In-Place Initializable Arrays

Takashi Katoh$^1$ and Keisuke Goto$^1$

$^1$Fujitsu Laboratories Ltd., Kawasaki, Japan., {kato.takashi_01,goto.keisuke}@jp.fujitsu.com

Abstract

Initializing all elements of an array to a specified value is a basic operation that frequently appears in numerous algorithms and programs. Initializable arrays are abstract arrays that support initialization as well as reading and writing of any element of the array in less than linear time proportional to the length of the array. On the word RAM model with $w$ bits word size, we propose an in-place algorithm using only 1 extra bit which implements an initializable array of length $N$ each of whose elements can store $\ell \in O(w)$ bits value, and supports all operations in constant worst case time. We also show that our algorithm is not only time optimal but also space optimal. Our algorithm significantly improves upon the previous best algorithm [Navarro, CSUR 2014] using $N + \ell + o(N)$ extra bits supporting all operations in constant worst case time.

Moreover, for a special cast that $\ell \geq 2\lceil \log N \rceil$ and $\ell \in O(w)$, we also propose an algorithm so that each element of initializable array can store $2^\ell$ normal states and a one optional state, which uses $\ell + \lceil \log N \rceil + 1$ extra bits and supports all operations in constant worst case time.

1 Introduction

On the word RAM model, arrays of fixed length are important data structures that support the fundamental read and write operations of any given element in constant worst case time. Another fundamental operation, called initialization, writes a given initial value to all elements of the array. Initialization appears frequently in numerous algorithms and programs. However, naive implementation for initialization writing an initial value to all elements takes linear time proportional to the length of the array, and this can create a bottleneck in applications that need huge arrays or that require frequent initializations.

Initializable Arrays: An initializable array $Z_{N,\ell}[0 \ldots N-1]$ is an abstract array of length $N$ supporting the following three fundamental operations in $o(N)$ time, where each element of the array can store an individual element of $\ell$ bits. When the context is clear, $Z_{N,\ell}$ is denoted by $Z$ for short, and other arrays are also.

- $\text{iread}(i)$: Return a value stored in the $i$-th element of $Z$.
- $\text{iwrite}(i,v)$: Set the $i$-th element of $Z$ to $v$.
- $\text{iinit}(v)$: Set all elements of $Z$ to $v$.

For ease of explanation, $Z[i]$ and $Z[i] \leftarrow v$ denote $\text{iread}(i)$ and $\text{iwrite}(i,v)$, respectively. Initializable arrays enhances normal arrays supporting $\text{iread}$ and $\text{iwrite}$ in constant worst case time, and $\text{iinit}$ in $\Theta(N)$ time. An abstract array means that we do not actually have to write a value to all elements in a normal array for initialization; we only have to behave as if we do that. Namely, when reading an $i$-th element, we simply have to return the value related to the most recent of initialize or write operations for the $i$-th element.

We assume the word RAM model with $w = \Omega(\log N)$ bits word size that all usual arithmetic (including multiplications) and bitwise operations on constant number of words take constant
worst case time, and we also assume that $\ell \in O(w)$. We focus on and evaluate the additional extra space over $N\ell$ bits because $Z$ is an extension of a normal array $A_{N,\ell}$, and both requires at least $N\ell$ bits. Moreover, we account only dynamic values for the space of algorithms, e.g., the space for an initial value or writable auxiliary arrays. On the other hand, we do not account static values that can be embedded into a program, e.g., the space for the length of the array or some static parameters of algorithms.

Initializable arrays have been studied for last four decades. A folklore algorithm supporting all operations in constant worst case time was first mentioned (but not described) in a work by Aho et al. [1, Ex. 2.12]. The complete description later appeared in works by Mehlhorn [10, Sec. III.8.1] and Bentley [2, Column 1]. The folklore algorithm manages written values of $Z$ after the last initialization by using two auxiliary arrays of length $N$ whose elements can store $\lceil \log N \rceil$ bits and two variables of $\ell$ bits and $\lceil \log N \rceil$ bits, respectively. It therefore requires $2N\lceil \log N \rceil + \ell + \lceil \log N \rceil$ extra bits. Navarro [13,14] reduced the space to $N + \ell + o(N)$ extra bits without increasing the time complexities. His algorithm combined the folklore algorithm with a bitmap technique using a bit array $B_{N,1}$ such that $B[i]$ is 1 if and only if the $i$-th element of the array has been written from the last initialization. Each runtime of an algorithm depends on the access frequency to the array, where the access frequency is the ratio of the number of read and write operations to the array length. Fredriksson and Kılıçalan measured the runtime performances of several algorithms [5]. According to their computational experiments, the folklore algorithm and Navarro’s algorithm are the most efficient when the access frequency is low (below 1%), while the bitmap solution and naive solution are the most efficient when the access frequency is within 1–10% and over 10%, respectively.

The construction of ZDD [11] is a good example that initializable arrays work effective. ZDD is a space efficient data structure to represent any family sets, and widely used for various practical applications [12,16]. Knuth used the folklore algorithm for the implementation of a hash table in the fast ZDD construction algorithm Simpath [7,8]. In each step of the algorithm, a hash table is used for representing millions of nodes, and it is initialized every time before each step, so the initializable array works effective.

The previous best efficient Navarro’s algorithm still requires linear extra bits, and we may not afford the cost when we need to treat huge arrays or we have only limited resources. Reducing the space while keeping the better time complexity is interested in both theoretical and practical aspects. The ultimate goal is to develop an in-place algorithm supporting all operations in constant worst case time, where in-place means that the algorithm uses only constant number of extra words.

**Our Contributions:** Our contributions are summarized below:

**In-place initializable arrays.** We propose a novel in-place chain technique which is almost same as the folklore algorithm but works in-place. By using the technique, we show that an initializable array supporting all operations in constant worst case time can be implemented with extra 1 bit. We analyze the lower bound of the space complexity in the next contribution, and we conclude that our algorithm runs in optimal time and space. Moreover, our algorithm is quite simple so that the pseudo code of the core idea in our algorithm is written within 90 lines in Algorithm 1–3.

**Lower bound of the time complexity for initialization without extra space.** We analyze the time and space complexities for initializable arrays without extra space, and found that initialization without extra space takes $\Omega(N)$ worst-case/amortized/expected times. These are strong result to indicate the difficulty to implement initializable arrays without extra space.

