A Wait-free Queue with Poly-logarithmic Step Complexity

ANONYMOUS AUTHOR(S)

We introduce a novel linearizable wait-free queue implementation using single-word CAS instructions. Previous lock-free queues all take $\Omega(p)$ steps per operation in the worst case, where p is the number of processes that can access the queue. We achieve $O(\log^2 p + \log q)$ steps per operation, where q is the size of the queue. **[[mention space usage; update time bound if space-efficient version takes more time per op]]**

Additional Key Words and Phrases: concurrent data structures, wait-free queues

1 INTRODUCTION

There has been a great deal of research in the past several decades on the design of linearizable, lock-free queues. Besides being a fundamental data structure, queues are used in significant concurrent applications, including OS kernels MP91 [25], memory reclamation ? [**?**], packet processing DPDK [6], and sharing resources or tasks. The lock-free MS-queue of Michael and Scott MS98 [26] is a classic shared queue implementation. IT uses a singly-linked list with pointers to the front and back nodes. To dequeue or enqueue an element, the front or back pointer is updated by a compare-and-swap (CAS) instruction. If this CAS fails, the operation must retry. In the worst case, this means that each successful CAS may cause all other processes to fail and retry, leading to an amortized step complexity of $\Omega(p)$ per operation. (To measure amortized step complexity of a lock-free implementation, we consider all possible finite executions and divide the number of steps in the execution by the number of operations initated in the execution.) Numerous papers have suggested modifications to the MS-queue DBLP:conf/opodis/HoffmanSS07,DBLP:conf/podc/KoganH14,DBLP:conf/ppopp/KoganP11,DBLP:journals/dc/Ladan [16, 21, 22, 24, 27, 28, 33], but all still have $\Omega(p)$ amortized step complexity as a result of contention on the front and back of the queue. Morrison and Afek DBLP:conf/ppopp/MorrisonA13 [29] called this the CAS retry problem. The same problem occurs in array-based implementations of queues DBLP:conf/iceccs/ColvinG0 [4, 13, 35, 39]. Solutions that tried to sidestep this problem using fetch&increment DBLP:conf/ppopp/MorrisonA13,DBLP:conf/p [29–31, 40] rely on slower mechanisms to handle worst-case executions and still have $\Omega(p)$ step complexity.

Many concurrent data structures that keep track of a set of elements also have an $\Omega(p)$ term in their step complexity, as observed by Ruppert Rup16 [34]. For example, lock-free lists FR04,Sha15 [12, 36], stacks Tre86 [38] and search trees EFHR14 [7] have an $\Omega(c)$ term in their step complexity, where c represents contention, the number of processes that access the data structure concurrently (which can be p in the worst case). Attiya and Fouren DBLP:conf/opodis/AttiyaF17 [2] proved that amortized $\Omega(c)$ steps per operation are indeed necessary for CAS-based implementations of lock-free bag data structures, which provide operations to insert an element or remove an arbitrary element (chosen non-deterministically). Since a queue trivially implements a bag, this lower bound also holds for queues. At first glance, this would seem to settle the question of the step complexity of lock-free queues. However, the lower bound leaves a loophole: it holds only if c is $O(\log\log p)$. Thus, the lower bound could be stated more precisely as an amortized bound of $\Omega(\min(c, \log\log p))$ steps per operation.

In this paper, we exploit this loophole. We show that it is in fact possible to implement a linearizable, lock-free queue whose step complexity does not depend linearly on p. Our implementation is the first to have step complexity that is polylogarithmic in p and in q, the number of elements stored in the queue. For simplicity of presentation, we first give an unbounded-space construction that uses $O(\log^2 p + \log q)$ steps per operation. Then, we show how to modify the implementation to use **[[$O(?)$ space]]** and **[[$O(?)$ steps per operation]]**. Both implementations use single-word CAS instructions and reasonably-sized words. Moreover, they use $O(\log p)$ CAS instructions per

2022. XXXX-XXXX/2022/12-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

operation in the worst case, whereas previous lock-free queues use $\Omega(p)$ CAS instructions in an amortized sense. For the space-bounded implementation, we unlink unneeded memory from our data structure. We do not address the orthogonal problem of reclaiming unlinked memory; we assume a safe garbage collector, such as the highly optimized one that Java provides.

Our queue uses a tournament tree where each process has its own leaf. Each process propagates its operations along the path from its leaf up to the root. The root of the tree provides an ordering for all the operations, and this ordering is used to linearize the operations and compute their responses. **[[Either here or in related work section, talk about previous usage of tournament tree and how ours differs from it]]** To ensure wait-freedom, we use a helping mechanism: a process propagating its operation to a node n helps propagate all operations from both of n's children. If we explicitly stored all operations in the nodes, this helping would become very costly. Instead, we use an implicit representation of collections of operations that allows for quickly merging two collections from the two children of a node, and for quickly accessing any particular operation in the collection. **[[Maybe say a little more about techniques used in the implementation, if space permits]]**

2 RELATED WORK

List-based Queues. The MS-queue MS98 [26] is a lock-free queue that has stood the test of time. A version of it is included in the standard Java Concurrency Package. See the paper that introduced it for a survey of the early history of concurrent queues. As mentioned above, the MS-queue suffers from the CAS retry problem because of contention at the front and back of the queue. Thus, it is lock-free but not wait-free and has an amortized step complexity of $\Theta(p)$ per operation.

A number of papers have described ways to reduce contention in the MS-queue. Moir et al. DBLP:conf/spaa/M [28] added an elimination array that allows an enqueue to pass its enqueued value directly to a concurrent dequeue when the queue is empty or when concurrent operations can be linearized to empty the queue. However, when there are p concurrent enqueues (and no dequeues), the CAS retry problem is still present. The baskets queue of Hoffman, Shalev, and Shavit DBLP:conf/opodis/HoffmanSS07 [16] attempts to reduce contention by grouping concurrent enqueues into baskets. An enqueue that fails its CAS is put in the basket with the enqueue that succeeded. Enqueues within a basket determine their order among themselves without having to access the back of the queue. However, if p concurrent enqueues are in the same basket the CAS retry problem occurs when they order themselves using CAS instructions. Both modifications still have $\Omega(p)$ amortized step complexity.

Kogan and Herlihy DBLP:conf/podc/KoganH14 [21] described how to use futures to improve the performance of the MS-queue. Operations return future objects instead of responses. Later, when an operation's response is needed, it can be evaluated using the corresponding future object. This allows batches of enqueues or dequeues to be done at once on an MS-queue. However, the implementation satisfies a weaker correctness condition than linearizability. Milman-Sela et al. MKLLP22 [27] extended this approach to allow batches to mix enqueues and dequeues. **[[Is this worth saying: They use some properties of the queue size before and after a batch, similar to a part of our work.]]** In the worst case, where operations require their response right away, batches have size 1, and both of these implementations behave like a standard MS-queue.

Ladan-Mozes and Shavit DBLP:journals/dc/Ladan-MozesS08 [24] presented an optimistic queue implementation. In the MS-queue, an enqueue requires two CAS steps. The optimistic queue uses a doubly-linked list to reduce the number of CAS instructions to one in the best case. Pointers in the doubly-linked list can be inconsistent, but are fixed when needed by traversing the list. Although this fixing is rare in practice, it yields an amortized complexity of $\Omega(qp)$ steps per operation for worst-case executions.

Kogan and Petrank [22] added Herlihy's helping technique [15] to the MS-queue to obtain a wait-free queue. Ramalhete and Correia [33] added a different helping mechanism. In both cases, the helping mechanisms only add to the step complexity of the MS-queue.

