Fillable arrays with constant time operations and a single bit of redundancy

Jacob Teo Por Loong[*] Jelani Nelson[†] Huacheng Yu[‡]

February 9, 2018

Abstract

In the *fillable array problem* one must maintain an array $\mathtt{A}[1..n]$ of w-bit entries subject to random access reads and writes, and also a $\mathtt{fill}(\Delta)$ operation which sets every entry of $\mathtt{A}$ to some $\Delta \in \{0, \ldots, 2^w - 1\}$. We show that with just one bit of redundancy, i.e. a data structure using $nw + 1$ bits of memory, `read`/`fill` can be implemented in worst case constant time, and `write` can be implemented in *either* amortized constant time (deterministically) *or* worst case expected constant (randomized). In the latter case, we need to store an additional $O(\lg n)$ random bits to specify a permutation drawn from an $1/n^2$-almost pairwise independent family.

1 Introduction

A classic dynamic data structural problem is that of the *fillable array* [AHU74, Exercise 2.12]. In this problem, one wants to maintain an array $\mathtt{A}[1..n]$ with entries in $\{0, \ldots, 2^w - 1\}$ subject to the following three operations:

- $\mathtt{write}(i, \Delta)$: $\mathtt{A}[i] \leftarrow \Delta$
- $\mathtt{fill}(\Delta)$: $\mathtt{A}[i] \leftarrow \Delta$ for all $i = 1..n$
- $\mathtt{read}(i)$: returns $\mathtt{A}[i]$

Note $\mathtt{read}(i)$ may not be defined, if $\mathtt{A}[i]$ was never set due to a lack of a previous `fill` or $\mathtt{write}(i, \cdot)$ operation since the data structure's initialization. In this case, we allow the return value to be arbitrary (in fact, the data structures we present here return 0 in this case, or some other pre-decided constant).

Most popular programming languages have some data structure implemented in its standard library supporting all these operations. For example, arrays in C/C++ can support `fill` via a call to `memset`, and a method even named `fill` is implemented in C++ (for `ForwardIterator`), Python (`numpy.ndarray`), and Java (`Arrays`). In fact, arrays in Java must be filled with some value upon initialization as part of the language specification [Ora17].

The standard approach to implementing a fillable array uses nw bits of memory, and in the word RAM model supports `write`/`read` each in $O(1)$ worst-case time and `fill` in time $O(n)$, simply via n sequential writes. Recently [HK17] showed this is best possible for any data structure using nw bits of memory. But what if we allow our data structure to use just a single bit of extra memory? Is is possible to then achieve

[*] National University of Singapore High School of Math & Science. `jacobtpl@gmail.com`. Work done while participating in the Research Science Institute, sponsored by the Center for Excellence in Education, in Summer 2017.

[†] Harvard University. `minilek@seas.harvard.edu`. Supported by NSF grant IIS-1447471 and CAREER award CCF-1350670, ONR Young Investigator award N00014-15-1-2388 and DORECG award N00014-17-1-2127, an Alfred P. Sloan Research Fellowship, and a Google Faculty Research Award.

[‡] Harvard University. `yuhch@g.harvard.edu`. Supported in part by ONR grant N00014-15-1-2388, a Simons Investigator Award, and NSF grant CCF-1565641.

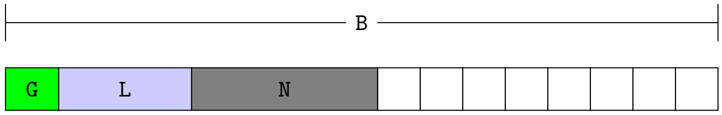

Figure 1: The organization of array `B` in linked list mode. `B` is divided into three subarrays, `G`, `L`, `N`, and each cell is a w-bit word. The white cells in the array are unused in linked list mode, except during the process of conversion into naive mode triggered by `numActive` reaching n/C_L after a `write`.

all operations in worst case constant time? Despite the ubiquity of this problem, this basic question is unanswered.

For a data structure using $nw + r$ bits of memory, we denote the value of r as the *redundancy.* The goal is to use as little redundancy as possible while supporting all three operations quickly. We assume the word RAM model with word size $w = \Omega(\lg n)$, so that at the very least indexing into `A` can be performed in constant time. A textbook exercise [AHU74, Exercise 2.12] shows that it is possible to achieve redundancy $r = 2n \lceil \lg_2 n \rceil + \lceil \lg_2(n+1) \rceil + w$ bits while supporting all three operations mentioned above in worst case time $O(1)$. As in previous work, we refer to this data structure as the "folklore" solution. The same running time was achieved with better redundancy $r = (1 + o(1))n$ by Navarro [Nav13]. Most recently, Hagerup and Kammer gave a solution with `read`/`write` time $O(t)$, `fill` time $O(1)$, and redundancy $r = \lceil n/(w/(Ct))^t \rceil$ for some constant $C > 1$ for any desired integer $1 \leq t \leq \lg_2 n$ [HK17]. All these times are worst case. For $t = \lg_2 n$, redundancy $r = 1$ is achieved.

Our main contribution. We show it is possible to achieve $O(1)$ time for all three operations with redundancy $r = 1$ if one settles for *amortized* complexity for `write` and worst-case complexity for `read` and `fill`. We also show that it is possible to replace the amortized $O(1)$ complexity for `write` with $O(1)$ worst case *expected* running time, via a randomized data structure. In this case though, we need to store an additional $O(\lg n)$ random bits to specify a permutation drawn from a $1/n^2$-almost pairwise independent family.

We point out here that simultaneously and independently of our work, Katoh and Goto in [KG17] showed an even stronger result: namely that redundancy $r = 1$ is achievable even while supporting all three operations in *worst case* $O(1)$ time. They additionally achieve this result when the elements stored in `A` are b bits each for any $b = O(w)$, whereas we assume $b = \Theta(w)$.

When describing our solutions, we assume n is larger than some fixed constant since otherwise the trivial solution with $O(n)$ `fill` time performs all operations in worst case time $O(1)$ with zero redundancy. We also henceforth use $[k]$ to denote $\{1, \ldots, k\}$ for integer k.