---

1 $\ell$ bits for storing an initial value are not accounted for the analysis of the original paper, but it is accounted in this paper.
Table 1: Comparison of the time and space complexities between the previous works, recent independent ones, and ours. Only \texttt{iwrite} by Loong et al. (bold) takes amortized time or worst case expected time, and the others take worst case time.

| Algorithms                      | Extra bits | iinit | iread | iwrite |
|---------------------------------|------------|-------|-------|--------|
| Normal array                    | 0          | \(\Theta(N)\) | \(O(1)\) | \(O(1)\) |
| Folklore \cite{1,2,10}          | \(2N\lceil\log N\rceil + \ell + \lceil\log N\rceil\) | \(O(1)\) | \(O(1)\) | \(O(1)\) |
| Navarro \cite{13,14}           | \(N + o(N)\) | \(O(1)\) | \(O(1)\) | \(O(1)\) |
| Hagerup and Kammer \cite{6}     | 1          | \(O(1)\) | \(O(\log N)\) | \(O(\log N)\) |
| Loong et al. \cite{9}           | 1          | \(O(1)\) | \(O(1)\) | \(O(1)\) |
| This paper                     | 1          | \(O(1)\) | \(O(1)\) | \(O(1)\) |

In-place initializable optional arrays for a special case. Many program languages provide the library to append a optional state for a variable of \(\ell\) bits so that it can represents \(2^\ell + 1\) states, e.g., \texttt{std::optional} in C++ \cite{3} and \texttt{Optional} in Java \cite{15}. Optional arrays \(Z_{N,\ell}\) is an array each of whose elements can store \(2^\ell + 1\) states, and which can be trivially implemented using \(N(\ell+1)\) bits. However, the space complexity is far from the theoretical lower bound \(\lceil N\log(2^\ell+1)\rceil\) bits. The optimal solution using \(\lceil N\log k\rceil\) bits has already proposed by Dodis et al. \cite{4} in the more general settings that implements an optional array of length \(N\) whose each element has any \(k\) states, but it takes initialization in \(\Theta(N)\) time.

For a special cast that \(\ell \geq 2\lceil\log N\rceil\) and \(\ell \in O(w)\), we extend our algorithm in Section 3 and propose in-place initializable optional arrays so that each element can store \(2^\ell\) normal states and a one optional state using \(\ell + \lceil\log N\rceil + 1\) extra bits. Our algorithm supports all operations in constant worst case time.

Related Works: Very recently and independently, Hagerup and Kammer \cite{6}, and Loong et al. \cite{9} also proposed in-place algorithms for initializable arrays \cite{2}. Hagerup and Kammer’s algorithm supports read/write in \(O(t)\) worst case time, initialization in constant worst case time using extra \(\lceil N/(w/ct)^t\rceil\) bits, where \(c\) is a constant value greater than 1, and \(t\) is a time and space trade off parameter within \(1 \leq t \leq \lceil \log N \rceil\). We obtain in-place algorithm using 1 extra bit by setting \(t = \lceil \log N \rceil\), but read/write operations takes \(O(\log N)\) worst case time. They also analyzed the time and space complexities for initialization without extra space, and proved the lower bound of the worst case time, but they did not analyze the amortized and expected time. Loong et al. proposed two algorithms, both uses 1 extra bit and supports read/initialization in constant worst case time, and for write, one of which takes amortized constant time and the other takes constant worst case expected time. Compared to their algorithms, our algorithm is quite simple and runs in optimal time and space. See also Table 1.

Organizations: The rest of the paper is organized as follows. Section 2 introduces the folklore algorithm which our algorithm is based on. Section 3 proposes an algorithm using \(\ell + \lceil\log N\rceil\) extra bits for a special case that \(N\) is even, \(\ell \geq \lceil\log N\rceil\), and \(\ell \in O(w)\). Section 4 proposes an algorithm using 1 extra bit for the more general case. Section 5 analyzes the time and space complexities for initializable arrays without extra space. Section 6 proposes a simple implementation of initializable optional arrays using the technique proposed in Section 3. Section 7 gives the summary and some future works.

2 Folklore Algorithm

The folklore algorithm implements an initializable array \(Z_{N,\ell}\) supporting all operations in constant worst case time by using three normal arrays of length \(N\), \(V_{N,\ell}\), \(F_{N,\lceil\log N\rceil}\), and \(T_{N,\lceil\log N\rceil}\)\footnote{Interestingly, their and our papers were firstly appeared at \url{https://arxiv.org} in the same week.}.
and by using two variables, an initial value \( \text{initv} \) of \( \ell \) bits and a stack pointer \( b \) of \( \lceil \log N \rceil \) bits, so it requires \( 2N\lceil \log N \rceil + \ell + \lceil \log N \rceil \) extra bits. \( \text{initv} \) stores an initial value, \( T \) is used as a stack, and \( b \) indicates the stack size of \( T \). We say that \( F[i] \) and \( T[j] \) are chained when they are linked to each other, namely, \( F[i] = j \), \( T[j] = i \), and \( j < b \). \( V[i] \) stores a written value, and we maintain the invariant that \( Z[i] = V[i] \) if \( F[i] \) is chained, and \( Z[i] = \text{initv} \) otherwise.

The folklore algorithm implements the operations of \( Z \) using the invariant as follows:

\[ \text{ired}(i) \]: Return \( V[i] \) if \( F[i] \) is chained, and \( \text{initv} \) otherwise.

\[ \text{fwrite}(i, v) \]: Set \( V[i] \) to \( v \), and if \( F[i] \) is unchained, create a new chain between \( F[i] \) and \( T[b] \), namely, \( T[b] \leftarrow i \), \( F[i] \leftarrow b \), and \( b \leftarrow b + 1 \).

\[ \text{init}(v) \]: Break all chains by setting \( b \) to zero, and set \( \text{initv} \) to a new initial value \( v \).

\( \text{ired} \) is trivially obtained from the invariant. \( \text{fwrite} \) creates a new chain of \( F \) and \( T \) only when an element is written for the first time, and thus the number of chains is at most \( N \), and the chain will never be broken until \( \text{init} \) is called. \( \text{init} \) breaks all chains by setting \( b \) to zero, and thus it implies that all elements of \( Z \) are initialized by a new value \( \text{initv} \). Each operation takes constant worst case time. In this way, the folklore algorithm maintains the invariant and implements an initializable array \( Z \) using \( 2N\lceil \log N \rceil + \ell + \lceil \log N \rceil \) extra bits, and supports all operations in constant worst case time.