Array-Based Queues. Arrays can be used to implement queues with bounded capacity [4, 35, 39]. Dequeues and enqueues update indexes of the front and back elements using CAS instructions. Gidenstam, Sundell, and Tsigas [13] avoid the capacity constraint by using a linked list of arrays. These solutions also use $\Omega(p)$ steps per operation due to the CAS retry problem.

Morrison and Afek [29] also used a linked list of (circular) arrays. To avoid the CAS retry problem, concurrent operations try to claim spots in an array using fetch&increment instructions. If livelock between enqueues and a racing dequeue prevent enqueues from claiming a spot, the enqueues fall back on using a CAS to add a new array to the linked list, and the CAS retry problem reappears. Similarly, other queues [30, 31, 40] used the fast-path slow-path methodology of Kogan and Petrank [23]. The fast path has good performance and the slow path guarantees termination. In worst-case executions, where processes must execute the slow path, these implementations also take $\Omega(p)$ steps per operation, due either to the CAS retry problem or helping mechanisms.

Universal Constructions. One can also build a queue using a universal construction [15]. Jayanti [17] observed that the universal construction of Afek, Dauber, and Touitou [1] can be modified to use $O(\log p)$ steps per operation, assuming that words can store $\Omega(p \log p)$ bits. (Thus, in terms of operations on reasonably-sized $O(\log p)$-bit words, their construction would take $\Omega(p \log p)$ steps per operation.) Fatourou and Kallimanis [10] used their universal construction based on fetch&add and LL/SC instructions to implement a queue, but its step complexity is also $\Omega(p)$. **[[Double check this.]]**

Restricted Queues. David introduced the first sublinear-time queue [5], but it works only for a single enqueuer. It uses fetch&increment and swap instructions and takes $O(1)$ steps per operation, but uses unbounded memory. Bounding the space increases the steps per operation to $\Omega(p)$.

Jayanti and Petrovic introduced a wait-free poly-logarithmic queue [18], but it works only for a single dequeuer. Our implementation uses their idea of having a tournament tree among processes to agree on the linearization of operations.

[[Should we mention https://arxiv.org/abs/2103.11926 , which seems to have a queue for 2 enqueuers and 2 dequeuers ? May not be worth mentioning because then p is a constant, so complexity in terms of p doesn't make sense.]]

Other Primitives. Khanchandani and Wattenhofer [20] gave a wait-free queue implementation with $O(\sqrt{p})$ step complexity using non-standard synchronization primitives called half-increment and half-max, which can be viewed as particular kinds of double-word read-modify-write operations. They use this as evidence that their primitives can be more efficient than CAS since previous CAS-based queues all required $\Omega(p)$ step complexity. Our new implementation counters this argument.

Fetch&Increment Objects. There are sublinear-time implementations of another object with consensus number 2. Two papers [8, 9] implemented fetch&increment objects from LL/SC objects using a polylogarithmic number of steps per operation. Our approach is similar to that of Ellen, Ramachandran and Woelfel [8], which uses a tournament tree to keep track of the fetch&increments that have been performed, as in the universal construction of [1]. However, our construction requires more intricate data structures to represent sets of operations, since the state of a queue cannot easily be represented as succinctly as the single-word state of a fetch&increment object.

scriptQ 3 QUEUE IMPLEMENTATION

We begin with an overview of the algorithm. We use a *tournament tree* to agree on a total ordering of the operations performed on the queue. The tree is a static binary tree of height $\lceil \log_2 p \rceil$ with one leaf assigned to each process. Each node of the tree stores an array of *blocks*, where each block represents a sequence of enqueues and a sequence of dequeues. To perform an operation on the queue, a process P creates a block containing that single operation and appends it to the blocks array in P's leaf. Then, P attempts to propagate the operation along the path from that leaf to the root of the tree. At each node v along this path, P combines information from the blocks in v's children into a single block and attempts to append this block to v's array using a CAS instruction. This cooperative approach is sufficient to ensure that if P fails to CAS a new block into v's array twice, then its operation has been propagated to v by some other process. We define a total order on all operations that have been propagated to the root, which is used to linearize the operations. In this section, we assume for simplicity that each node has an infinite blocks array. We describe in Section reducing 6 how to replace the infinite array by a representation that uses bounded space.

If P's operation is an enqueue, it can terminate when the operation has been propagated to the root. If P's operation is a dequeue, P must use information in the tree to compute the value that the dequeue must return. To do this, P first determines which block in the root contains its operation (since the operation may have been propagated to the root by some other process). Then, P determines whether the queue is empty when its dequeue is linearized. If so, it returns nil. If not, P computes the rank r of its dequeue in the linearization ordering among all dequeues that do not return nil, finds the rth enqueue in the linearization and returns that enqueue's value.

The primary challenge is thus figuring out what information to store in each block so that the following tasks can be done efficiently (in a polylogarithmic number of steps).

- Construct a block for node v that represents the operations contained in $O(p)$ consecutive blocks in v's children.
- Given an operation in a leaf that has been propagated to the root, find that operation's location in the root's blocks array.
- Given the location of a dequeue operation in the root's blocks array, find the enqueue whose value it should return (or determine that the queue is empty when the dequeue is linearized).

An operation can only terminate after it a block containing it has been appended to the root's blocks array. So, if an operation op_1 terminates before another operation op_2 begins, op_1 will be in an earlier block than op_2 in root's blocks array. Our linearization orders operations according to the block they belong to in the root's blocks array. Operations that appear in the same block are necessarily concurrent, so they can be ordered arbitrarily. We next describe how to order them so that computing results of dequeue operations is efficient.

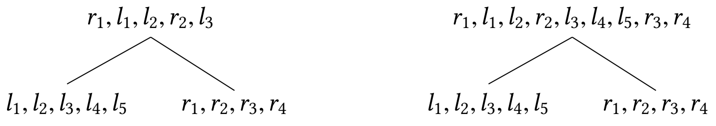
Each block B in a node v represents several consecutive blocks in each of v's children. The blocks in the children are called the *direct subblocks* of B. A block B' in a descendant of v is a *subblock* of B if it is a direct subblock of B or a subblock of a direct subblock of B. Recall that blocks in the leaves contain a single operation. A block B represents the set of operations in all of B's subblocks in leaves of the tree. We order the operations within a block as follows.

[[At what point would a figure illustrating of the tournament tree be useful?]]

———

To propagate operations to a node n in the tree, a process observes the operations in both of n's children that are not already in n, merges them to create an ordering and then tries to append the ordering to the sequence stored in n. We call this procedure n.`Refresh` (see Figure fig::propagstep 1). A `Refresh` on n with a successful append helps to propagate their operations up to n. We shall prove that if a process invokes `Refresh` on the node n two times and fails to append the new operations to n

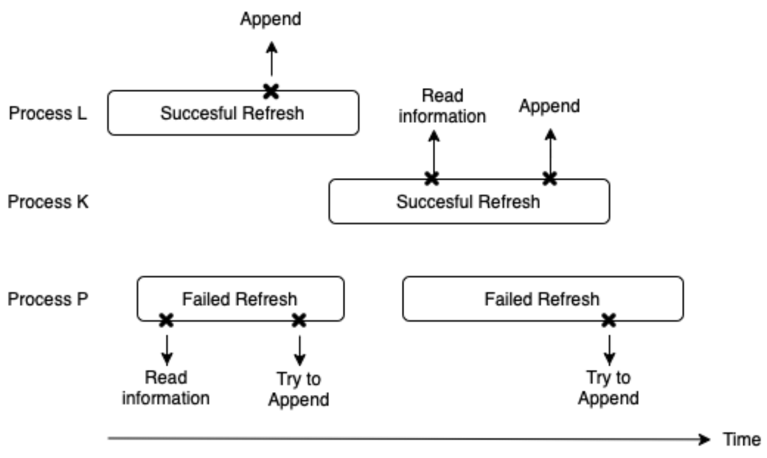
both times, the operations that were in n's children before the first Refresh are guaranteed to be in n after the second failed Refresh. We sketch the argument here.


r_1, l_1, l_2, r_2, l_3
l_1, l_2, l_3, l_4, l_5
r_1, r_2, r_3, r_4
$r_1, l_1, l_2, r_2, l_3, l_4, l_5, r_3, r_4$
l_1, l_2, l_3, l_4, l_5
r_1, r_2, r_3, r_4

(a) Before the Refresh. (b) After the Refresh.

pagstep

Fig. 1. Before and after a n.Refresh with a successful append. Operations propagating from the left child are labelled with l and from the right child with r.