2 Amortized solution

We here describe and analyze our amortized solution, which is quite simple. The data structure operates in two *modes* and maintains a single mode bit which we refer to as `naive`. If `naive` is set to `True`, then we are in *naive mode.* If set to `False`, then we are in *linked list mode.* The single bit to store `naive` is the sole redundant bit in our representation, yielding $r = 1$. This data structure, in either mode, also maintains an array `B`$[1..n]$ such that each `B`$[i]$ is a w-bit word. The data structure, when first initialized, starts in linked list mode.

We first describe naive mode. In this mode, we maintain the invariant that `B`$[i]$ = `A`$[i]$ for all $i = 1 \ldots n$. Thus `write`(i, Δ) is implemented by performing the operation `B`$[i] \leftarrow \Delta$, and `read`(i) is executed by simply returning `B`$[i]$. To execute `fill`(Δ), we set `naive` to `False` then initialize the data structure into linked list mode with value Δ (this initialization is to be explained shortly).

Memory layout in linked list mode is depicted in Figure 1, together with the one extra `naive` bit not depicted there (set to `False`). We say an index $i \in [n]$ is *active* if it has been written since the most recent initialization into linked list mode. $\texttt{G}$ has size 2 and stores the argument Δ_{last} to the last $\texttt{fill}$ call, as well as the number $\texttt{numActive}$ of active indices. $\texttt{L}$ is an instance of the folklore data structure for an array with $\lceil \lg_2 n \rceil$ bit cells (sufficiently large to server as pointers into $\texttt{N}$), and with array length n/C_L for a constant $C_L > 1$ to be determined later. We abuse notation and let $\texttt{L}[j] \leftarrow \Delta$ denote $\texttt{L.write}(j, \Delta)$ and let $\texttt{L}[j]$ denote the value returned by $\texttt{L.read}(j)$. The main idea is that for each $j \in [n/C_L]$, $\texttt{L}[j]$ is a pointer to the head node of a doubly linked list which contains all active indices i in the range $\{(j-1) \cdot C_L + 1, \ldots, j \cdot C_L\}$. For any such i, there is a node in the linked list containing the pair $(i, \texttt{A}[i])$. Note that the linked list pointed to by $\texttt{L}[j]$ is guaranteed to have at most $C_L = O(1)$ nodes. As mentioned in Section 1, $\texttt{L}$ occupies at most $3n/C_L + 2 \le 4n/C_L$ cells in $\texttt{B}$. The actual linked list nodes are then allocated in the $\texttt{N}$ array, which has a length that will be determined later. Each linked list node occupies 4 w-bit cells, to store $\texttt{prev}$ and $\texttt{next}$ pointers (which are stored as indices into $\texttt{N}$), as well as the two values i and $\texttt{A}[i]$ corresponding to that node. Null pointers are represented by the value n, which is unambiguous since $\texttt{N}$ has size much less than n. The number of allocated nodes will always be equal to $\texttt{numActive}$, and thus whenever we wish to allocate a new node, we will do so by incrementing $\texttt{numActive}$ then using memory cells in the length-4 subarray $\texttt{N}[(4 \cdot (\texttt{numActive} - 1) + 1)..(4 \cdot \texttt{numActive})]$.

Now we describe how to perform operations in linked list mode. To perform $\texttt{fill}(\Delta)$, no matter which mode we are in when the $\texttt{fill}$ was called, we set $\texttt{naive}$ to $\texttt{False}$ and do $\texttt{L.fill}(null)$ (as mentioned previously, $null$ can be unambiguously represented by the value n in this context). We also set $\texttt{numActive}$ to 0 and Δ_{last} to Δ. Initializing the entire data structure at the beginning of the operation sequence is identical, except that we set Δ_{last} to be 0 (or whatever other pre-specified constant we would like to return when an $\texttt{A}[i]$ value has never been set). Answering a $\texttt{read}(i)$ query is also simple. Set $j \leftarrow \lfloor (i-1)/C_L \rfloor$. We first check whether $\texttt{L}[j]$ is $null$. If so, we return Δ_{last}. Otherwise, we traverse the linked list $\texttt{L}[j]$. If this list contains a node with a pair with index i, then we return the associated value in that node. Otherwise, we return Δ_{last}. Note $\texttt{fill}$ takes worst-case constant time as does $\texttt{read}$. This is because all $\texttt{read}/\texttt{write}/\texttt{fill}$ operations on $\texttt{L}$ take constant time, and traversing $\texttt{L}[j]$ during a $\texttt{read}$ takes time $O(C_L) = O(1)$.

The most involved operation to implement is the $\texttt{write}(i, \Delta)$ operation, which we now describe. We first determine whether i was already active before this $\texttt{write}$ by performing the steps of $\texttt{read}(i)$. For $j = \lfloor (i-1)/C_L \rfloor$ as defined above, note i is active iff $\texttt{L}[j] \neq null$ and the linked list $\texttt{L}[j]$ contains a node with stored index i. If i was already active, we simply ovewrite Δ as the associated value in the linked list node containing i. Otherwise, we increment $\texttt{numActive}$ then allocate a new node v containing (i, Δ) and insert it to the front of the linked list $\texttt{L}[j]$. If $\texttt{L}[j]$ was $null$, then we set $\texttt{L}[j]$ to the first cell of v in $\texttt{N}$. The main issue with this solution is that once $\texttt{numActive}$ is sufficiently large, we will run out of memory. This is because, on top the memory used to store $\texttt{G}, \texttt{L}$, every active index also uses up 4 memory cells in $\texttt{N}$. Since the number of active indices can be as big as n and $\texttt{B}$ only contains $n < 4n$ cells, we may run out of memory in $\texttt{N}$ if the number of active indices becomes too large.