Although not mentioned in the previous works, we can easily reduce the space of the folklore algorithm to \( 2\lceil N/c \rceil \lceil \log N \rceil + \ell + \lceil \log N \rceil \) extra bits for a constant number \( c \geq 1 \). We use this technique also for our algorithm in Section 3. We split \( V_{N,\ell} \) into \( \lceil N/c \rceil \) blocks, and reduce the size of \( F_{N,\lceil \log N \rceil} \) and \( T_{N,\lceil \log N \rceil} \) to \( F_{\lceil N/c \rceil,\lceil \log N \rceil} \) and \( T_{\lceil N/c \rceil,\lceil \log N \rceil} \), respectively. An initializable array \( Z_{N,\ell} \) consists of an initializable array \( X_{\lceil N/c \rceil,\ell} = Z[0 \ldots N - (N \mod c) - 1] \) and a normal array \( Y_{N \mod c, \ell} = V[N - (N \mod c) \ldots N - 1] = Z[N - (N \mod c) \ldots N - 1] \), and it is implemented by initializing two arrays for \( \text{init} \), and reading and writing the corresponding array for \( \text{ired} \) and \( \text{fwrite} \), respectively. Since \( Y \) is a normal array of length \( c \), \( \text{init} \) takes \( O(c) \) worst case time, and \( \text{ired} \) and \( \text{fwrite} \) take constant worst case time. \( X \) can be implemented using \( V[0 \ldots N - (N \mod c) - 1] \), \( F \), \( T \), \( \text{initv} \), and \( b \) with \( 2\lceil N/c \rceil \lceil \log N \rceil + \ell + \lceil \log N \rceil \) extra bits. Let \( B_i = V[(i - 1)c + 1 \ldots i + 1] \) be the \( i \)-th block. We manage invariants that all elements of the block \( B_i \) have been written if \( F[i] \) is chained. When writing an element \( V[i] \) of a block \( B_i' \) (\( i' = [i/c] \)) for the first time, we initialize \( B_{i'} \) by setting all \( c \) elements of \( B_i' \) to initial values in \( O(c) \) time, and write \( V[i] \) with a given value, and make chain \( F[i'] \). \( \text{init} \) and \( \text{fwrite} \) can be done in constant worst case time in the same way as the folklore algorithm. Each operation of \( Z_{N,\ell} \) can be done by the constant number of operations for \( X \) and \( Y \). In total, we have an initializable array \( Z \) supporting \( \text{ired} \) in constant worst case time, and supporting \( \text{fwrite} \) and \( \text{init} \) in \( O(c) \) worst case time with \( 2\lceil \ell/c \rceil \lceil \log N \rceil + \ell + \lceil \log N \rceil \) extra bits. Note that \( \text{fwrite} \) and \( \text{init} \) also take constant worst case time when \( c \) is constant.

3 In-Place Initializable Arrays for a Special Case

Our algorithm implements an initializable array \( Z_{N,\ell} \) by using only one normal array \( A_{N,\ell} \) and by using two variables, an initial value \( \text{initv} \) of \( \ell \) bits and a stack pointer \( b \) of \( \lceil \log N \rceil \) bits, so it requires \( \ell + \lceil \log N \rceil \) extra bits (that is in-place). In this section, we assume a special case that \( N \) is even, \( \ell \geq \lceil \log N \rceil \), and \( \ell \in O(w) \), and we consider the more general case in Section 4.

The main concept underlying our algorithm is almost the same as that of the folklore algorithm. Our algorithm also uses \( V \), \( F \), and \( T \), but embeds them into \( A \) sparsely. This idea intuitively seems impossible because all \( 3N \) elements of \( V \), \( F \), and \( T \) are required in the worst case in the folklore algorithm, and thus \( A \) of length \( N \) cannot store all of them. To avoid this problem, we reduce the number of chains. First, we split \( A \) into \( N/2 \) blocks each of whose blocks contains two elements in a similar way to the space saving technique in Section 2. Second,
we also split \( A \) into two areas: the unwritten chained area (UCA) \( A[0 \ldots 2b - 1] \) and the written chained area (WCA) \( A[2b \ldots N - 1] \), and chain blocks in UCA to blocks in WCA. The key tricks are that a block in UCA is chained if and only if the elements of the block has not been written from the last initialization (we call such block an \( \text{unchained block} \)), and that a block in WCA is chained if and only if the elements of the block has been written from the last initialization.

This idea comes from the important observation that if we manage written elements with chains (like the folklore algorithm), we need only a few chains at the beginning after an initialization, but this increases gradually and eventually reaches to \( N \). On the other hand, if we manage the unwritten elements with chains, we need a few chains at the beginning after an initialization and we need roughly \( N \) chains at the beginning. Here, the threshold of the areas \( 2b \) is set to the position that the number of chains is the smallest, namely, the number of unwritten blocks in UCA and the number of written blocks in WCA are equalized. When increasing \( b \), the number of unwritten blocks in UCA is increasing, and on the other hand, the number of written blocks in WCA is decreasing. Therefore, such threshold position \( 2b \) must uniquely exist.

Let \( B_i = A[2i \ldots 2i + 1] \) be the \( i \)-th block. Each \( A[i] \) belongs to the block \( B_{i/2} \) and any block must contain two elements since \( N \) is even. We say that blocks \( B_i \) and \( B_j \) are chained if \( A[2i] = 2j \) and \( A[2j] = 2i \) and neither of blocks is in the same area. Note that any element can store any index of \( A \) since \( \ell \geq \lceil \log N \rceil \).

In our algorithm, we maintain the following four invariants, where \( V[i] \), \( F[i] \), and \( T[i] \) respectively represent the functional aspects of \( A[i] \) as in the folklore algorithm. See also Figure 1.