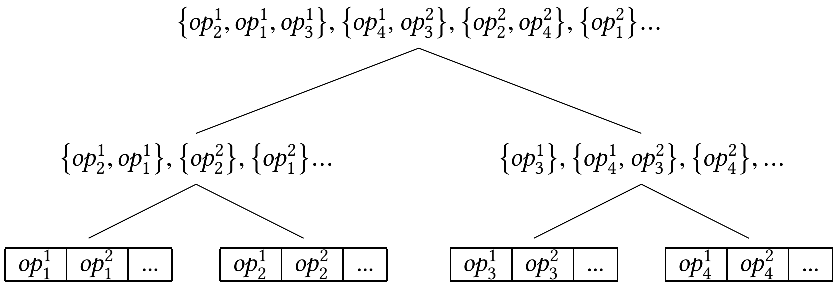
Refresh

Fig. 2. Time relations between the concurrent successful Refreshes and the two consecutive failed Refreshes.

We use CAS (Compare&Swap) instructions to implement the Refresh's attempt to append described in the previous paragraph. The second failed Refresh of P is assuredly concurrent with a successful Refresh that has read its information after the invocation of the first failed Refresh (see Figure 2). This is because some process L does a successful append during P's first failed attempt, and some process K performs a Refresh that reads its information after L's append and then performs a successful append during P's second failed Refresh. Process K's Refresh helps to append the new operations that were in n's children before P's first failed Refresh, in case they were not already appended. After a process appends its operation into its leaf it can call Refresh on the path up to the root at most two times on each node. So, with $O(\log p)$ CASes an operation can ensure it appears in the linearization. This cooperative solution allows us to overcome the CAS Retry Problem.

fig::simpleDoubleRefresh

It is not efficient to explicitly store the sequence of operations in each node because each operation would have to be copied all the way up to the root; doing this would not be possible in poly-logarithmic time. Instead we use an implicit representation of the operations propagated together. Furthermore, we do not need to maintain an ordering on operations propagated together in a node until they have reached the root. It is sufficient to only keep track of sets of operations propagated together in each `Refresh` and then define the linearization ordering only in the root (see Figure fig::set 3). Achieving a constant-sized implicit representation of operations in a `Refresh` allows us to do `CAS` instructions on fixed-size objects in each `Refresh`. To do that, we introduce *block*s. A block stores information about the operations propagated by a `Refresh`. It contains the number of operations from the left and the right child propagated to the node by the `Refresh` procedure. See Figure fig:block 4 for an example. A node stores an array of blocks of operations propagated up to it. A propagate step aggregates the new blocks in the children into a new block and stores the new block in the parent. We call the aggregated blocks *subblocks* of the new block and the new block the *superblock* of them. In each `Refresh` there is at most one operation from each process trying to be propagated, because one process cannot invoke two operations concurrently. Thus, there are at most p operations in a block. Furthermore, since the operations in a `Refresh` step are concurrent we can linearize them among themselves in any order we wish, because if two operations are read in one successful `Refresh` step in a node they are going to be propagated up to the root together. Our choice is to put the operations propagated from the left child before the operations propagated from the right child. In this way, if we know the number of operations from the left child and the number of operations from the right child in a block, we have a complete ordering of the operations.

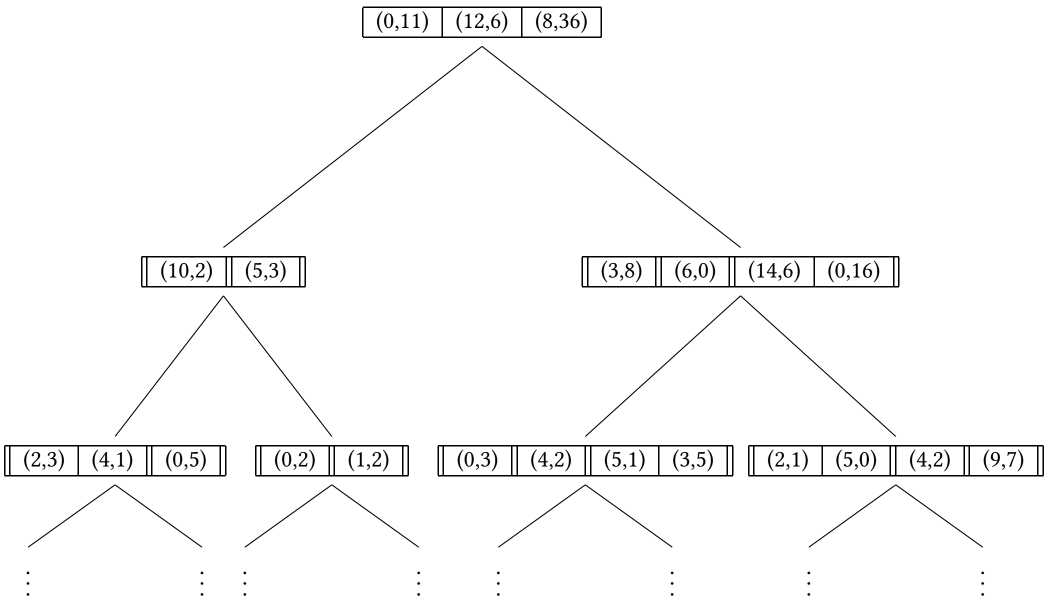

$\{op_2^1, op_1^1, op_3^1\}, \{op_4^1, op_3^2\}, \{op_2^2, op_4^2\}, \{op_1^2\}\ldots$
$\{op_2^1, op_1^1\}, \{op_2^2\}, \{op_1^2\}\ldots$
$\{op_3^1\}, \{op_4^1, op_3^2\}, \{op_4^2\}, \ldots$
op_1^1 op_1^2 ...
op_2^1 op_2^2 ...
op_3^1 op_3^2 ...
op_4^1 op_4^2 ...

ig::set

Fig. 3. Leaves are for processes P_1 to P_4 from left to right. In each internal node one can arbitrarily linearize the sets of concurrent operations propagated together in a `Refresh`. For example op_4^1 and op_3^2 have propagated together in one `Propagate` step and they will be propagated up to the root together. Since their execution time intervals overlap, they can be linearized in any order.

So far, we have a shared tree that processes use to agree on the implicit ordering stored in its root. With this agreement on the linearization ordering, we can design a universal construction; for a given object, we can perform an operation op by applying all the operations up until op in the root on a local copy of the object and then returning the response for op. However, this approach is not enough for an efficient queue. We show that we can build an efficient queue if we can compute two things about the ordering in the root: (1) the ith propagated operation and (2) the rank of a propagated operation in the linearization. We explain how to implement (1) and (2) in poly-logarithmic steps.