To avoid the above issue, we convert from linked list mode to naive mode whenever $\texttt{numActive}$ becomes too large; in particular, whenever it reaches n/C_L. Note then $\texttt{N}$ need only be of length $4n/C_L$. To perform this conversion, we first set $\texttt{naive} \leftarrow \texttt{True}$. We then set all white cells in $\texttt{B}$ (see Figure 1) to 0. We then loop from $j = n/C_L$ down to j^*, for j^* also to be determined later, and for each such j we free all nodes in $\texttt{L}[j]$. To free a node v with prev/next pointers to $v.\texttt{prev}$ and $v.\texttt{next}$ and storing index $v.i$ and value $v.\texttt{val}$, we first set $\texttt{B}[v.i] \leftarrow \texttt{B}[v.\texttt{val}]$. We then set the $\texttt{next}$ pointer of $\texttt{N}[v.\texttt{prev}]$ and $\texttt{prev}$ pointer of $\texttt{N}[v.\texttt{next}]$ to point to each other, if not $null$. We then move the last node stored in $\texttt{N}$ (which is stored in the 4 cells starting at $4 \cdot (\texttt{numActive} - 1) + 1$, inclusive) into the 4 cells of $\texttt{N}$ that used to store v. We then decrement $\texttt{numActive}$. In this way, during conversion into naive mode $\texttt{numActive}$ keeps track of the number of active indices that are yet to be converted into the naive representation. Note that if we divide $\texttt{A}$ into contiguous blocks of length C_L, then active indices are converted into the naive representation in descending block order (though the order of conversion within a block may be arbitrary since linked lists are not sorted by index). We choose the value j^* to be such that the j^*th block of indices in $\texttt{A}$ is the closest block immediately to the right of the indices used in storing $\texttt{L}$. In this way, the conversion continues until we pause midway, when we have converted all blocks of indices that do not intersect $\texttt{G}, \texttt{L}, \texttt{N}$.

We now describe how to complete the conversion into naive mode, that is to convert all the indices in the remaining blocks $1, \ldots, j^*-1$. Let the white part of the array `B` (see Figure 1) be denoted as subarray `H`. The idea here is to use *gaps* of three consecutive zeroes in `H` to represent linked list nodes. Our goal is to build a linked list using the memory in these gaps to store all indices pointing to cells in `G`, `L`, `N` that are waiting to be converted. Let us now set some values. Note `G`, `L`, `N` combined use at most $2 + 4n/C_L + 4n/C_L = 8n/C_L + 2$ cells. As mentioned in Section 1, we can assume n is larger than some constant. In particular, we assume $n \geq 2C_L$ so that $8n/C_L + 2 \leq 9n/C_L$. Thus we have $j^* - 1 \leq 9n/C_L^2 + 1 \leq 10n/C_L^2$ assuming also $n \geq C_L^2/9$, and thus have at most $10n/C_L$ indices remaining to be converted. We need to make sure these cells can all be written into the gaps in `H`. Note `H` has length at least $(1 - 9/C_L)n$ and contains a total of at most n/C_L entries that are not zero (due to conversions of indices in blocks j^* and above). Thus `H` contains at least $\lfloor(1 - 12/C_L)n/3\rfloor$ disjoint gaps of three consecutive zero entries. We need $\lfloor(1 - 12/C_L)n/3\rfloor \geq 10n/C_L$ to ensure these items all fit in the gaps and `H`, and thus it suffices to set $C_L = 50$ for $n \geq 10$. Thus overall we have assumed $n \geq \max\{2C_L, C_L^2/9, 10\} = 350$. We then use two pointers to simultaneously walk over the first `numActive` nodes in `N` while walking over `H`, copying nodes into the gaps of three consecutive zeroes to form a link list in the gaps of `H`. We also use a single register during the conversion process to store the first cell of the first gap of three in `H` (i.e. so that we know the head of the linked list). After we have finished copying over the remaining indices in `N` to the gaps in `H`, we then walk over the `B` entries used to store `G`, `L`, `N` then set them all to zero, then walk over the linked list in the gaps in `H` and write the values of all these indices into their respective indices in index sections `G`, `L`, `N`. We then perform one more walk over this gap linked list and rewrite zero in all its cells.

Note that this conversion process from linked list mode back to naive mode takes time $O(n)$, which can be charged to the n/C_L active indices since the last `fill`. Thus overall this conversion process takes amortized time $O(1)$.

Theorem 1. *There is a deterministic data structure implementing fillable arrays with one bit of redundancy, supporting worst-case $O(1)$ time for `read`/`fill` and $O(1)$ amortized time per `write`.*

3 Randomized solution

In this section, we present a randomized implementation of a fillable array providing constant time per operation in expectation in the worst-case, and using one bit of redundancy. In fact, `read` and `fill` will take $O(1)$ time with probability 1, whereas each `write` will run in expected time $O(1)$. Our analysis assumes oracle access to a permutation F drawn from an $1/n^r$-almost r-wise independent distribution of permutations on the set $[n]$ for an even $r \geq 2$. As we show in Appendix A.1, such an F can be stored in $O(\lg n)$ bits of space and evaluated in worst-case constant time on any $i \in [n]$, and it can be found in expected time *poly*$(\lg n)$ in pre-processing (see Remark 9). We use the following standard definition of δ-almost k-wise independent permutation families. See for example [KNR09].

Definition 2. *Let D_1, D_2 be distributions over a finite set Ω. The* variation distance between *between D_1 and D_2 is*

$$\|D_1 - D_2\| := \frac{1}{2}\sum_{\omega\in\Omega} |D_1(\omega) - D_2(\omega)|$$

We say that D_1, D_2 are δ-close if $\|D_1 - D_2\| \leq \delta$.

Definition 3. *Let $U_{\{n_k\}}$ denote the uniform distribution over the set of all k-tuples of distinct integers in $[n]$. A set Π of permutations on $[n]$ is δ-almost k-wise independent if for every k-tuple of distinct elements $x_1, \ldots, x_k \in [n]$, the distribution $(f(x_1), \ldots, f(x_k))$ for uniformly random $\pi \in \Pi$ is δ-close to $U_{\{n_k\}}$.*

The high-level idea of the randomized solution is similar to the amortized solution presented in the previous section. The data structure will have two modes: the *naive mode* and the *linked list mode.* In the amortized solution, the only operation that takes more than constant time is when we need to convert the data structure from linked list mode to naive mode, which takes linear time. However, this only happens after $\Theta(n)$ `write` operations after a `fill`. To obtain expected worst-case constant time, the main idea is to gradually convert to naive mode over the $\Theta(n)$ `write` operations. Since we put the last C_L elements into the last linked list, it allows us to fill the last C_L words of the array with their current values by going over the last linked list, and delete the last linked list. Then we can view our data structure as in linked list mode for the first $n - C_L$ elements and in naive mode for the last C_L elements. However, if we keep inserting the elements that are in the first, say half, of the blocks, and convert to naive mode from the last blocks, we will at some point run out of space. To avoid this issue, we apply a random permutation on the array `A`, and prove that in expectation, we will "run out of space" only when there are a constant number of blocks left. In the following, we present this approach with details.

