. Increasing the least-significant digit by one may only break $\mathbf{C}_1$ if the sum of the digits exceeds the threshold by one. The *fix-carry* operation decreases the sum of the digits by one returning it back below the threshold, unless no digit with a value greater than one exists. In this latter case, assuming that the sum of the digits after the *increment* operation is s, then the resulting integer is at least 2^{s-1}; this ensures the validity of $\mathbf{C}_1$. If before this operation there exists $\alpha \neq 0$ where d_α has value 2, then after the *increment* operation d_α is 0 and at most one digit other than d_0 with value 2 may exist; this digit is $d_{\alpha+1}$. If before the *increment* operation there was no digit other than d_0 with value 2, at most one digit may have value 2 after the *increment* operation; this digit is d_1. The validity of $\mathbf{C}_2$ follows.

It is easy to verify that the *increment* operation maintains $\mathbf{P}_1$ and $\mathbf{P}_2$. We show next, using induction on the counter values, that $\mathbf{P}_3$ and accordingly $\mathbf{P}_4$ are true. Initially, for $\boldsymbol{d} = 12$ the first case of (1) holds. Later on, an *increment* applied to $\boldsymbol{d} \in 11^*2$ results in $\boldsymbol{d} \in 1^*21$ and the second case of (1) holds. Starting with $\boldsymbol{d} \in 1^*20^*x$, an *increment* will increase both x and ζ by one, and the second equation of (1) will still be valid. Starting with $\boldsymbol{d} \in 1(0|1)^*0x$ when $x \geq 2$, an *increment* will decrease both x and ζ by one, and the second equation of (1) will still be valid. Alternatively, starting with $\boldsymbol{d} \in 1(0|1)^*0(0|1)^*1x$ when $x \geq 2$, an *increment* will decrease x by one, while ζ will not change, and the third case of (1) will then be fulfilled. A complementing state for these last two is when $x = 1$; we may assume then, using $\mathbf{P}_4$, that $\boldsymbol{d} \in 1^*1$. In such a case, an *increment* will result in $\boldsymbol{d} \in 1^*2$, and we are back to the first case of (1). Lastly, consider the third case where $\boldsymbol{d} \in 1(0|1)^*0(0|1)^*20^*x$. Starting with $\boldsymbol{d} \in 1(0|1)^*0(0|1)^*120^*x$, an *increment* will increase both x and ζ by one, and the third equation of (1) will still be valid. Staring with $\boldsymbol{d} \in 1(0|1)^*020^*x$, an *increment* will increase x by one, while ζ will not change, and the second case of (1) will then be fulfilled. □

Note that, for any sequence of *increment* operations, $\mathbf{C}_1$ can even be shown to hold with equality. To advocate for this, we point out that every *fix-carry* operation decreases the sum of the digits by one, except when all the digits are 1's. In other words, the only case the sum of the digits increases is when the value of the counter becomes a power of two after the *increment* operation.

2.2 Decrements

Our objective is to implement the *decrement* operations as the reverse of the corresponding *increment* operations. To efficiently implement *fix-borrow*, we need to figure out the changes that were made to the counter when it was last increased beyond its current value. More precisely, we need to know the index of

the digit that the corresponding *fix-carry* operation changed back in history. The *fix-carry* operation used the index, say γ, of the currently second-least-significant non-zero digit. Assuming that γ is available, the *fix-borrow* operation decreases the corresponding digit d_γ by one and increases its preceding digit by two. This may result in losing track of γ!