After propagating an operation op to the root, processes can find out information about the linearization ordering using (1) and (2). To get the ith operation in the root, we find the block B containing the ith operation in the root, and then recursively find the subblock of B in the descendent of the root that contains that ith operation. When we reach a block in a leaf, the operation is explicitly stored there. To make this search faster, instead of iterating over all blocks in the node, we store the prefix sum of the number of elements in the blocks sequence to permit a binary search for the required block. We also store pointers to determine the range of subblocks of

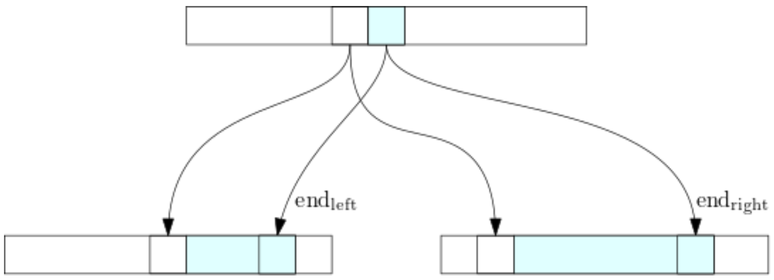


g:block

Fig. 4. Using blocks to represent operations. Blocks between two lines || are propagated together to the parent. Each block consists of a pair (left, right) indicating the number of operations from the left and the right child, respectively. For example, (12,6) in the root contains (10,2) from the left child and (6,0) from the right child. The third block in the root (8,36) is created by merging (5,3) from the left child and (14,6) and (0,16) from the right child. (5,3) is the superblock of (0,5) and (1,2) and (5,1),(3,5) and (4,2) are subblocks of (14,6).

a block to make the binary search faster. In each block, we store the prefix sum of operations from the left child and from the right child. Moreover, for each block, we store two attributes $\mathtt{end}_{\mathtt{left}}$ and $\mathtt{end}_{\mathtt{right}}$, the indices of the last left and right subblock (see Figure fig::pointer 5). We know a block size is at most p, so binary search takes at most $O(\log p)$ time, since the $\mathtt{end}_{\mathtt{left}}$ and $\mathtt{end}_{\mathtt{right}}$ indices of a block and its previous block reduce the search range size to $O(p)$.

To compute the rank in the root of an operation in a leaf, we need to find the superblock of the block that the operation is in. After a block is installed in a node we store the approximate index of its superblock in it to make this faster.



pointer

Fig. 5. Each block stores the index of its last subblock in each child.

In an execution on a queue where no dequeue operation returns `null`, the kth dequeue returns the argument of the kth enqueue. In the general case a dequeue returns `null` if and only if the queue size after the previous operation is 0. We refer to such a dequeue as a *null dequeue*. If the dequeue is the kth non-null dequeue, it returns the argument of the kth enqueue. Having the size of the queue after an operation we can compute the number of non-null dequeues from the number of enqueues before the operation. So, if we store the size of the queue after each block of operations in the root, we can compute the index of the enqueue whose argument is the response to a given dequeue in constant time.

In our case of implementing a queue, a process only needs to compute the rank of a Dequeue and get an Enqueue with a specific position. We know we can linearize operations in a block in any order; here, we choose to put Enqueue operations in a block before Dequeue operations. Consider the following operations, where operations in a cell are concurrent.

Deq	Enq(5), Enq(2), Enq(1), Deq	Enq(3), Deq	Enq(4), Deq, Deq, Deq, Deq

Table 1. An execution on a queue separated by the operations in the root blocks.

The Dequeue operations return `null, 5, 2, 1, 3, 4, null`, respectively. Now, we claim that by knowing the size of the queue, we can compute the rank of the required Enqueue for any non-null Dequeue. We apply this approach to blocks; if we store the size of the queue after each block of operations, we can compute the index of each Dequeue's result in O(1) steps.

	Deq	Enq(5), Enq(2), Enq(1), Deq	Enq(3), Deq	Enq(4), Deq, Deq, Deq, Deq
#Enqs	0	3	1	1
#Deqs	1	1	1	4
Size at end	0	2	2	0

Table 2. Augmented history of operation blocks on the queue.

The size of the queue after the bth block in the root could be computed as

$$\max\Big(\text{size after } (b-1)\text{th block} + \text{\#Enqueues in } b\text{th block} - \text{\#Dequeues in } b\text{th block}, 0\Big).$$

Moreover, the total number of non-null dequeues in blocks $1, 2, \ldots, b$ in the root is

$$\sum_{i=1}^{b} \text{\#Enqueues in } i\text{th block} - \text{size after } b\text{th block}.$$

Given a Dequeue is in block B, its response is the argument of the Enqueue whose rank is

$$\text{\#non-null Dequeues in blocks } 1, 2, \ldots, b-1 + \text{index of the Dequeue among } B\text{'s Dequeue}$$

if

$$\Big(\text{size of the queue after } b-1\text{th block} + \text{\#Enqueues in } b\text{th block} - \text{index of Dequeue in } B\text{'s Dequeues}\Big) \geq 0.$$

Otherwise, the response would be `null`.

3.1 Details of the Implementation

Section algQ 3.2 gives the pseudocode for the queue implementation. It uses the following two types of objects.

Node. In each Node we store pointers to its parent and its left and right child, an array of `Blocks` called `blocks` and the index `head` of the first empty entry in `blocks`.

Block. The information stored in a Block depends on whether the Block is in an internal node or a leaf. If it is in a leaf, we use a LeafBlock which stores one operation. If a block B is in an internal node n, then it contains subblocks in the left and right children of n. The left subblocks of B are some consecutive blocks in the left child of n starting from where the left subblocks of the block prior to B ended. The right subblocks of B are defined similarly. In each block we store four essential fields that implicitly summarize which operations are in the block $\texttt{sum}_{\texttt{enq-left}}$, $\texttt{sum}_{\texttt{deq-left}}$, $\texttt{sum}_{\texttt{enq-right}}$, $\texttt{sum}_{\texttt{deq-right}}$. The $\texttt{sum}_{\texttt{enq-left}}$ field is the total number of Enqueue operations in the blocks before the last subblock of B in the left child. The other fields' semantics are similar. The $\texttt{end}_{\texttt{left}}$ and $\texttt{end}_{\texttt{right}}$ field store the index of the last subblock of a block in the left and the right child, respectively. The approximate index of the superblock of non-root blocks is stored in their super field. The size field in a block in the root node stores the size of the queue after the operations in the block have been performed.

We now describe the routines used in the implementation.

Enqueue(e). An Enqueue operation does not return a response, so it is sufficient to propagate the Enqueue operation to the root and then use its position in the linearization for future Dequeue operations. Enqueue(e) creates a LeafBlock with element $= e$, sets its $\texttt{sum}_{\texttt{enq}}$ and $\texttt{sum}_{\texttt{deq}}$ fields and then appends it to the tree.

Dequeue(). Dequeue creates a LeafBlock, sets its $\texttt{sum}_{\texttt{enq}}$ and $\texttt{sum}_{\texttt{deq}}$ fields and appends it to the tree. Then, it computes the position of the appended Dequeue operation in the root using IndexDequeue and after that finds the response of the Dequeue by calling FindResponse.

Append(B). The head field is the index of the first empty slot in blocks in a LeafBlock. There are no multiple write accesses on head and blocks in a leaf because only the process that the leaf belongs to appends to it. Append(B) adds B to the end of the blocks field in the leaf, increments head and then calls Propagate on the leaf's parent. When Propagate terminates, it is guaranteed that the appended block is a subblock of a block in the root.

Propagate(). Propagate on node n uses the double refresh idea described earlier and invokes two Refreshes on n in Lines 52 and 53. Then, it invokes Propagate on n.parent recursively until it reaches the root.