The folklore solution with `delete`. The randomized solution we present in this section uses an implementation of the folklore solution supporting `delete` operation as a subroutine. More specifically, the subroutine maintains an array `A` of length n using $3n+2$ words, supporting

- `read`(i): return `A`$[i]$;
- `write`(i, Δ): set `A`$[i]$ to Δ;
- `fill`(Δ): set `A`$[i]$ to Δ for all $1 \leq i \leq n$;
- `delete`(n): deletes the last (n-th) element of the array `A`, such that the data structure only uses the first $3(n-1)+2$ words of the memory.

The subroutine supports every operation deterministically in constant time in worst case. We defer the details to Appendix B.

Memory layout. As in the amortized solution, we refer to the one redundant bit as `naive`, which stores the mode of the data structure. The rest of the data structure is stored in the memory `B` of n w-bit words.

When `naive` is `True`, the data structure is in naive mode. In this case, we store `A`$[i]$ in `B`$[F(i)]$ for each i, where F is the permutation previously mentioned.

When `naive` is set to `False`, the data structure is in linked list mode. In this case, we partition the array `A` into $\lceil n/C_L \rceil$ blocks. The j-th block contains all the entries $i \in [n]$ such that $(j-1) \cdot C_L + 1 \leq F(i) \leq j \cdot C_L$. Each block is associated with a *doubly* linked list, in which, we store all elements that have been performed a `write` operation on since the last `fill`. The n-word memory `B` is partitioned into five subarrays in the following order (see Figure 2).

- `G`: this subarray has five words. The first four words store the pointers to the first word of the following subarrays. The last word stores Δ_{last}, the value to which the last `fill` operation sets.
- `L`: this subarray stores a folklore data structure for the heads of all doubly linked lists.
- `N`: this subarray stores all nodes in all linked lists. Each node has four fields, which are store in four words: the pointer to its predecessor, the pointer to its successor, the index and the value. To indicated the end of a linked list, the successor pointer of the last node will point to a word not in `N`, e.g., the first word of `G`. The same convention applies to `L` when a linked list is empty, i.e., the header points to the first word of `G`.
- `U`: this subarray is unused.
- `NI`: this subarray stores values of all entries that are mapped to this range by F, i.e., we set `B`$[F(i)]$ = `A`$[i]$ for all $F(i)$ in this range.

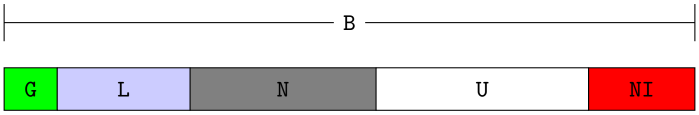

Figure 2: Memory layout of the data structure in linked list mode

Operation `read`(i). If the data structure is in naive mode, the value of $\texttt{A}[i]$ is stored in the $F(i)$-th word of the memory. If the data structure is in linked list mode, we first check if $\texttt{B}[F(i)]$ is in `NI`, i.e., $\texttt{A}[i]$ is converted to the naive mode already. If it is, we simply return $\texttt{B}[F(i)]$ as in the naive mode. Otherwise, the $\lfloor i/C_L \rfloor$-th block contains the element $\texttt{A}[i]$. We `read` the folklore data structure in subarray `L`, and obtain the header of the linked list associated with this block. Then we traverse this linked list to find all elements in the block that have been written since the last `fill`. If $\texttt{A}[i]$ is found in the linked list, we simply return its value stored in it. Otherwise, we return Δ_{last}.

Operation `write`(i, Δ). If the data structure is in naive mode, we simply write the value Δ to the $F(i)$-th word. If the data structure is in linked list mode, we first `read` the folklore data structure to find the header of the linked list associated with $\lfloor i/C_L \rfloor$-th block. Next we traverse the linked list to check if $\texttt{A}[i]$ is already in it. If it is, we overwrite the value field of the node for $\texttt{A}[i]$ with Δ. Otherwise, we allocate four more words at the end of `N`, which can be done by increasing the pointer to subarray `U` by four words. Then we create a new node there for $\texttt{A}[i]$, and insert it to the linked list.

An important idea of our randomized data structure is to gradually convert to naive mode. Thus, in addition to the above procedure, we will perform *convert* after each `write` operation.

Convert. The *convert* procedure converts the last C_T blocks into naive mode that are not converted yet, for some constant C_T to be set later. We first check if there is sufficient unused space (`U`) left. To do this, we first calculate the number of blocks k that are still in linked list mode. This number can be obtained from the size of `NI`, i.e., $k = (n - |\texttt{NI}|)/C_L$. If $|\texttt{U}| \leq k \cdot C_L \cdot C_U$ or $k \leq 10$ for some constant C_U (i.e., the data structure is running out of space soon), we run the linear time conversion algorithm on the remaining k blocks as in the amortized solution, and set `naive` to `True`. Note that as long as we set C_U to be greater than 0.95, the linear time conversion algorithm will have sufficient working memory as we described in the previous section.

If $|\texttt{U}| > k \cdot C_L \cdot C_U$ and $k > 10$, we still have enough unused space, and can safely convert the blocks. To convert the k-th block, we first decrease the pointer to `NI` by C_L words, and fill all of these C_L words with Δ_{last}. Next, we traverse the k-th linked list, and for all elements in the linked list, fill the $F(i)$-th word with the value of $\texttt{A}[i]$. Then, we need to delete the last linked list. When deleting a node, we may create four unused words in the middle of `N`. In this case, we simply move the last node in `N` to this place, update the pointers and decrease the pointer to `U` by four words. To delete the header, it suffices to run the `delete` operation on the folklore data structure (see Appendix B). If a gap of more than four words is created between `L` and `N`, we again move the last node here, update the pointers and decrease the pointer to `U`.

Finally, we repeat the above procedure C_T times to convert the last C_T blocks.

Operation `fill`(Δ). We set `naive` to `False` no matter which mode the data structure was in, and set Δ_{last} to Δ. Then we update the pointers in `G` such that `G` has five words, `L` has the size of a folklore solution on $\lceil n/C_L \rceil$ elements $(3\lceil n/C_L \rceil + 2)$, `N` and `NI` are empty, and `U` has the remaining memory. Finally we `fill` the folklore data structure with pointers to the first word of `G`, i.e., empty all linked lists.