Algorithm *fix-borrow*($\boldsymbol{b}$, ℓ, x, *carry*, γ, ζ)

```
1:  if carry = 1
2:      b_γ ← 1
3:  else if γ = ℓ − 1
4:      ℓ ← ℓ − 1
5:  else
6:      b_γ ← 0
7:      ζ ← ζ + 1
8:  if γ > 1
9:      γ ← γ − 1
10:     b_γ ← 1
11:     carry ← 1
12:     ζ ← ζ − 1
13: else          // we lose track of the correct value of γ if carry = 0
14:     x ← x + 2
15:     carry ← 0
```

If it happens that $d_\alpha \neq 0$, then we are lucky as γ is equal to α that we are already maintaining. However, d_α may become 0 following the *decrement* operation. To see the problem, consider for example the case when *decrement* is to be applied to a number $10000001x$. Before this operation, γ is equal to 1. But the preceding number that resulted in this number via the corresponding *increment* operation is $10000000(x+1)$ with $\gamma = 8$. How would we find the new value of γ in constant time within the *decrement* operation?

The critical property that gets us to a worst-case constant-time implementation of *decrement* is $\mathbf{P}_3$. Our idea is to decrease the value of the least-significant digit with every *decrement*, while possibly not performing a *fix-borrow*. More precisely, within some *decrement* operations we incrementally walk through the zero digits, two digits at a time, until we reach the digit d_γ. We then perform the postponed *fix-borrow* operations within the upcoming *decrement* operations working with double the speed (two *fix-borrow* operations at a time). To efficiently implement the *decrement* operations, we maintain the index α that we are up to so far in our search for γ, leaving behind its old (before starting the search) value stored as β. In accordance, the *decrement* operations work in three modes: In the *normal mode*, when $d_\beta \neq 0$ (which implies $\alpha = \beta$), each *decrement* operation performs one *fix-borrow* operation using the index α as the argument. In the *search mode*, each *decrement* operation sequentially traverses the next two digits and increases α by two as long as both digits are 0's. Once a *decrement* operation reaches a non-zero digit, i.e. $\alpha = \gamma$, we switch to the *rapid mode* as there are postponed *fix-borrow* operations; in this mode each *decrement* operation performs two *fix-borrow* operations using the index α as the argument.

Another subtle issue to be considered is when the corresponding *increment* operation has not performed a *fix-carry*. This only happens if the number we want to decrease is $\boldsymbol{d} \in 1^*2$. To be able to distinguish this case from the case $\boldsymbol{d} \in 11^*01^*2$, we check if there are no 0's in the number, i.e. $x = 2$ and $\zeta = 0$. In such a case, we skip the *fix-borrow* operation altogether. We also skip the *fix-borrow* operation if the number that is to be decreased is 1.

Algorithm *decrement*($\boldsymbol{b}$, ℓ, x, *carry*, α, β, ζ)

```
1:  if ℓ = 1 or (x = 2 and ζ = 0)
2:    x ← x − 1
3:    return
4:  if b_β ≠ 0          // work in normal mode
5:    fix-borrow(b, ℓ, x, carry, α, ζ)
6:    β ← α
7:  else if b_α = 0     // work in search mode
8:    α ← α + 1
9:    if b_α = 0
10:     α ← α + 1
11:   else              // switch to rapid mode
12:     fix-borrow(b, ℓ, x, carry, α, ζ)
13: else                // work in rapid mode
14:   fix-borrow(b, ℓ, x, carry, α, ζ)
15:   fix-borrow(b, ℓ, x, carry, α, ζ)
16: x ← x − 1
```

The correctness of the construction is a consequence of property $\mathbf{P}_3$ that implies that x will always be non-zero, and hence we can decrease its value while the *decrement* operations are working in any of the three modes. However, when incorporating the *decrement* operations, $\mathbf{P}_3$ would not hold as tight as before and must be relaxed as follows:

$\mathbf{P}_3$'. If there is a digit in $\boldsymbol{d}$ with value 2 other than the least-significant digit, the number of zero digits in $\boldsymbol{d}$ is bounded by *twice* the value of the least-significant digit. Otherwise, the number of zero digits is bounded by *twice* the value of the least-significant digit minus two.

$$\begin{cases} \zeta \leq 2x - 2 & \text{if } \mathit{carry} = 0, \\ \zeta \leq 2x & \text{if } \mathit{carry} = 1. \end{cases}$$