1. A block \( B_{i1} \) in UCA is a written block. \( \Leftrightarrow \) \( B_{i1} \) in UCA is unchained. \( \Rightarrow \) \((A[2i1], A[2i1 + 1]) = (V[2i1], V[2i1 + 1]), \) and \((Z[2i1], Z[2i1 + 1]) = (V[2i1], V[2i1 + 1])\).
2. A block \( B_{i2} \) in UCA is an unwritten block. \( \Leftrightarrow \) \( B_{i2} \) in UCA is chained with a block \( B_{i3} \) in WCA. \( \Rightarrow \) \((A[2i2], A[2i2 + 1]) = (T[2i2], V[2i2]), \) and \((Z[2i2], Z[2i2 + 1]) = (initv, initv)\).
3. A block \( B_{i3} \) in WCA is a written block. \( \Leftrightarrow \) \( B_{i3} \) in WCA is chained with a block \( B_{i4} \) in UCA. \( \Rightarrow \) \((A[2i3], A[2i3 + 1]) = (F[2i3], V[2i3 + 1]), \) and \((Z[2i3], Z[2i3 + 1]) = (V[2i3], V[2i3 + 1] = A[2i3 + 1])\).
4. A block \( B_{i4} \) in WCA is an unwritten block. \( \Leftrightarrow \) \( B_{i4} \) in WCA is unchained. \( \Rightarrow \) \((A[2i4], A[2i4 + 1]) \) are any values but \( A[2i4] \) must not be chained to any value \( A[2j] \) in UCA, and \((Z[2i4], Z[2i4 + 1]) = (initv, initv)\).

\( \text{iread} \) is trivially implemented from the invariants. \( \text{iinit} \) is implemented in the same way as the folklore algorithm by setting \( b \) and \( \text{initv} \) to zero and a given initial value, respectively. The pseudo codes of \( \text{iinit} \) and \( \text{iread} \) are described in Algorithm 2 in Appendix. \( \text{iwrite} \) is more complicated than \( \text{iread} \) and \( \text{iinit} \) since it may break the invariants by writing a new

---

**Figure 1:** Four blocks chained or unchained in unwritten chained area (UCA) or written chained area (WCA) in \( A \) and \( Z \). Bold borders indicate blocks. Blocks \( B_{i2} \) and \( B_{i3} \) are chained since they are in the different areas, \( A[2i3] = F[2i3] = 2i2 \) and \( A[2i2] = T[2i2] = 2i3 \).
value. We use the following tools\footnote{Some of these functions take and return blocks as their arguments and outputs, respectively. Actual implementations treat such blocks as pointers, so copy and comparison of constant number of blocks take constant worst case time. However, in our pseudo codes, we represent a block $B_i$ as just $B_i$, instead of a pointer $i$ to emphasize that we are indicating a block.} to implement $\texttt{write}$ which are described in Algorithm\footnote{Implementations treat such blocks as pointers, so copy and comparison of constant number of blocks take constant worst case time. However, in our pseudo codes, we represent a block $B_i$ as just $B_i$, instead of a pointer $i$ to emphasize that we are indicating a block.} in Appendix.

- **chainedTo($B_i$)**: Return the block chained to $B_i$ if $B_i$ is chained, and return a symbol $\texttt{None}$ otherwise.

- **makeChain($B_i, B_j$)**: Make a new chain between $B_i$ in UCA and $B_j$ in WCA.

- **breakChain($B_i$)**: Break the chain of the block $B_i$ in UCA.

- **initBlock($B_i$)**: Initialize the block $B_i$ with initv, namely, write initv to $A[2i]$ and $A[2i + 1]$.

- **extend()**: Extend UCA by one block and return an unwritten block in UCA that has not been chained yet and that is initialized with ($initv, initv$).

$\texttt{chainedTo}$, $\texttt{makeChain}$, and $\texttt{initBlock}$ are simply implemented from their functional aspects. $\texttt{breakChain}$ breaks an unexpected chain between $B_i$ and $B_k$ by rewriting $A[2k]$ with itself $2k$. $\texttt{extend}$ tries to get a new block $B_{b-1}$ that turns out to be in UCA after extending this area by $b \leftarrow b + 1$. If the block $B_{b-1}$ is unchained before extending, it is initialized with initv. This initialization may make a chain between $B_{b-1}$ in UCA and a block $B_{initv/2}$ in WCA accidentally, that is, initv is even, initv $\geq 2b$, $A[2(b - 1)] = initv$, and $A[initv] = 2(b - 1)$. In such case, we call $\texttt{breakChain}(B_{b-1})$, and return $B_{b-1}$. If the block $B_{b-1}$ is chained with a block $B_k$ in UCA before extending, we cannot initialize $B_{b-1}$ because the information for $Z[2(b - 1)]$ and $Z[2(b - 1) + 1]$ will be lost. In this case, we simply write $Z[2(b - 1)]$ and $Z[2(b - 1) + 1]$ into $A[2(b - 1)]$ and $A[2(b - 1) + 1]$, and return $B_k$ initialized with initv.

$\texttt{write}$ is described in Algorithm\footnote{Implementations treat such blocks as pointers, so copy and comparison of constant number of blocks take constant worst case time. However, in our pseudo codes, we represent a block $B_i$ as just $B_i$, instead of a pointer $i$ to emphasize that we are indicating a block.} When $\texttt{write}(i, v)$ is called, let $'v = \lfloor i/2 \rfloor$, there are four major conditions, and we write $v$ while managing the invariants in each state as follows.

- In Lines 3–7 $B_{i'}$ is unchained in UCA, so it is a written block. $Z[i]$ has already been written, so we simply rewrite it with a new value $v$. We expect $B_{i'}$ is unchained from the invariant, but $B_{i'}$ may turn out to be chained accidentally by writing $v$ to $A[i]$. We call $\texttt{breakChain}(B_{i'})$ to break such a chain.

- In Lines 9–10 $B_{i'}$ is chained in UCA, so it is an unwritten block. Since $B_{i'}$ is not chained, we do not have enough space for storing $v$. To circumvent this, we extend UCA, obtain an unwritten block $B_j$ in UCA that has not been chained yet, exchange it with $B_{i'}$, and write $v$ to $A[i]$ in the block $B_{i'}$. There are considerable points as follows, (1) $B_{i'}$ may be equal to $B_j$ before exchanging. (2) $B_{i'}$ may turn out to be chained accidentally by writing $v$ to $A[i]$, which is the same situation in Lines 6–7. For case 1, we do not exchange $B_j$ with $B_{i'}$, and simply write $v$ to $A[i]$. For case 2, we break the chain by $\texttt{breakChain}(B_{i'})$.

- In Lines 19–22 $B_{i'}$ is in WCA and chained to a block $B_k$ in UCA, so it is a written block. $Z[i]$ has already been written, hence we simply write $v$ in the corresponding position $A[i]$ or $A[2k + 1]$.