Analysis The correctness of the data structure is straightforward. It is also easy to verify that the only part of the data structure that may take super-constant time is the convert procedure.

In the convert procedure, when too little unused space is left compared to the number of blocks remaining ($|\mathtt{U}| \le k \cdot C_L \cdot C_U$), we convert all remaining blocks at once. In the following, we will show that this event happens with very small probability when the number of remaining blocks is large.

Fix a sequence of operations, and one operation in this sequence. Now we analyze the expected time spent on this operation by the data structure. If it is a `fill` or a `read`, the data structure does not invoke the convert procedure, and thus takes constant time in worst case. Otherwise, it is a `write` operation, and if the data structure has run a linear time conversion algorithm since the last `fill`, this `write` operation will take constant time in the worst case.

Otherwise, let k_U be the number of `write` operations since the last `fill`. The data structure invokes convert exactly once during each of the k_U `write`s. The convert procedure converts C_T blocks each time. Thus, we will have exactly $k = \lceil n/C_L \rceil - k_U \cdot C_T$ blocks left.

Let X be the number of entries written in those k_U `write` operations and mapped to the first k blocks, i.e., the number of elements that are inserted an still in linked list mode. We need to run a linear time conversion algorithm only when $|\mathtt{U}| \le k \cdot C_L \cdot C_U$. On the other hand, we have

$$\begin{aligned} |\mathtt{U}| &\ge k \cdot C_L - |\mathtt{N}| - |\mathtt{L}| - |\mathtt{G}| - 3 \\ &\ge k \cdot C_L - 4X - (3k+2) - 8 \\ &= k \cdot C_L - 4X - 3k - 10. \end{aligned}$$

That is, we run the linear time conversion algorithm, only when

$$X \ge \frac{1}{4}\left(k \cdot C_L \cdot (1 - C_U) - 3k - 10\right).$$

However,

$$\mathbb{E}\, X \le k_U \cdot \frac{k \cdot C_L}{n} \le \frac{n}{C_L \cdot C_T} \cdot \frac{k \cdot C_L}{n} = \frac{k}{C_T},$$

which is much smaller. Now we are going to upper bound the probability using the r-wise independence of F. Recall that F is sampled from a distribution $\mathcal{D}_r$ such that $F(i)$'s are r-wise $1/n^r$-almost independent. Let $\mathcal{U}$ be the uniform distribution over all permutations $[n] \to [n]$. Let X_i be the indicator variable for the event that $F(i) \le k \cdot C_L$, and let S be the set of k_U entries that are written after the last `fill`.

By definition, we have the following:

- $X = \sum_{i \in S} X_i$;
- $\mathbb{E}_{F \sim \mathcal{U}}\, X_i = \frac{k \cdot C_L}{n}$;
- for any subset $T \subseteq S$ and $|T| \le r$, by the r-wise almost independence, we have

$$\left| \mathop{\mathbb{P}}_{F \sim \mathcal{D}_r}\left(\bigwedge_{i \in T} X_i = 1\right) - \mathop{\mathbb{P}}_{F \sim \mathcal{U}}\left(\bigwedge_{i \in T} X_i = 1\right) \right| \le 1/n^r;$$

- for uniform F and any $T \subseteq S$, we have

$$\begin{aligned} \mathop{\mathbb{P}}_{F \sim \mathcal{U}}\left(\bigwedge_{i \in T} X_i = 1\right) &= \frac{k \cdot C_L}{n} \cdot \frac{k \cdot C_L - 1}{n - 1} \cdot \dots \cdot \frac{k \cdot C_L - |T| + 1}{n - |T| + 1} \\ &\le \left(\frac{k \cdot C_L}{n}\right)^{|T|} = \prod_{i \in T} \mathop{\mathbb{P}}_{F \sim \mathcal{U}}(X_i = 1) \end{aligned} \tag{1}$$

  and

$$\mathop{\mathbb{P}}_{F \sim \mathcal{U}}\left(\bigvee_{i \in T} X_i = 0\right) \le \prod_{i \in T} \mathop{\mathbb{P}}_{F \sim \mathcal{U}}(X_i = 0). \tag{2}$$

Now we set $C_L = 100$, $C_T = 8$ and $C_U = 0.95$, and have

$$\begin{aligned}
&\mathop{\mathbb{P}}_{F\sim\mathcal{D}_r}\left(X \geq \frac{1}{4}\left(k\cdot C_L\cdot(1-C_U) - 3k - 10\right)\right)\\
\leq\ &\mathop{\mathbb{P}}_{F\sim\mathcal{D}_r}\left(X \geq \frac{k}{4}\right)\\
\leq\ &\mathop{\mathbb{P}}_{F\sim\mathcal{D}_r}\left(\sum_{i\in S}(X_i - \mathbb{E}\,X_i) \geq \frac{k}{8}\right)\\
\leq\ &\mathop{\mathbb{P}}_{F\sim\mathcal{D}_r}\left(\left(\sum_{i\in S}(X_i - \mathbb{E}\,X_i)\right)^r \geq \left(\frac{k}{8}\right)^r\right)\\
\leq\ &\frac{\mathbb{E}_{F\sim\mathcal{D}_r}\left(\sum_{i\in S}(X_i - \mathbb{E}\,X_i)\right)^r}{(k/8)^r}\\
\leq\ &\frac{\mathbb{E}_{F\sim\mathcal{U}}\left(\sum_{i\in S}(X_i - \mathbb{E}\,X_i)\right)^r + (2|S|)^r/n^r}{(k/8)^r}. \qquad (3)
\end{aligned}$$

Thus, it suffices to upper bound the r-th moment of $X_i - \mathbb{E}\,X_i$ when F is a uniformly random permutation. We will first apply the following generalized Chernoff bound to upper bound the tail probability.