In the revised implementation of *increment* we have to consider the case when there are postponed *fix-borrow* operations. To be able to detect this, we make use of the index β. Whenever $\alpha > \beta$, it means that there have been more *decrement* operations than *increment* operations since the time we switched to the search mode. Here the *increment* operation only needs to undo what a preceding *decrement* has done. More precisely, in this case the *increment* operation increases the least-significant digit and instead of performing a *fix-carry* operation it moves β two steps forward. Once β and α meet, i.e. $\alpha = \beta$, this means that there is no postponed *fix-borrow* operations, and the *increment* operation should work in the normal mode by calling *fix-carry*.

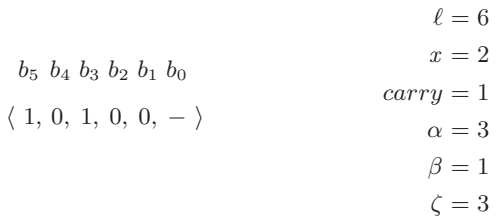

Fig. 2. A representation of the integer 50 when both *increment* and *decrement* are supported; this representation was obtained by starting from the representation of Fig. 1, performing three *increment* operations followed by three *decrement* operations

Algorithm *increment*($\boldsymbol{b}$, ℓ, x, *carry*, α, β, ζ)

1: **if** $\alpha = \beta$ // work in normal mode
2: *fix-carry*($\boldsymbol{b}$, ℓ, x, *carry*, α, ζ)
3: $\beta \leftarrow \alpha$
4: **else** // work in search mode
5: $\beta \leftarrow \beta + 2$
6: $x \leftarrow x + 1$

In Fig. 2 one representation of the integer 50 is given when the *decrement* operation and the revised *increment* operation are in use. This example shows that the representation of our integers is not unique, since a number can have several representations depending on the sequence of operations applied.

Note that the *increment* and *decrement* operations keep either $\alpha = \beta$ or $\alpha > \beta$ with $\alpha - \beta$ being an even positive integer.

Lemma 2. *The increment and decrement operations sustain the characteristics and the relaxed properties.*

Proof. The *fix-borrow* operation increases the sum of the digits by one. When two *fix-borrow* operations are performed per *decrement* in the rapid mode, the sum of the digits increases as a result. However, a second *fix-borrow* is executed only if it was postponed in a previous *decrement*, indicating that $\mathbf{C}_1$ was satisfied with a strict inequality; this ensures the validity of $\mathbf{C}_1$ following any *decrement* operation. On the other hand, the *increment* operation skips calling *fix-carry* only when $\alpha > \beta$. We start the search mode when $\alpha = \beta = 1$, a *decrement* moves α two steps forward, and an *increment* moves β two steps forward. The condition $\alpha > \beta$ then implies that there are postponed *fix-borrow* operations. Hence, there is no need to execute a *fix-carry* within such *increment* operation; this ensures the validity of $\mathbf{C}_1$ following any *increment* operation.

If there is a digit d_α whose value is 2, the *fix-borrow* operation decreases d_α to 1 and then adds two to the preceding digit. In accordance, at most one 2 may exist and is preceded by 0's up to (and not including) the least-significant digit. The same fact holds as a result of the *fix-carry* operation. The validity of $\mathbf{C}_2$ and that of $\mathbf{P}_2$ are thus sustained.

To prove $\mathbf{P}_3$', we use the relation $\alpha-\beta \leq \zeta$ that is true because $d_j = 0$ for all j satisfying $\alpha - 1 \geq j \geq \beta$. It follows that we only need to show the dichotomy:

$$\begin{cases} \zeta \leq x + (\alpha - \beta)/2 - 1 & \text{if } carry = 0, \\ \zeta \leq x + (\alpha - \beta)/2 & \text{if } carry = 1. \end{cases} \tag{2}$$