- In Lines 23–30 $B_{i'}$ is unchained in WCA, so it is an unwritten block. $Z[i]$ has been unwritten, so we have to make a chain between the block $B_{i'}$ and a block in UCA. We extend UCA and obtain a new initialized block $B_k$ in UCA. If $B_{i'} = B_k$, $B_{i'}$ has turned out to be located in UCA. It is the same situation in the Lines 6–7 so we do
Algorithm 1: iwrite(i, v)

1 Function iwrite(i, v):
2     i' ← ⌊i/2⌋  // B_i' is a block which contains A[i].
3     B_k ← chainedTo(B_i);
4     if i' < b then
5         if B_k = None then
6             // B_i' is a written block in UCA.
7             A[i] ← v;
8             breakChain(B_i');
9         else
10            // B_i' is an unwritten block in UCA.
11            B_j ← extend();
12            if B_i' = B_j then
13                // Now, it is the same situation of just before Line 6.
14                We perform the same procedure as in Lines 6–7;
15            else
16                // Exchange B_i' with B_j
17                (A[2j], A[2j + 1]) ← (A[2i'], A[2i' + 1]);
18                makeChain(B_j, B_k);
19                initBlock(B_i');
20                // Now, it is the same situation of just before Line 6.
21                We perform the same procedure as in Lines 6–7;
22            else
23                if B_k ≠ None then
24                    // B_i' is written block in WCA.
25                    if i mod 2 = 0 then
26                        A[2k + 1] ← v  // Write v to the second element of B_k
27                    else
28                        A[i] ← v  // Write v to the second element of B_i'
29                else
30                    // B_i' is an unwritten block in WCA.
31                    B_k ← extend();
32                    if B_i' = B_k then
33                        // Now, it is the same situation of just before Line 6.
34                        We perform the same procedure as in Lines 6–7;
35                    else
36                        initBlock(B_i');
37                        makeChain(B_k, B_i');
38                        // Now, it is the same situation of just before Line 19.
39                        We perform the same procedure as in Lines 19–22;
the same procedure. Otherwise, we initialize the block \( B_k \) and make the chain between \( B_k \) and \( B_k' \). Here, it is the same situation in Lines 19, 22, so we do the same procedure.

Roughly speaking, our algorithm extends UCA (suppressing WCA) by increasing \( b \) by one when writing a new value. This is similar to how a normal array initializes itself by writing a value from left to right. Our algorithm does the same thing, but writes only two values when increasing \( b \). In the extreme case that \( 2b = N \), all elements have already been written, and the contents of \( A \) are exactly the same as those of a normal array, that is, \( Z[i] \) is stored at \( A[i] \).

In this way, our algorithm maintains the invariants for writing steps, and each operation takes constant worst case time. Therefore, we have the following lemma.

**Lemma 1.** There is an initializable array \( Z_{N, \ell} \) supporting all operations in constant worst case time for even number \( N \), \( \ell \geq \lceil \log N \rceil \) and \( \ell \in O(w) \) using \( \ell + \lceil \log N \rceil \) extra bits.

4 In-Place Initializable Arrays for a General Case

In the previous section, we showed that there is an implementation of initializable array \( Z_{N, \ell} \) for the special case that \( N \) is even, \( \ell \geq \lceil \log N \rceil \), and \( \ell \in O(w) \). In this section, we describe how to implement an initializable array \( Z_{N, \ell} \) for a general \( N \) and \( \ell \in O(w) \) in the same space \( \ell + \lceil \log N \rceil \) extra bits. Moreover, we also show that we can reduce the space to just 1 extra bit.

The requirement that \( N \) is even and \( \ell \geq \lceil \log N \rceil \) can be removed, and we can implement an initializable array \( Z_{N, \ell} \) for a general \( N \) and \( \ell \in O(w) \). It is not trivial since an \( \ell < \lceil \log N \rceil \) bits element cannot store a pointer to a position in an array of length \( N \). We pack \( p \) elements of \( \ell \) bits each into a word, and implement an initializable array \( Z_{N, \ell} \) by using an initializable array \( X_{\lceil N/p \rceil, \ell} \) and a normal array \( Y_{N \mod p, \ell} \).

**Lemma 2.** There is an initializable array \( Z_{N, \ell} \) for \( \ell \in O(w) \) using \( \ell + \lceil \log N \rceil \) extra bits which supports all operations in constant worst case time.

**Proof.** Let \( p = \lceil \frac{\log N}{\ell} \rceil \), \( N' = \lfloor \frac{N}{p} \rfloor \), and \( \ell' = p \ell \). \( Z_{N, \ell} \) can be implemented with an initializable array \( X_{N', \ell'} \) and a normal array \( Y_{c, \ell} \), where \( c = N \mod p = N - pN' \). Since \( \frac{\log N}{\ell} \leq p \leq \frac{\log N}{\ell} + 1 \), we have \( \ell' \geq \lceil \log N \rceil \), \( \ell' \leq \lceil \log N \rceil + \ell \in O(w) \), and \( c = (N - pN') \ell = (N - p) \lceil \frac{N}{p} \rceil \ell \leq (N - p(N' - 1)) \ell = p \ell = \ell' \in O(w) \). Let \( \text{initv} \) be an \( \ell \) bits initial value for \( Z_{N, \ell} \), and \( \text{initv}' \) be an \( \ell' \) bits initial value for \( X_{N', \ell'} \). By Lemma 1, an initializable array \( X_{N', \ell'} \) can be implemented using \( \ell' + \lceil \log N \rceil \) extra bits, where the first term \( \ell' \) bits are for storing an initial value \( \text{initv}' \), and the second term \( \lceil \log N \rceil \) bits are for storing a stack pointer \( b \). \( \text{initv}' \) consists of \( p \) initial values \( \text{initv} \), and it is obtained by the bit repeat operation \( 5 \) that copies \( \text{initv} \) at \( p \) times in constant worst case time. Therefore, we can reduce the space to \( \ell + \lceil \log N \rceil \) extra bits by storing \( \text{initv} \) instead of storing \( \text{initv}' \). Since \( Y_{c, \ell} \) is a normal array of \( O(w) \) bits, \( \text{iinit} \) takes constant worst case time by the bit repeat operation, and \( \text{iread} \) and \( \text{iwrite} \) also take constant worst case time. By combining \( X_{N', \ell'} \) and \( Y_{c, \ell} \), \( Z_{N, \ell} \) can be implemented using \( \ell + \lceil \log N \rceil \) extra bits.