Refresh() and Advance(). The goal of a Refresh on node n is to create a block of n's children's new blocks and append it to n.blocks. The variable h is read from n.head at Line 60. The new block created by Refresh will be inserted into n.blocks[h]. Lines 61–66 of n.Refresh help to Advance n's children. Advance increments the children's head if necessary and sets the super field of their most recently appended blocks. The reason behind this helping is explained later when we discuss IndexDequeue. After helping to Advance the children, a new block called new is created in Line 67. Then, if new is empty, Refresh returns true because there are no new operations to propagate, and it is unnecessary to add an empty block to the tree. Later we will use the fact that all blocks contain at least one operation. Line 70 tries to install new. If it was successful, all is good. If not, it means someone else has already put a block in n.blocks[h]. In this case, Refresh helps advance n.head to h+1 and update the super field of n.blocks[h] at Line 71.

CreateBlock(). n.CreateBlock(h) is used by Refresh to construct a block containing new operations of n's children. The block new is created in Line 80 and its fields are filled similarly for both left and right directions. The variable $\texttt{index}_{\texttt{prev}}$ is the index of the block preceding the first subblock in the child in direction dir that is aggregated into new. Field new.$\texttt{end}_{\texttt{dir}}$ stores the index of the rightmost subblock of new in the child. Then $\texttt{sum}_{\texttt{enq-dir}}$ is computed from the sum of the number of Enqueue operations in the new block from direction dir and the value stored in *n*.`blocks[h-1]`.$\texttt{sum}_{\texttt{enq-dir}}$. The field $\texttt{sum}_{\texttt{deq-dir}}$ is computed similarly. Then, if `new` is going to be installed in the `root`, the `size` field is also computed.

`IndexDequeue`(b, i). A call to *n*.`IndexDequeue`(b, i) computes the block and the rank within the block in the root of the *i*th `Dequeue` of the *b*th block of *n*. Let R_n be the successful `Refresh` on node *n* that did a successful `CAS(null, ` B`)` into `n.blocks[`b`]`. Let *par* be *n*.`parent`. Without loss of generality, assume for the rest of this section that *n* is the left child of *par*. Let R_{par} be the first successful *par*.`Refresh` that reads some value greater than *b* for `left.head` and therefore contains *B* as a subblock of its created block in Line invokeCreateBlock 67. Let *j* be the index of the block that R_{par} puts in *par*.`blocks`.

Since the index of the superblock of *B* is not known until *B* is propagated, R_n cannot set the `super` field of *B* while creating *B*. One approach for R_{par} is to set the `super` field of *B* after propagating *B* to *par*. This solution would not be efficient because there might be up to *p* subblocks that R_{par} propagated, which need to update their `super` field. However, intuitively, once *B* is installed, its superblock is going to be close to *n*.`parent.head` at the time of installation. If we know the approximate position of the superblock of *B* then we can search for the real superblock when it is needed. Thus, *B*.`super` does not have to be the exact location of the superblock of *B*, but we want it to be close to *j*. We can set *B*.`super` to *par*.`head` while creating *B*, but the problem is that there might be many `Refresh`es on *par* that could happen after R_n reads *par*.`head` and before propagating *B* to *par*. If R_n sets *B*.`super` to *par*.`head` after appending *B* to *n*.`blocks` (Line setSuper1 76), R_n might go to sleep at some time after installing *B* and before setting *B*.`super`. In this case, the next `Refresh`es on *n* and *par* help fill in the value of *B*.`super`.

Block *B* is appended to *n*.`blocks[`b`]` on Line cas 70. After appending *B*, *B*.`super` is set on Line setSuper1 76 of a call to `Advance` from *n*.`Refresh` by the same or another process or by Line helpAdvance 64 of a *n*.`parent.Refresh`. We shall show that this is sufficient to ensure that *B*.`super` differs from the index of *B*'s superblock by at most 1.

`FindResponse`(b, i). To compute the response of the *i*th `Dequeue` in the *b*th block of the root Line checkEmpty 19 computes whether the queue is empty or not. If there are more `Dequeue`s than `Enqueue`s the queue would become empty before the requested `Dequeue`. If the queue is not empty, Line computeE 22 computes the rank *e* of the `Enqueue` whose argument is the response to the `Dequeue`. Knowing the response is the *e*th `Enqueue` in the root (which is before the *b*th block), we find the block and position containing the `Enqueue` operation using `DoublingSearch` and after that `GetEnqueue` finds its `element`.

`GetEnqueue`(b, i) *and* `DoublingSearch`(e, end). We can describe an operation in a node in two ways: the rank of the operation among all the operations in the node or the index of the block containing the operation in the node and the rank of the operation within that block. If we know the block and rank within the block of an operation, we can find the subblock containing the operation and the operation's rank within that subblock in poly-log time. To find the response of a `Dequeue`, we know about the rank of the response `Enqueue` in the root (e in Line computeE 22). We also know the eth `Enqueue` is in `root.blocks[1..end]`. `DoublingSearch` uses doubling to find the range that contains the answer block (Lines dsearchStart dsearchEnd 38–41) and then tries to find the required indices with a binary search (Line dsearchBinarySearch 42). A call to *n*.`GetEnqueue`(b, i) returns the `element` of the *i*th enqueue in the *b*th block of *n*. The range of subblocks of a block is determined using the $\texttt{end}_{\texttt{left}}$ and $\texttt{end}_{\texttt{right}}$ fields of the block and its previous block. Then, the subblock is found using binary search on the $\texttt{sum}_{\texttt{enq}}$ field (Lines leftChildGet 99 and rightChildGet 103).

algQ

3.2 Pseudocode

We present our algorithm in pseudocode. page 22 contains the description of the fields in the tree nodes and the blocks. The value of any uninitialized field is `null`. page 23 contains major routines and the rest of this section consists of the auxiliary routines. The abbreviations below are used in the pseudocode and the proof of correctness.

- $\texttt{blocks}[b]\texttt{.sum}_{\texttt{x}}\texttt{=blocks}[b]\texttt{.sum}_{\texttt{x-left}}\texttt{+blocks}[b]\texttt{.sum}_{\texttt{x-right}}$ (for internal blocks where $b \geq 0$ and x $\in$ {enq, deq})
- $\texttt{blocks}[b]\texttt{.num}_{\texttt{x}}\texttt{=blocks}[b]\texttt{.sum}_{\texttt{x}}\texttt{-blocks}[b-1]\texttt{.sum}_{\texttt{x}}$ (for all blocks where $b > 0$ and x $\in$ {enq, deq, enq-left, enq-right, deq-left, deq-right})

Algorithm Tree Fields Description

```
◇ Shared
• A binary tree of Nodes with one leaf for each process. root is the root node.
◇ Local
• Node leaf: process's leaf in the tree.
▶ Node
• *Node left, right, parent : Initialized when creating the tree.
• Block[] blocks : Initially blocks[0] contains an empty block with all fields equal to 0.
• int head= 1: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.
▶ Block
• int super : approximate index of the superblock, read from parent.head when appending the block to the node
▶ InternalBlock extends Block
• int $end_{left}$, $end_{right}$ : indices of the last subblock of the block in the left and right child
• int $sum_{enq-left}$: #enqueues in left.blocks[1..$end_{left}$]
• int $sum_{deq-left}$: #dequeues in left.blocks[1..$end_{left}$]
• int $sum_{enq-right}$: #enqueues in right.blocks[1..$end_{right}$]
• int $sum_{deq-right}$: #dequeues in right.blocks[1..$end_{right}$]
▶ LeafBlock extends Block
• Object element : Each block in a leaf represents a single operation. If the operation is enqueue(x) then
  element=x, otherwise element=null.
• int $sum_{enq}$, $sum_{deq}$ : # enqueue, dequeue operations in this block and its previous blocks in the leaf
▶ RootBlock extends InternalBlock
• int size : size of the queue after performing all operations in this block and its previous blocks in the root
```

4 PROOF OF CORRECTNESS

We adopt linearizability as our definition of correctness. In our case, where we create the linearization ordering in the root, we need to prove (1) the ordering is legal, i.e, for every execution on our queue if operation op_1 terminates before operation op_2 then op_1 is linearized before operation op_2 and (2) if we do operations sequentially in their linearization order, operations get the same results as in our queue. The proof is structured like this. First, we define and prove some facts about blocks and the node's `head` field. Then, we introduce the linearization ordering formally. Next, we prove double `Refresh` on a node is enough to propagate its children's new operations up to the node, which is used to prove (1). After this, we prove some claims about the size and operations of each block, which we use to prove the correctness of `DoublingSearch()`, `GetEnqueue()` and `IndexDequeue()`. Finally, we prove the correctness of the way we compute the response of a dequeue, which establishes (2).