Theorem 4 ([PS97, IK10])**.** *Let X be the sum of n Boolean random variables $X_1, \ldots, X_n$. Suppose that there are $0 \leq \delta_i \leq 1$, for $1 \leq i \leq n$, for all $T \subset [n]$,*

$$\mathbb{P}(\wedge_{i\in T} X_i = 1) \leq \prod_{i\in T}\delta_i.$$

Let $\delta = (1/n)\sum_{i=1}^n \delta_i$. Then for any $\gamma > \delta$,

$$\mathbb{P}(X \geq \gamma n) \leq e^{-nD(\gamma||\delta)},$$

where $D(\gamma||\delta) = \gamma\ln(\gamma/\delta) + (1-\gamma)\ln((1-\gamma)/(1-\delta))$.

We can also prove the following inequalities about $D(\gamma||\delta)$ (see Appendix C):

- $D(\delta(1+\epsilon)||\delta) \geq \frac{1}{3}\epsilon^2\delta$ for $0 \leq \epsilon \leq 1$ and $0 \leq \delta \leq 1/(1+\epsilon)$;
- $D(\delta(1+\epsilon)||\delta) \geq \frac{1}{3}\epsilon\delta$ for $\epsilon > 1$ and $0 \leq \delta \leq 1/(1+\epsilon)$;
- $D((1-\delta(1-\epsilon))||1-\delta) \geq \frac{1}{2}\epsilon^2\delta$ for $0 \leq \epsilon \leq 1$ and $0 \leq \delta \leq 1$.

Now we apply Theorem 4 to $(X_i)_{i\in S}$ and $(1-X_i)_{i\in S}$ respectively. By Equation (1) and (2), we have for any $0 < c < 1$,

$$\mathop{\mathbb{P}}_{F\sim\mathcal{U}}(X \geq (1+c)\cdot\mathbb{E}\,X) \leq e^{-\frac{1}{3}c^2\,\mathbb{E}\,X},$$

and

$$\mathop{\mathbb{P}}_{F\sim\mathcal{U}}(X \leq (1-c)\cdot\mathbb{E}\,X) \leq e^{-\frac{1}{2}c^2\,\mathbb{E}\,X},$$

for any $c > 1$,

$$\mathop{\mathbb{P}}_{F\sim\mathcal{U}}(X \geq (1+c)\cdot\mathbb{E}\,X) \leq e^{-\frac{1}{3}c\,\mathbb{E}\,X}.$$

Now we are ready to upper bound the r-th moment:

$$\begin{aligned}
\mathop{\mathbb{E}}_{F\sim\mathcal{U}}(X-\mathbb{E}\,X)^r &= \int_0^\infty \mathbb{P}(|X-\mathbb{E}\,X|\geq x)\cdot rx^{r-1}\mathrm{d}x \\
&= (\mathbb{E}\,X)^r\cdot\int_0^\infty \mathbb{P}(|X-\mathbb{E}\,X|\geq c\cdot\mathbb{E}\,X)\cdot rc^{r-1}\mathrm{d}c \\
&\leq (\mathbb{E}\,X)^r\cdot\biggl(\int_0^1 e^{-\frac{1}{3}c^2\,\mathbb{E}\,X}\cdot rc^{r-1}\mathrm{d}c \\
&+\int_1^\infty e^{-\frac{1}{3}c\,\mathbb{E}\,X}\cdot rc^{r-1}\mathrm{d}c \\
&+\int_0^1 e^{-\frac{1}{2}c^2\,\mathbb{E}\,X}\cdot rc^{r-1}\mathrm{d}c\biggr).
\end{aligned}$$

Similar to the moments of Gaussian distributions and exponential distributions [Kri06], we have

$$\begin{aligned}
\int_0^1 e^{-\frac{1}{3}c^2\,\mathbb{E}\,X}\cdot rc^{r-1}\mathrm{d}c &< r\cdot\int_0^\infty e^{-\frac{1}{3}c^2\,\mathbb{E}\,X}\cdot c^{r-1}\mathrm{d}c \\
&= \frac{r!!}{(2\,\mathbb{E}\,X/3)^{r/2}}
\end{aligned}$$

for any even r; we also have

$$\begin{aligned}
\int_1^\infty e^{-\frac{1}{3}c\,\mathbb{E}\,X}\cdot rc^{r-1}\mathrm{d}c &< r\cdot\int_0^\infty e^{-\frac{1}{3}c\,\mathbb{E}\,X}\cdot c^{r-1}\mathrm{d}c \\
&= \frac{r!}{(\mathbb{E}\,X/3)^r};
\end{aligned}$$

and similarly,

$$\int_0^1 e^{-\frac{1}{2}c^2\,\mathbb{E}\,X}\cdot rc^{r-1}\mathrm{d}c < \frac{r!!}{(\mathbb{E}\,X)^{r/2}}.$$

Thus, for any even constant r, we have

$$\mathop{\mathbb{E}}_{F\sim\mathcal{U}}(X-\mathbb{E}\,X)^r < O(k^{r/2}).$$

Therefore, by Equation (3), the probability that the data structure runs a $O(k)$-time conversion algorithm is at most $O(k^{-r/2})$, i.e., the running time on this operation is $O(1)$ in expectation and with high probability.[1]

Theorem 5. *There is a Las Vegas randomized implementation of the fillable arrays with one bit of redundancy such that for any sequence of operations, each `read`/`fill` operation takes constant time in worst case, and each `write` operation takes constant time in expectation and with high probability, assuming it has oracle access to a permutation F drawn from a $1/n^r$-almost r-wise independent family of permutations over $[n]$.*

As described in Section A.1, for any integer $r>0$, the permutation F from an $1/n^r$-almost r-wise independent family can be represented in $O(r^2\lg n)$ bits of memory, sampled in $\mathit{poly}(\lg n)$ time, and evaluated in $O(r^2)$ time.

Acknowledgments

We thank Omer Reingold explaining to us the construction in Appendix A.1, and for allowing us to include a description of this construction.

[1] Note that we do not have to sum over all k, since each operation has a fixed k.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[HK17] Torben Hagerup and Frank Kammer. On-the-fly array initialization in less space. In *Proceedings of the 28th International Symposium on Algorithms and Computation (ISAAC)*, pages 44:1–44:12, 2017.

[Hux72] Martin N. Huxley. On the difference between consecutive primes. *Inventiones Mathematicae*, pages 164–170, 1972.

[IK10] Russell Impagliazzo and Valentine Kabanets. Constructive proofs of concentration bounds. In *Proceedings of the 14th International Workshop on Randomization and Computation (RANDOM)*, pages 617–631, 2010.