Finally, we prove that we can reduce the \( \ell + \lceil \log N \rceil \) extra bits for an initializable array \( Z_{N, \ell} \) for \( \ell \in O(w) \) into just 1 extra bit.

**Theorem 1.** There is an initializable array \( Z_{N, \ell} \) for \( \ell \in O(w) \) using 1 extra bit which supports all operations in constant worst case time.

\(^5\)The bit repeat can be implemented by multiplication. We use a precomputed static bit pattern of length \( \ell' \) bits that each \( \ell \)-th bit from left is 1 and others are 0, and embed it in the program. We can obtain \( \text{initv}' \) by multiplying the bit pattern with \( \text{initv} \) in constant worst case time.
Figure 2: Initializable array \( X_{N', \ell'} \) with 1 extra bit implemented on a normal array \( A_{N', \ell'} \). The top represents \( A \) and flag when \( 2b < N' \), and the bottom represents \( A \) and flag when \( 2b = N' \). If flag = 0, the first element of the last block \( B_i \) contains three values, initial value \( \text{initv} \), stack pointer \( b \), and pointer to the block which is chained to \( B_i \). The front space colored in gray of chained blocks in UCA is unused. Note that \( B_i \) may be unchained. If flag = 1, all values of \( X \) have been written, and it can be seen as a normal array.

Proof. In a similar way to Lemma 2, an initializable array \( Z_{N, \ell} \) using 1 extra bit can be implemented by using an initializable array \( X_{N', \ell'} \) and a normal array \( Y_{c, \ell} \). However, we use the different sizes for \( N' \) and \( \ell' \) to embed an initial value \( \text{initv} \) and a stack pointer \( b \) into the first \( \ell' \) bits element of the last block of \( X \).

Let \( p = 2^{\lceil \frac{\log N}{\ell} \rceil} + 1, N' = \lceil \frac{N}{p} \rceil, \) and \( \ell' = p\ell \). Since \( 2^{\lceil \frac{\log N}{\ell} \rceil} + 1 \leq p \leq 2(\lceil \frac{\log N}{\ell} \rceil + 1) + 1 = 2^{\frac{\log N}{\ell}} + 3, \) we have \( \ell' \geq 2\lceil \log N \rceil + \ell \) and \( \ell' \leq 2\lceil \log N \rceil + 3\ell \in O(w) \). Since \( c\ell \in O(w), Y_{c, \ell} \) can be initialized in constant worst case time in a similar way of Lemma 2. Next, we focus on how to implement \( X \). The first element of a block in WCA stores a pointer for a chain but does not store written values. This implies that the first element of a block in WCA can afford to store information of \( \lceil \log N \rceil + \ell \) bits in addition to a pointer for chain. We embed an initial value \( \text{initv} \) and a stack pointer \( b \) into the first element of the last block, and we can access the last block and obtain these values in constant worst case time. However, they are overwritten when \( 2b = N' \). We have only 1 extra bit for flag which is 1 if and only if \( 2b = N' \). Let \( A_{N', \ell'} \) be a normal array that \( X \) is implemented on. If flag = 0, \( A \) behave in the same way as in Section 3.

If flag = 1, all elements of \( X \) have already been written and it is equivalent to a normal array \( A \) such that each \( A[i] \) stores \( X[i] \). In this case, \( \text{initv} \) and \( b \) are overwritten with some values, but this is not a problem because they are no longer used since \( X \) is now equal to a normal array \( A \). See Figure 2.

In this way, \( X \) can be implemented using 1 extra bit, and thus, \( Z \) is also implemented using 1 extra bit.

5 Lower Bound for Initializable Arrays Without Extra Space

A natural question is whether it is possible to implement an initializable array without extra bit. Theorem 2 below gives a negative answer to this question.

Theorem 2. An initialization on any initializable array \( Z_{N, \ell} \) without extra space takes \( \Omega(N) \) worst-case/amortized/expected time.

Proof. We assume a simple case that there is an initializable array \( Z \) implemented on a normal array \( A_{N, \ell} \) without extra space such that \( \ell = w \). Let \( S = \{ x \mid 0 \leq x < 2^w \}^N \) be a universal set of word sequences of length \( N \), and \( x[i] \) for \( x \in S \) denotes an \( i \)-th element of the sequence \( x \). Since \( Z \) can store any sequence of \( S \) and the bit-length of \( Z \) is exactly equal to the bit-length of \( A \), there exists a one-to-one mapping \( f : S \to S \) such that \( Z = x \) and \( A = f(x) \) for any \( x \in S \).
We begin with explaining any initialization for any state of $Z$ requires at least $N$ times access to $A$. We assume there exists a sequence $x_0 \in S$ that $\text{iinit}(v)$ changes $Z$ from $x_0$ to $x_0' = v, \ldots, v$, and simultaneously changes $A$ from $y_0 = f(x_0)$ to $y_0' = f(x_0')$ with accessing (reading and writing) to $A$ strictly less than $N$ times. It implies that there exists an index $0 \leq p < N$ of $A$ such that $\text{iinit}$ does not access to $A[p]$. Let $y_1 \in S$ be a sequence such that $y_1[i] = y_0[i](0 \leq i < p \text{ or } p < i < N)$ and $y_1[p] \neq y_0[p]$. For $A = y_1$, $\text{iinit}(v)$ changes $A = y_1$ to $A = y_1'$ in the same way as it changes $A = y_0$ to $A = y_0'$ since $y_1$ and $y_0$ are same without a position $p$ which is not accessed for $\text{iinit}(v)$ for $A = y_0$. We have $y_1'[p] = y_1[p] \neq y_1[p] = y_0'[p]$ since $y_1[p]$ and $y_0[p]$ were not accessed in $\text{iinit}(v)$, but it contradicts the definition of $\text{iinit}$ that changes $y_0$ and $y_1$ to $y_0' = y_1' = f(x_0')$. Therefore, we need at least $N$ access to $A$ for initializing any state of $A$.

Let $Q$ be a sequence of $M \geq 2N$ operations for $Z$ including (i) $cM \text{iinit}$ operations and (ii) $N$ consecutive $\text{write}$ operations preceding to each $\text{iinit}$ operation, where $0 < c < 1$ is any constant. By the condition (ii), $A$ has any sequence of $S$ just before $\text{iinit}$ is called, so $\text{iinit}$ takes $\Omega(N)$ worst case time and also expected time. Next, we analyze the amortized cost. By the condition (i), the initializable array accesses to $A$ at least $cMN$ times in total. Summing up, $\text{iinit}$ takes $cNM/M = cN \in \Omega(N)$ amortized time.