4.1 Basic Properties

In this subsection, we talk about some properties of blocks and fields of the tree nodes.

Algorithm Queue

```
void Enqueue(Object e)                          ▷ Creates a block with element e and adds it to the tree.
    block newBlock= new(LeafBlock)
    newBlock.element= e
    newBlock.sum$_{enq}$= leaf.blocks[leaf.head].sum$_{enq}$+1
    newBlock.sum$_{deq}$= leaf.blocks[leaf.head].sum$_{deq}$
    leaf.Append(newBlock)
end Enqueue

▷ Creates a block with null value element, appends it to the tree and returns its response.
Object Dequeue()
    block newBlock= new(LeafBlock)
    newBlock.element= null
    newBlock.sum$_{enq}$= leaf.blocks[leaf.head].sum$_{enq}$
    newBlock.sum$_{deq}$= leaf.blocks[leaf.head].sum$_{deq}$+1
    leaf.Append(newBlock)
    <b, i>= IndexDequeue(leaf.head, 1)
    output= FindResponse(b, i)                  deqRest
    return output
end Dequeue

▷ Returns the response to $D_i(root, b)$, the ith Dequeue in root.blocks[b].
element FindResponse(int b, int i)
    if root.blocks[b-1].size + root.blocks[b].num$_{enq}$ - i < 0 then    ▷ Check if the queue is empty.    ckEmpty
        return null                             urnNull
    else                                        ▷ The response is $E_e(root)$, the eth Enqueue in the root.
        e= i + (root.blocks[b-1].sum$_{enq}$-root.blocks[b-1].size)    omputeE
        return root.GetEnqueue(root.DoublingSearch(e, b))    dAnswer
    end if
end FindResponse
```

Algorithm Node

```
⇝ Precondition: blocks[start..end] contains a block with sum$_{enq}$ greater than or equal to x
▷ Does a binary search for the value x of sum$_{enq}$ field and returns the index of the leftmost block in
▷ blocks[start..end] whose sum$_{enq}$ is ≥ x.
int BinarySearch(int x, int start, int end)
    while start<end do
        int mid= floor((start+end)/2)
        if blocks[mid].sum$_{enq}$<x then
            start= mid+1
        else
            end= mid
        end if
    end while
    return start
end BinarySearch
```

A block is an object storing some statistics, as described in Algorithm Queue. A block in a node implicitly represents a set of operations.

Definition 4.1 (Ordering of a block in a node). Let B be n.blocks[b] and B' be n.blocks[j]. We call i the *index* of block B. Block B is *before* block B' in node n if and only if $i < j$.

Algorithm Root

```
    ⇝ Precondition: root.blocks[end].sum_enq ≥ e
    ▷ Returns <b,i> such that $E_e$(root) = $E_i$(root, b), i.e., the eth Enqueue in the root is the ith Enqueue within ▷ the
    bth block in the root.
37: <int, int> DoublingSearch(int e, int end)
38:     start= end-1                                          chStart
39:     while root.blocks[start].sum_enq>=e do                oubling
40:         start= max(start-(end-start), 0)                  archEnd
41:     end while
42:     b= root.BinarySearch(e, start, end)                   ySearch
43:     i= e- root.blocks[b-1].sum_enq                        omputei
44:     return <b,i>
45: end DoublingSearch
```

Algorithm Leaf

```
46: void Append(block B)                 ▷ Only called by the owner of the leaf.
47:     blocks[head]= B                                       endLeaf
48:     head= head+1                                          pendEnd
49:     parent.Propagate()
50: end Append
```

Algorithm Node

```
    ▷ n.Propagate propagates operations in this.children up to this when it terminates.
51: void Propagate()
52:     if not Refresh() then                                 Refresh
53:         Refresh()                                         Refresh
54:     end if
55:     if this is not root then
56:         parent.Propagate()
57:     end if
58: end Propagate

    ▷ Creates a block containing new operations of this.children, and then tries to append it to this.
59: boolean Refresh()
60:     h= head                                               eadHead
61:     for each dir in {left, right} do                      pChild1
62:         h_dir= dir.head                                   ildHead
63:         if dir.blocks[h_dir]!=null then                   notNull
64:             dir.Advance(h_dir)                            Advance
65:         end if
66:     end for                                               pChild1
67:     new= CreateBlock(h)                                   teBlock
68:     if new.num==0 then return true                        addOP
69:     end if
70:     result= blocks[h].CAS(null, new)                      cas
71:     this.Advance(h)                                       advance
72:     return result
73: end Refresh
```

Next, we show that the value of head in a node can only be increased. By the termination of a Refresh, head has been incremented by the process doing the Refresh or by another process.

Observation 1. *For each node n, n.*head *is non-decreasing over time.* ingHead

Algorithm Node

```
74: void Advance(int h)                          ▷ Sets blocks[h].super and increments head from h to h+1.
75:     $h_p$= parent.head                                                          entHead
76:     blocks[h].super.CAS(null, $h_p$)                                           tSuper1
77:     head.CAS(h, h+1)                                                            entHead
78: end Advance

79: Block CreateBlock(int i)                     ▷ Creates and returns the block to be installed in blocks[i].
80:     block new= new(InternalBlock)                                               ewBlock
81:     for each dir in {left, right} do
82:         $index_{prev}$= blocks[i-1].$end_{dir}$                                  revLine
83:         new.$end_{dir}$= dir.head-1          ▷ new contains dir.blocks[blocks[i-1].$end_{dir}$..dir.head-1].   astLine
84:         $block_{prev}$= dir.blocks[$index_{prev}$]
85:         $block_{last}$= dir.blocks[new.$end_{dir}$]
86:         new.$sum_{enq-dir}$= blocks[i-1].$sum_{enq-dir}$ + $block_{last}$.$sum_{enq}$ - $block_{prev}$.$sum_{enq}$   EnqLeft
87:         new.$sum_{deq-dir}$= blocks[i-1].$sum_{deq-dir}$ + $block_{last}$.$sum_{deq}$ - $block_{prev}$.$sum_{deq}$   nqRight
88:     end for
89:     if this is root then
90:         new.type= InternalBlock-->RootBlock
91:         new.size= max(root.blocks[i-1].size + new.$num_{enq}$- new.$num_{deq}$, 0)   eLength
92:     end if
93:     return new
94: end CreateBlock
```

Proof. The claim follows trivially from the code since head is only changed by incrementing in Line 77 of Advance. □

headInc

Lemma 4.2. *Let R be an instance of* Refresh *on a node n that creates a non-empty block* new *(*new.sum $\neq 0$*). After R terminates, n.*head *is greater than the value read in Line 60 of R.*

Proof. If the CAS in Line 77 is successful, then the claim holds. Otherwise, n.head has changed from the value read in Line 60. By Observation 1 this means another process has incremented n.head. □

Now we show n.blocks[n.head] is either the last block written into node n or the first empty block in n.

osition

Invariant 1 (headPosition). *If the value of n.*head *is h then n.*blocks[i] = null *for $i > h$ and n.*blocks[i] $\neq$ null *for $0 \leq i < h$.*

Proof. Initially the invariant is true since n.head $= 1$, n.blocks[0] $\neq$ null and n.blocks[x] = null for every $x > 0$. The truth of the invariant may be affected by writing into n.blocks or incrementing n.head. We show that if the invariant holds before such a change, then it still holds after the change.