The proof is by induction on the operations sequence. The base case follows by noting that an initial sequence of *increment* operations maintains (1) and $\alpha = \beta$. When the *decrement* operations work in the search mode, $carry = 0$ and the first inequality of (2) holds. With each of those *decrement* operations, x decreases by one and α increases by two; the first inequality is still guaranteed. As an exception, the last *decrement* operation working in the search mode may execute a *fix-borrow* operation. As a result, the value of α does not change; but then we have $carry = 1$, guaranteeing the second inequality of (2). For each of the following *decrement* operations working in the rapid mode, x decreases by one and α decreases by two, but ζ decreases by two (via two *fix-borrow* operations); the second inequality of (2) is still valid. For a *decrement* operation to reset the *carry* bit back to 0, the 2 must vanish; this only happens if $\alpha = 1$. In such a case, as a result of the *decrement* operation, x increases by one and both α and ζ do not change; so now the first inequality of (2) is guaranteed. Each of the other *decrement* operations working in the normal mode keeps $\alpha = \beta$ and either decreases both x and ζ by one or increases both by one. An exception is when the *decrement* operation is applied to a number where $x = 2$ and $\zeta = 0$. In this case, the resulting number has all 1's, and the first inequality of (2) is still valid. An *increment* operation working in the search mode increases x by one and β by two. We also need to mention that the *increment* operations working in the normal mode maintain the induction hypothesis. This follows using arguments similar to those of Lemma 1 and by noting that these operations keep $\alpha = \beta$. In conclusion, the two operations in all modes maintain (2), and $\mathbf{P}_3$' is satisfied. It directly follows from the first inequality of $\mathbf{P}_3$' that $\mathbf{P}_4$ is also satisfied.

Using $\mathbf{P}_3$', since $\zeta \geq 0$, the only case where x could have possibly been 0 is when $carry = 1$. Contradictorily, it follows that in this case $\zeta = 0$. The value of the least-significant digit must then be greater than 0, and $\mathbf{P}_1$ holds. □

In fact, we can prove a tighter version of $\mathbf{C}_1$. We namely argue that $\sum_{i=0}^{\ell-1} d_i = \lceil \lg(1+K) \rceil - (\alpha - \beta)/2$. The equation is true when $\alpha = \beta$ (as we have shown for the *increment* operations); that is when the operations work in the normal mode. A *decrement* operation working in the search mode increases α by two, and hence decreases the right-hand side by one, but it decreases the left-hand side by one as well. A *decrement* operation working in the rapid mode calls *fix-borrow* twice and decreases x by one. As a result, the left-hand side increases by one and the right-hand side also increases by one (as α decreases by two). An *increment* operation working in the search mode increases the left-hand side by one, but then it increases the right-hand side by one (as β increases by two).

3 Remarks

Let $\langle d_{n-1}, \ldots, d_1, d_0 \rangle$ be a string of bits representing a positive integer, where $d_{n-1} \neq 0$. To distinguish all the possible representations, at least $n - O(1)$ bits are needed by any counter. A binary counter stores the bits and the length of the representation. Thus, the total space usage is $n + O(\lg n)$ bits. Our in-place counter achieves the same space bound. In addition, our counter supports *increment* and *decrement* operations in $O(1)$ worst-case time.

The main ingredients of our counter are: 1) a binary encoding of an integer in the leading $n - 1$ bits of the representation, 2) at most one carry whose position is recalled, 3) the least-significant digit that can take any value up to n, and 4) at most one delayed query "find the next non-zero digit" that is in progress.

We can efficiently support the addition of two of our counters by summing the individual bit-array representations, the two carries, and the values of the two least significant digits for both, and then converting the resulting binary number back to the required form. It is possible to do this conversion such that the properties are maintained. Most importantly, the least-significant digit must be made larger than the number of 0 bits. When implemented this way, an addition requires $O(n)$ bit operations, n being the number of bits of the longer counter. Moreover, for the addition of binary numbers we can rely on word-wise operations so that the addition takes $O(n/w)$ worst-case time on the word RAM, w being the size of the machine word in bits.