In this way, an initialization on any initializable array $Z$ without extra space takes $\Omega(N)$ worst-case/amortized/expected time.

Our implementation of an initializable array with 1 extra bit can represent $2^{N\ell+1}$ states, and maps multiple states of $A$ to a state of $Z$. Hence, it can change any state of $A$ to a state of $A$ mapped to an initialized state of $Z$ in constant worst case time.

From Theorem 1 and 2, we have the following corollary.

**Corollary 1.** An initializable array $Z_{N,\ell}$ using 1 extra bit for $\ell \in O(w)$ is space optimal.

### 6 In-Place Initializable Optional Arrays for a Special Case

In this section, we extend the algorithm in Section 3 and propose an in-place initializable optional arrays $Z_{N,\ell}$ using $\ell + \lceil \log N \rceil + 1$ extra bits for a special case that $\ell \geq 2\lceil \log N \rceil$ and $\ell \in O(w)$. Each element of $Z$ can store $2^\ell$ normal states and one optional state, $Z$ can write and initialize with a normal state or an optional state, and all operations $\text{read, write, and iinit}$ take constant worst case time.

A technical issue is how to store an optional state in space less than $N$ bits which is a trivial solution. We can solve the issue by using the data structure in Section 3. We regard that an unwritten block block on the data structure in Section 3 represents an optional state. We first implement an initializable optional array $X_{N,\ell}$ that can initialize only with an optional state. For simplicity, we assume $\ell = 2\lceil \log N \rceil$, but it can be extended to the more general case $\ell \geq 2\lceil \log N \rceil$ and $\ell \in O(w)$. Let $\ell' = \lceil \log N \rceil$ be half of $\ell$, and $A_{2N,\ell'}$ be a normal array to implement an initializable optional array $X$. We split $A$ into two areas optional chained area (OCA) $A[0 \ldots 2b - 1]$ and unoptional chained area (UCA) $A[2b \ldots 2N - 1]$, and also split $A$ into blocks each of which contains two elements. A threshold $2b$ is set to a position that the number of optional elements in OCA and the number of unoptional elements in UCA are equal, and it must be unique as well as that $2b$ in Section 3 is unique. Any state of $X$ can be represented by using in-place chain technique, see Figure 3.

Reading a state of each element is trivial from the invariants and initialization can be done in the same way to Section 3 but writing with a normal state or an optional state is not trivial. Changing a normal state to a normal state is trivial since we just rewrite corresponding elements determined by the invariants. Therefore, we consider the case changing a normal state to an optional state and the reverse case, which is equivalent to change any block on $A$ to be chained and unchained. When making a block in UCA chained, we get an optional block in OCA which
has not been chained yet, and make a chain between them. Similarly, when making a block in OCA chained, we get an unoptional block in UCA which has not been chained yet, and make a chain between them. To manage the invariants, we need two functions: (i) extend OCA and get an optional block in OCA which must be chained but not yet, (ii) shrink OCA and get an unoptional block in UCA which must be chained but not yet. We already have the former one as extend, so we propose the latter one shrink described in Algorithm 3 in Appendix. shrink is implemented in almost same as extend. We have already shown how to change chained block in OCA to be unchained, and unchained block in UCA to be chained using extend, which are case (2) and (4) in Figure 3 respectively. In the same way, we can change unchained block in OCA to be chained, and chained block in UCA to be unchained using shrink, which are case (1) and (3) in Figure 3 respectively. In this way, we can change any block to be chained and unchained, that is, we can write any element with a normal state or an optional state. Obviously, all operations take constant worst case time.

Z can be easily implemented using X and a bit variable flag. We manage the invariants that flag = 1 if and only if the last initialization initialized Z with the optional state, and initv stores the initial value for the last initialization when flag = 0. If flag = 1, we have Z = X, and otherwise, Z is obtained by swapping initv and an optional state in X. More precisely, we have Z[i] = X[i] if flag = 1 or X[i] \neq initv or X[i] \neq optional, and Z[i] = optional if flag = 0 and X[i] = initv, and Z[i] = initv if flag = 0 and X[i] = optional. In this way, Z supporting all operations in constant worst case time is implemented by using a normal array A of Nℓ bits, and three variables, initv of ℓ bits, b of ⌈log N⌉ bits, and flag of 1 bit. Note that initv and b cannot be embedded to A in the same way to Section 4 because we cannot represent a block that contains an optional state and normal states.

Finally, we have the following lemma.

**Lemma 3.** There is an initializable optional array Z_{N,ℓ} for ℓ ≥ 2⌈log N⌉ and ℓ ∈ O(w) using ℓ + ⌈log N⌉ + 1 extra bits which supports all operations in constant worst case time.

### 7 Conclusions and Future Works

In this paper, we discussed array initialization on the word RAM model, and showed that we can implement an abstract array with only 1 extra bit to support read, write, and initialize operations in constant worst case time. We also prove that our algorithm is time and space optimal, and any algorithm without extra space takes Ω(N) worst-case/amortized/expected time for initialize operation. Our algorithm is quite simple and the core technique can also be applied for implementing optional arrays for a special case which are also useful data structures.
As future works, we are interested in developing more useful operations which enhances an initializable array $Z_{N,\ell}$ as follows.

- **rangeInit**($Z, i, n, v$): Initialize elements $Z[i \ldots i+n]$ to an initial value $v$.
- **rangeMove**($Z, i_1, i_2, n$): Move $Z[i_1 \ldots i_1+n]$ to $Z[i_2 \ldots i_2+n]$ and then does not care about the elements of $Z[i_1 \ldots i_1+n]$ after the move.
- **rangeCopy**($Z, i_1, i_2, n$): Copy $Z[i_1 \ldots i_1+n]$ to $Z[i_2 \ldots i_2+n]$.
- **rangeSwap**($Z, i_1, i_2, n$): Swap $Z[i_1 \ldots i_1+n]$ and $Z[i_2 \ldots i_2+n]$.

If we can implement a multi-functional array that supports operations described above efficiently, it can be a powerful tool to design efficient algorithms for various problems.

Acknowledgements

We would like to thank Shunsuke Inenaga and Hideo Bannai for many constructive suggestions, and anonymous reviewers for their insightful comments.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Jon Louis Bentley. *Programming pearls*. Addison-Wesley, 1986.