In the algorithm, n.blocks is modified only on Line 70, which updates n.blocks[h] where h is the value read from n.head in Line 60. Since the CAS in Line 70 is successful, it means n.head has not changed from h before doing the CAS; if n.head had changed before the CAS then it would be greater than h by Observation 1 and hence n.blocks[h]$\neq$null and by the induction hypothesis, so the CAS would fail. Writing into n.blocks[h] when $h = n$.head preserves the invariant, since the claim does not talk about the content of n.blocks[n.head].

The value of n.head is modified only in Line 77 of Advance. If n.head is incremented to $h+1$ it is sufficient to show n.blocks[h]$\neq$null. Advance is called in Lines 64 and 71. For Line

Algorithm Node

```
    ⇝ Precondition: blocks[b].num_enq ≥i≥ 1
95:  element GetEnqueue(int b, int i)                         ▷ Returns the element of E_i(this, b).
96:      if this is leaf then
97:          return blocks[b].element                                                      aseCase
98:      else if i <= blocks[b].num_enq-left then     ▷ E_i(this, b) is in the left child of this node.   OrRight
99:          subblockIndex= left.BinarySearch(i+blocks[b-1].sum_enq-left, blocks[b-1].end_left+1,   hildGet
                         blocks[b].end_left)
100:         return left.GetEnqueue(subblockIndex, i)
101:     else
102:         i= i-blocks[b].num_enq-left
103:         subblockIndex= right.BinarySearch(i+blocks[b-1].sum_enq-right, blocks[b-1].end_right+1,   hildGet
                          blocks[b].end_right)
104:         return right.GetEnqueue(subblockIndex, i)
105:     end if
106: end GetEnqueue

    ⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num_deq ≥i.
107: <int, int> IndexDequeue(int b, int i)          ▷ Returns <x, y> if D_i(this, b) = D_y(root, x).
108:     if this is root then
109:         return <b, i>                                                                 aseCase
110:     else
111:         dir= (parent.left==n ? left: right)
112:         superblockIndex= parent.blocks[blocks[b].super].sum_deq-dir > blocks[b].sum_deq ?   teSuper
                            blocks[b].super: blocks[b].super+1
113:         if dir is left then                                                          erStart
114:             i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-left       ousLeft
115:         else                                                                         erRight
116:             i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-right      usRight
117:             i+= parent.blocks[superblockIndex].num_deq-left                           reRight
118:         end if                                                                       uperEnd
119:         return this.parent.IndexDequeue(superblockIndex, i)
120:     end if
121: end IndexDequeue
```

helpAdvance 64, n.blocks[h] ≠ null because of the if condition in Line ifHeadnotNull 63. For Line advance 71, Line cas 70 was finished before doing Line advance 71. Whether Line cas 70 is successful or not, n.blocks[h] ≠ null after the n.blocks[h].CAS. □

We define the subblocks of a block recursively.

ubblock *Definition 4.3 (Subblock).* A block is a *direct subblock* of the ith block in node n if it is in

$$n\texttt{.left.blocks[}n\texttt{.blocks[}i-1\texttt{].end}_{\texttt{left}}\texttt{+1}\cdots n\texttt{.blocks[}i\texttt{].end}_{\texttt{left}}\texttt{]}$$

or in

$$n\texttt{.right.blocks[}n\texttt{.blocks[}i-1\texttt{].end}_{\texttt{right}}\texttt{+1}\cdots n\texttt{.blocks[}i\texttt{].end}_{\texttt{right}}\texttt{]}.$$

Block B is a *subblock* of block C if B is a direct subblock of C or a subblock of a direct subblock of C. We say block B has been *propagated* to node n if B is in n.blocks or is a subblock of a block in n.blocks.

The following lemma is used to prove that subblocks of two blocks in a node are disjoint.

rogress LEMMA 4.4. *If* n.blocks[b] ≠ null($b > 0$) *then* n.blocks[i].end$_{\text{left}}$ ≥ n.blocks[$i-1$].end$_{\text{left}}$ *and* n.blocks[i].end$_{\text{right}}$ ≥ n.blocks[$i-1$].end$_{\text{right}}$.

Proof. Consider the block B written into `n.blocks[`b`]` by CAS at Line 70. Block B is created by the `CreateBlock(`b`)` called at Line 67. Prior to this call to `CreateBlock(`b`)`, `n.head` $= b$ at Line 60, so `n.blocks[`$b-1$`]` is already a non-null value B' by Invariant 1. Thus, the `CreateBlock(`$b-1$`)` that created B' has terminated before the invocation of `CreateBlock(`b`)` that created B. The value written into B`.end`$_{\texttt{left}}$ at Line 83 of `CreateBlock(`b`)` was one less than the value of n`.left.head` read at Line 83 of `CreateBlock(`b`)`. Similarly, the value in `n.blocks[`$b-1$`].end`$_{\texttt{left}}$ was one less than the value read from `n.left.head` during the call to `CreateBlock(`$b-1$`)`. By Observation 1, n`.left.head` is non-decreasing, so B'`.end`$_{\texttt{left}} \le B$`.end`$_{\texttt{left}}$. The proof for `end`$_{\texttt{right}}$ is similar. □

istinct

Lemma 4.5. *The sets of subblocks of any two blocks in a node are disjoint.*

Proof. We are going to prove the lemma by contradiction. Consider the lowest node n in the tree that violates the claim. Then, subblocks of n`.blocks[`i`]` and n`.blocks[`j`]` overlap for some $i < j$. Since n is the lowest node in the tree violating the claim, direct subblocks of blocks of n`.blocks[`i`]` and n`.blocks[`j`]` have to overlap. Without loss of generality, assume left child subblocks of n`.blocks[`i`]` overlap with the left child subblocks of n`.blocks[`j`]`. By Lemma 4.4 we have n`.blocks[`i`].end`$_{\texttt{left}} \le n$`.blocks[`$j-1$`].end`$_{\texttt{left}}$, so the range $[n$`.blocks[`$i-1$`].end`$_{\texttt{left}} + 1 \cdots n$`.blocks[`$i$`].end`$_{\texttt{left}}]$ cannot have overlap with the range $[n$`.blocks[`$j-1$`].end`$_{\texttt{left}} + 1 \cdots n$`.blocks[`$j$`].end`$_{\texttt{left}}]$. Therefore, direct subblocks of n`.blocks[`i`]` and n`.blocks[`j`]` cannot overlap, which is in contradiction with the assumption. □

Definition 4.6 (Superblock). Block B is *superblock* of block C if C is a direct subblock of B.