A binary counter that supports increments, decrements, and additions is often used in the implementation of mergeable priority queues. For example, a *binomial queue* [17], which is a sequence of heap-ordered binomial trees (for the definition of a binomial tree, we refer to any well-equipped textbook on data structures, e.g. [15, Section 11.4]), relies on a binary counter. A 1 bit at position r in the numeral representation of N, where N is the number of elements stored, means that the data structure contains a binomial tree of 2^r nodes. When a carry is propagated, two binomial trees are joined, and when a borrow is propagated, a binomial tree is split into two; both of these operations can be carried out in $O(1)$ worst-case time on binomial trees. By replacing the ordinary binary counter with our counter, the data structure can support both *insert* and *borrow* operations in $O(1)$ worst-case time, whereas Vuillemin's original implementation supports both of these operations in $\Theta(\lg N)$ worst-case time.

A *run-relaxed heap* [5], which is a sequence of almost heap-ordered binomial trees, uses a counter that bounds the sum of the digits to at most $\lceil \lg(1 + N) \rceil$, where N is the number of elements. A *two-tier relaxed heap* [8] uses the zeroless version of the extended-regular counter [3, 10, 13] to keep track of the trees in the structure (for the description of zeroless counters, see [14, Chapter 9]). In both of these data structures the counters used could be replaced by our counter. This replacement has an interesting trade-off: *borrow* and *delete* operations would become more efficient, but *decrease* and *union* operations would become less efficient. The main reason why we cannot support the last two operations efficiently is that our counter does not support increments of arbitrary digits; it just efficiently supports the increment and decrement of the value by one.

We leave it as an open problem to extend our in-place counter to support a larger operation set efficiently.

References

1. Bose, P., Carmi, P., Jansens, D., Maheshwari, A., Morin, P., Smid, M.H.M.: Improved methods for generating quasi-gray codes. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 224–235. Springer, Heidelberg (2010)
2. Brodal, G.S., Greve1, M., Pandey, V., Rao, S.S.: Integer representations towards efficient counting in the bit probe model. In: Ogihara, M., Tarui, J. (eds.) TAMC 2011. LNCS, vol. 6648, pp. 206–217. Springer, Heidelberg (2011)
3. Clancy, M.J., Knuth, D.E.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Computer Science Department, Stanford University, Stanford (1977)
4. Demaine, E., et al.: Advanced data structures: Lecture 17 (2012), `http://courses.csail.mit.edu/6.851/spring12/scribe/L17.pdf`
5. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Commun. ACM 31(11), 1343–1354 (1988)
6. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Trans. Algorithms 5(1), Acticle 14 (2008)
7. Elmasry, A., Jensen, C., Katajainen, J.: Two new methods for constructing double-ended priority queues from priority queues. Computing 83(4), 193–204 (2008)
8. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Inform. 45(3), 193–210 (2008)
9. Elmasry, A., Jensen, C., Katajainen, J.: Strictly-regular number system and data structures. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 26–37. Springer, Heidelberg (2010)
10. Elmasry, A., Katajainen, J.: Worst-case optimal priority queues via extended regular counters. In: Hirsch, E., Karhumäki, J., Lepistö, A., Prilutskii, M. (eds.) CSR 2012. LNCS, vol. 7353, pp. 130–142. Springer, Heidelberg (2012)
11. Gray, F.: Pulse code communications. U.S. Patent 2632058 (1953)
12. Hagerup, T.: Sorting and searching on the word RAM. In: Morvan, M., Meinel, C., Krob, D. (eds.) STACS 1998. LNCS, vol. 1373, pp. 366–398. Springer, Heidelberg (1998)
13. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and Boolean union-find. In: STOC 2002. pp. 573–582. ACM, New York (2002)
14. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
15. Preiss, B.R.: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc., New York (1999)
16. Rahman, M.Z., Munro, J.I.: Integer representation and counting in the bit probe model. Algorithmica 56(1), 105–127 (2010)
17. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM 21(4), 309–315 (1978)