[3] cppreference.com. C++ reference, 2017. Last accessed 11/3/2017. URL: [http://en.cppreference.com/w/cpp/utility/optional](http://en.cppreference.com/w/cpp/utility/optional).

[4] Yevgeniy Dodis, Mihai Patrascu, and Mikkel Thorup. Changing base without losing space. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 593–602, 2010. URL: [http://doi.acm.org/10.1145/1806689.1806770](http://doi.acm.org/10.1145/1806689.1806770), doi:10.1145/1806689.1806770.

[5] Kimmo Fredriksson and Pekka Kilpeläinen. Practically efficient array initialization. *Softw., Pract. Exper.*, 46(4):435–467, 2016. URL: [https://doi.org/10.1002/spe.2314](https://doi.org/10.1002/spe.2314), doi:10.1002/spe.2314.

[6] Torben Hagerup and Frank Kammer. On-the-fly array initialization in less space. In *28th International Symposium on Algorithms and Computation, ISAAC 2017, December 9-12, 2017, Phuket, Thailand*, pages 44:1–44:12, 2017. URL: [https://doi.org/10.4230/LIPIcs.ISAAC.2017.44](https://doi.org/10.4230/LIPIcs.ISAAC.2017.44), doi:10.4230/LIPIcs.ISAAC.2017.44.

[7] Donald E. Knuth. Simpath, 2008. Last accessed 11/3/2017. URL: [https://www-cs-faculty.stanford.edu/~knuth/programs/simpath.html](https://www-cs-faculty.stanford.edu/~knuth/programs/simpath.html).

[8] Donald E. Knuth. *The Art of Computer Programming: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley, 2009.

[9] Jacob Teo Por Loong, Jelani Nelson, and Huacheng Yu. Fillable arrays with constant time operations and a single bit of redundancy. *CoRR*, abs/1709.09574, 2017. URL: [http://arxiv.org/abs/1709.09574](http://arxiv.org/abs/1709.09574), arXiv:1709.09574.

[10] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984. URL: [https://doi.org/10.1007/978-3-642-69672-5](https://doi.org/10.1007/978-3-642-69672-5), doi:10.1007/978-3-642-69672-5.
11 Shin-ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993., pages 272–277, 1993. URL: http://doi.acm.org/10.1145/157485.164890 doi:10.1145/157485.164890

12 Shin-ichi Minato. Power of enumeration - recent topics on bdd/zdd-based techniques for discrete structure manipulation. IEICE Transactions, 100-D(8):1556–1562, 2017. URL: http://search.ieice.org/bin/summary.php?id=e100-d_8_1556.

13 Gonzalo Navarro. Constant-time array initialization in little space. Manuscript Nov 2012, 2012. URL: http://www.dcc.uchile.cl/~gnavarro/ps/sccc12.pdf.

14 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. ACM Comput. Surv., 46(4):52:1–52:47, 2014. URL: http://doi.acm.org/10.1145/2535933 doi:10.1145/2535933

15 Oracle. API specification, 2017. Last accessed 11/3/2017. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

16 Tsutomu Sasao and Jon T. Butler. Applications of Zero-Suppressed Decision Diagrams. Synthesis Lectures on Digital Circuits and Systems. Morgan & Claypool Publishers, 2014. URL: https://doi.org/10.2200/S00612ED1V01Y201411DCS045 doi:10.2200/S00612ED1V01Y201411DCS045
Algorithm 2: iinit\((v)\) and iread\((i)\)

1 Function iinit\((v)\):
2   \(\begin{align*}
3     & b \leftarrow 0 ; \\
4     & \textit{initv} \leftarrow v ; \\
7   \end{align*}\)

4 Function iread\((i)\):
5   \(i' \leftarrow \lfloor i/2 \rfloor \) // \(B_{i'}\) is a block which contains \(A[i]\).
6   \(B_k \leftarrow \text{chainedTo}(B_{i'}) ; \)
7   if \(i < 2b\) then
8      if \(B_k \neq \text{None}\) then
9         return \textit{initv} ;
10        else
11           return \(A[i] ; \)
12      else
13         if \(B_k \neq \text{None}\) then
14            if \(i \mod 2 = 0\) then
15                return \(A[A[i] + 1] ; \)
16            else
17                return \(A[i] ; \)
18            else
19                return \textit{initv} ;

Appendix
Algorithm 3: Tools

1 Function chainedTo(Bᵢ):
2   \[ k' \leftarrow A[2i] \]
3   \[ k \leftarrow \lfloor k'/2 \rfloor \] // \( B_k \) is a block which contains \( A[k'] \).
4   if \( k' \mod 2 = 0 \) and \( A[k'] = 2i \) and \( (i < b \leq k \text{ or } k < b \leq i) \) then
5       return \( B_k \);
6   else
7       return None;
8 Function makeChain(Bᵢ, Bᵢ):
9   \[ A[2i] \leftarrow 2j \]
10  \[ A[2j] \leftarrow 2i \]
11 Function breakChain(Bᵢ):
12   \[ B_k \leftarrow \text{chainedTo}(Bᵢ) \]
13   if \( B_k \neq \text{None} \) then
14     \[ A[2k] = 2k \]
15 Function initBlock(Bᵢ):
16   \[ A[2i] \leftarrow \text{initv} \]
17  \[ A[2i+1] \leftarrow \text{initv} \]
18 Function extend():
19   \[ B_k \leftarrow \text{chainedTo}(B_b) \]
20   \[ b \leftarrow b + 1 \]
21   if \( B_k = \text{None} \) then
22     \[ k \leftarrow b - 1 \]
23   else
24     \[ B_{b-1} \leftarrow (A[2k+1], A[2b+1]) \]
25     breakChain(B_{b-1})
26     initBlock(B_k)
27     breakChain(B_k)
28     return \( B_k \)
29     // \( B_k \) is an unwritten block in the unwritten chained area which is not chained yet.
30 Function shrink():
31   \[ B_k \leftarrow \text{chainedTo}(B_{b-1}) \]
32   \[ b \leftarrow b - 1 \]
33   if \( B_k = \text{None} \) then
34     \[ k \leftarrow b \]
35   else
36     \[ A[2k] \leftarrow A[2b-1] \]
37     return \( B_k \) // \( B_k \) is an unoptional block in the unoptional chained area which is not chained yet.