[KG17] Takashi Katoh and Keisuke Goto. In-place initializable arrays. *CoRR*, abs/1709.08900, 2017.

[KNR09] Eyal Kaplan, Moni Naor, and Omer Reingold. Derandomized constructions of k-wise (almost) independent permutations. *Algorithmica*, 55(1):113–133, 2009.

[Kri06] K. Krishnamoorthy. *Handbook of Statistical Distributions with Applications*. Chaman & Hill/CRC, 2006.

[Nav13] Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2013.

[NR99] Moni Naor and Omer Reingold. On the construction of pseudorandom permutations: Luby-rackoff revisited. *J. Cryptology*, 12(1):29–66, 1999.

[Ora17] Oracle. Java language specification, 2017. Last accessed 9/25/2017. `https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.12.5`.

[PS97] Alessandro Panconesi and Aravind Srinivasan. Randomized distributed edge coloring via an extension of the Chernoff–Hoeffding bounds. *SIAM J. Comput.*, 26(2):350–368, April 1997.

A Appendix

A.1 Almost k-wise independent permutations for all n

We here describe how to obtain an $O(1/n^c)$-almost k-wise independent permutation family Π over $[n]$ of size $poly(n)$ for any n larger than some constant, such that given an $O(ck \lg n)$-bit description of a π drawn randomly from Π we can compute $\pi(i)$ for any i in $O(ck)$ time. Here $c > 0$ and $k \geq 2$ may be arbitrary integers. This construction is used in Section 3.

The starting point of the construction of Π is a $2k^2/|\mathbb{F}|$-almost k-wise independent permutation family over $\mathbb{F}^2$ for any field $\mathbb{F}$ [NR99]. For any element $x = (x_1, x_2) \in \mathbb{F}^2$, define $x|_L = x_1$ and $x|_R = x_2$. For any function $f : \mathbb{F} \to \mathbb{F}$, define permutation $\mathbf{D}_f$ over $\mathbb{F}^2$ as

$$\mathbf{D}_f(x_1, x_2) := (x_2, x_1 + f(x_2)).$$

Theorem 6 ([NR99]). *Let h_1, h_2 be two independent random permutations over $\mathbb{F}^2$ such that for every $x \neq y$, $\mathbb{P}[h_1(x)|_R = h_1(y)|_R] \leq |\mathbb{F}|^{-1}$ and $\mathbb{P}[h_2(x)|_L = h_2(y)|_L] \leq |\mathbb{F}|^{-1}$. Let $f_1, f_2 : \mathbb{F} \to \mathbb{F}$ be two functions sampled independently from a family of k-wise independent functions. Then $S = h_2^{-1} \circ \mathbf{D}_{f_2} \circ \mathbf{D}_{f_1} \circ h_1$ is a $2k^2/|\mathbb{F}|$-almost k-wise independent permutation.*

The set $\{x \mapsto \sum_{i=0}^{k-1} a_i x^i : a_i \in \mathbb{F} \text{ for } i \in [k]\}$ is a standard construction of a family of k-wise independent functions. Each function in this family has $O(k \lg |\mathbb{F}|)$ description size and takes $O(k)$ field operations to evaluate. It is also not hard to construct families of permutations for h_1 and h_2:

$$h_1(x_1, x_2) := (x_1, a_1 x_1 + b_1 x_2)$$

and

$$h_2(x_1, x_2) := (a_2 x_1 + b_2 x_2, x_2)$$

for $a_1, a_2, b_1, b_2 \in \mathbb{F}$ and $b_1, b_2 \neq 0$ chosen uniformly at random. Both functions have $O(\lg |\mathbb{F}|)$ description size and take $O(1)$ field operations to evaluate.

To generalize the above construction to any set size n, we make use of the following theorem.

Theorem 7 ([Hux72])**.** *For any $\theta > 7/12$ there exists a constant $n_0 > 0$ such that for all $n > n_0$, the interval $[n - n^\theta, n]$ contains $\Theta(n^\theta / \lg n)$ prime numbers.*

We first describe how to extend the construction to an $O(1/n^{(1-\theta)/2})$-almost k-wise independent family over permutations on $[n]$ for arbitrary integer $n > n_0^2$ (for smaller n, one can just use the family of *all* permutations on $[n]$, which has constant size).

Pick a prime $p \in [n^{1/2} - n^{\theta/2}, n^{1/2}]$, which we know exists by Theorem 7. Then $\pi \sim \Pi$ will be specified by picking a random permutation S according to Theorem 6 (setting $\mathbb{F} = \mathbb{F}_p$) and an integer $r \in \{0, \ldots, n-1\}$ uniformly at random. By abuse of notation, we may also treat S as a random permutation over the set $[p^2]$. Define $shift_r(x) = x + r \mod n$. Then for $x \in [n]$ we define

$$\pi(x) = \begin{cases} shift_r(x), & \text{if } shift_r(x) \geq p^2 \\ S(shift_r(x)), & \text{otherwise.} \end{cases}$$

It is clear that any such π is a permutation on $[n]$ and that $\pi(x)$ can be evaluated in worst case time $O(1)$, and furthermore a simple computation shows that Π is $O(n^{(1+\theta)/2}/n)$-almost k-wise independent by Theorem 6 for any constant k.

In order to decrease δ from $n^{(1+\theta)/2}/n$ down to $O(1/n^c)$, we use the following theorem of [KNR09].

Theorem 8. *[KNR09, Theorem 3.8] For a set of functions $\mathcal{F}$, let $\mathcal{F}^\ell$ denote the set of all functions $\{f_1 \circ f_2 \cdots f_\ell : f_1, \ldots, f_\ell \in \mathcal{F}\}$ so that $|\mathcal{F}^\ell| = |\mathcal{F}|^\ell$. Then if Π is a δ-almost k-wise independent permutation family, then for any integer $\ell > 1$, Π^ℓ is a $(\frac{1}{2}(2\delta)^\ell)$-almost k-wise independent permutation family.*

Thus to decrease δ, we can apply Theorem 8 with $\ell = \lceil 2c/(1-\theta) \rceil = O(c)$. The seed length and evaluation time to compute π drawn randomly from Π^ℓ then both increase by only $O(c)$ factors.

Remark 9. Note that to apply the above construction, we need to find a prime $p \in [n^{1/2} - n^{\theta/2}, n^{1/2}]$ during pre-processing. By Theorem 7, there are many such primes p in this interval. In particular, we succeed in finding a prime with probability $\Omega(1/\lg n)$ by picking a random p in this interval, which we can then test for primality in *poly*$(\lg n)$ deterministically [AKS04]. Thus we can find this p with a Las Vegas algorithm in pre-processing in expected time (and even with high probability) *poly*$(\lg n)$.

B Folklore solution with delete

In this subsection, we present an implementation of the folklore data structure for fillable array `A` of length n using $3n + 2$ words of space. Moreover, this implementation supports an extra operation `delete`(n), which deletes the last (n-th) element in `A` such that the data structure only uses first $3(n-1) + 2$ words of the memory after the operation.

The data structure will maintain the following variable/arrays:

- `numActive`: the number of different elements written since the last `fill`

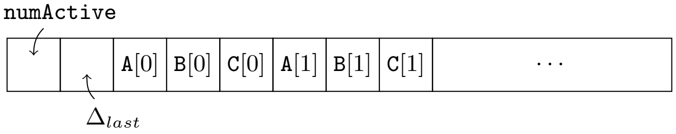

Figure 3: Memory layout of the folklore solution.

- Δ_{last}: the value that the last `fill` sets the array to
- `A`: the array `A`
- `B`: first `numActive` entries store all elements written since the last `fill`
- `C`: pointers to `B`, i.e., `B`[`C`[i]] $= i$ if i is written since the last `fill`

All three arrays `A`, `B` and `C` has length n. In total, the data structure uses $3n + 2$ words of space. To accomondate the `delete` operation, the three arrays will be interleaved with each other in memory (see Figure B).

Now we show how to implement the operations:

- To `fill` the array with Δ, it suffices to set `numActive` to 0, and set Δ_{last} to Δ.
- To `read` `A`[i], we first check if `C`[i] $\leq$ `numActive`. If `C`[i] $>$ `numActive`, we know i-th entry must have not been written since the last `fill`. In this case, we return Δ_{last}. If `C`[i] $\leq$ `numActive`, we then check if `B`[`C`[i]] $= i$. If `B`[`C`[i]] $= i$, we return `A`[i]. Otherwise, we return Δ_{last}.
- To `write` Δ to `A`[i], we first set `A`[i] to Δ. Next, we check if this is the first we write to `A`[i] since the last `fill` in the same way as we `read` `A`[i]: check if `C`[i] $>$ `numActive` or `B`[`C`[i]] $\neq i$. If it is, we increment `numActive` by one, set `B`[`numActive`] to i and set `C`[i] to `numActive`.
- To `delete` the last element, we first check if it has been written since the last `fill`. If it has not, we do not have to do anything, and just ignore the last three words from now on. Otherwise, we need to delete the record of n in `B`. This can be done by moving `B`[`numActive`] to `B`[`C`[n]] and setting `C`[`B`[`numActive`]] to `C`[n].

C Inequalities about $D(\gamma||\delta)$

In this section, we prove the following three inequalities:

1. $D(\delta(1+\epsilon)||\delta) \geq \frac{1}{3}\epsilon^2\delta$ for $0 \leq \epsilon \leq 1$ and $0 \leq \delta \leq 1/(1+\epsilon)$;
2. $D(\delta(1+\epsilon)||\delta) \geq \frac{1}{3}\epsilon\delta$ for $\epsilon > 1$ and $0 \leq \delta \leq 1/(1+\epsilon)$;
3. $D((1-\delta(1-\epsilon))||1-\delta) \geq \frac{1}{2}\epsilon^2\delta$ for $0 \leq \epsilon \leq 1$ and $0 \leq \delta \leq 1$.

Recall that $D(\gamma||\delta) = \gamma\ln(\gamma/\delta) + (1-\gamma)\ln((1-\gamma)/(1-\delta))$.

1. By definition, $D(\delta(1+\epsilon)||\delta) = \delta(1+\epsilon)\ln(1+\epsilon) + (1-\delta(1+\epsilon))\ln\frac{1-\delta(1+\epsilon)}{1-\delta}$. When $\epsilon = 0$, both sides are equal to 0. Now take the derivative with respect to ϵ and divide by δ on both sides, it suffices to show
$$\ln(1+\epsilon) - \ln\frac{1-\delta(1+\epsilon)}{1-\delta} \geq \frac{2}{3}\epsilon$$
when $0 \leq \epsilon \leq 1$ and $0 \leq \delta < 1/(1+\epsilon)$. The left-hand side is at least $\ln(1+\epsilon)$, and it suffices to prove $\ln(1+\epsilon) \geq \frac{2}{3}\epsilon$. This is true, since $\ln(1+\epsilon) - \frac{2}{3}\epsilon$ is concave, and $\ln(1+0) - \frac{2}{3}\cdot 0 = 0$ and $\ln(1+1) - \frac{2}{3}\cdot 1 > 0$.

2. By the first bullet, the left-hand side is larger when $\epsilon = 1$. Take the derivative with respect to ϵ and divide by δ on both sides, it suffices to show

$$\ln(1+\epsilon) - \ln \frac{1-\delta(1+\epsilon)}{1-\delta} \geq \frac{1}{3}$$

when $\epsilon > 1$ and $0 \leq \delta < 1/(1+\epsilon)$. This is true, since the left-hand side is at least $\ln(1+\epsilon) \geq \ln 2 > \frac{1}{3}$.

3. By definition, we have

$$D((1-\delta(1-\epsilon))||1-\delta) = (1-\delta(1-\epsilon)) \ln \frac{1-\delta(1-\epsilon)}{1-\delta} + \delta(1-\epsilon)\ln(1-\epsilon).$$

When $\epsilon = 0$, both sides are equal to 0. Take the derivative with respect to ϵ and divide by δ on both sides, it suffices to show

$$\ln \frac{1-\delta(1-\epsilon)}{1-\delta} - \ln(1-\epsilon) \geq \epsilon$$

when $0 \leq \epsilon, \delta < 1$. The left-hand side is at least $-\ln(1-\epsilon)$. When $\epsilon = 0$, both sides are 0. Take the derivative with respect to ϵ again, it suffices to show

$$\frac{1}{1-\epsilon} \geq 1,$$

which obviously holds.