Corollary 4.7. *Every block has at most one superblock.*

Proof. A block having more than one superblock contradicts Lemma 4.5. □

Now we can define the operations of a block using the definition of subblocks.

ef::ops

Definition 4.8 (Operations of a block). A block B in a leaf represents an `Enqueue` if B`.element` $\neq$ `null`. Otherwise, if B`.element` $=$ `null`, B represents a `Dequeue`. The set of operations of block B is the union of the operations in leaf subblocks of B. We denote the set of operations of block B by $ops(B)$ and the union of operations of a set of blocks $\mathcal{B}$ by $ops(\mathcal{B})$. We also say B contains op if $op \in ops(B)$.

The next lemma proves that each operation appears at most once in the blocks of a node.

licates

Lemma 4.9. *For any node n, if op is in n*`.blocks[`i`]` *then there is no $j \neq i$ such that op is in* n`.blocks[`j`]`.

Proof. We prove this claim by contradiction using Lemma 4.5. Assume op is in the subblocks of both n`.blocks[`i`]` and n`.blocks[`j`]`. From Lemma 4.5 we know that the subblocks of these blocks are different, so there are two leaf blocks containing op. Since each process puts each operation in only one block of its leaf, op cannot be in two leaf blocks. This is a contradiction. □

Definition 4.10. n`.blocks[`i`]` is *established* if n`.head` $> i$. An operation is *established* in node n if it is in an established block of n. EST_n^t is the set of established operations in node n at time t.

Now we want to say that blocks of a node grow over time.

ksOrder

Observation 2. *If time $t <$ time t' (t is before t'), then $ops(n.blocks)$ at time t is a subset of $ops(n.blocks)$ at time t'.*

Proof. Blocks are only appended (not modified) with `CAS` to n`.blocks[`n`.head]`, so the set of the blocks of a node after the `CAS` contains the set of the blocks before the `CAS`. □

4.2 Ordering Operations

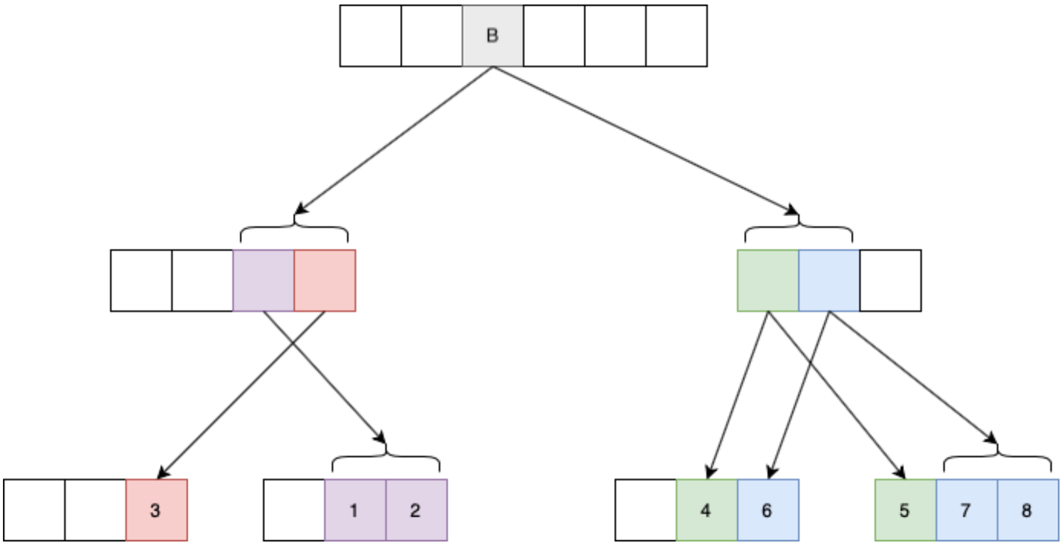


Fig. 6. Order of operations in the block B. Operations in the leaves are ordered in the numerical order shown in the drawing.

Now we define the ordering of operations stored in each node. In the non-root nodes, we only need to order operations of a type among themselves (that is, we order the Enqueues in the node and order the Dequeues in the node separately). Processes are numbered from 1 to p, and leaves of the tree are assigned from left to right. We will show in Lemma oneOpinBlock 4.23 that there is at most one operation from each process in a given block.

rdering *Definition 4.11 (Ordering of operations inside the nodes).*

- $E(n, b)$ is the sequence of enqueue operations in $ops(n.\texttt{blocks}[b])$ defined recursively as follows. $E(leaf, b)$ is the single enqueue operation in $ops(leaf.\texttt{blocks}[b])$ or an empty sequence if $leaf.\texttt{blocks}[b]$ represents a dequeue operation. If n is an internal node, then

$$\begin{aligned}E(n,b) =& E(n.\texttt{left}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{left}}+1)\cdots E(n.\texttt{left}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{left}})\cdot\\ & E(n.\texttt{right}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{right}}+1)\cdots E(n.\texttt{right}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{right}}).\end{aligned}$$

- $E_i(n, b)$ is the ith enqueue in $E(n, b)$.
- The order of the enqueue operations in the node n is $E(n) = E(n, 1) \cdot E(n, 2) \cdot E(n, 3) \cdots$
- $E_i(n)$ is the ith enqueue in $E(n)$.
- $D(n, b)$ is the sequence of dequeue operations in $ops(n.\texttt{blocks}[b])$ defined recursively as follows. $D(leaf, b)$ is the single dequeue operation in $ops(leaf.\texttt{blocks}[b])$ or an empty sequence if $leaf.\texttt{blocks}[b]$ represents an enqueue operation. If n is an internal node, then

$$\begin{aligned}D(n,b) =& D(n.\texttt{left}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{left}}+1)\cdots D(n.\texttt{left}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{left}})\cdot\\ & D(n.\texttt{right}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{right}}+1)\cdots D(n.\texttt{right}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{right}}).\end{aligned}$$

- $D_i(n, b)$ is the ith enqueue in $D(n, b)$.
- The order of the dequeue operations in the node n is $D(n) = D(n, 1) \cdot D(n, 2) \cdot D(n, 3)\ldots$
- $D_i(n)$ is the ith dequeue in $D(n)$.

The linearization ordering is given by the order in which operations appear in the blocks in the root.

ef::lin

Definition 4.12 (Linearization).

$$L = E(root,1)\cdot D(root,1)\cdot E(root,2)\cdot D(root,2)\cdot E(root,3)\cdot D(root,3)\cdots$$

The following observation follows from the Definition of $\mathsf{num_x}$ on page 22.

umToNum

Observation 3. *For any node n and indices $i<j$ of* `blocks` *in n, we have*

$$n.\texttt{blocks[}j\texttt{].sum}_\mathsf{x} - n.\texttt{blocks[}i\texttt{].sum}_\mathsf{x} = \sum_{k=i+1}^{j} n.\texttt{blocks[}k\texttt{].num}_\mathsf{x}$$

where $\mathsf{x} \in \{\mathsf{enq}, \mathsf{deq}, \texttt{enq-left}, \texttt{enq-right}, \texttt{deq-left}, \texttt{deq-right}\}$.

The next claim is also valid if we replace enq with deq and E with D.

m::numX

Lemma 4.13. *Let B and B' be n.*`blocks[`b`]` *and n.*`blocks[`$b-1$`]`*, respectively. If n is an internal node, then*

(1) $B.\mathsf{num}_{\texttt{enq-left}} = \left|E(n.\texttt{left}, B'.\mathsf{end}_{\texttt{left}}+1)\cdots E(n.\texttt{left}, B.\mathsf{end}_{\texttt{left}})\right|$.

(2) $B.\mathsf{num}_{\texttt{enq-right}} = \left|E(n.\texttt{right}, B'.\mathsf{end}_{\texttt{right}}+1)\cdots E(n.\texttt{right}, B.\mathsf{end}_{\texttt{right}})\right|$.

And for every node n, we have

(3) $B.\mathsf{num}_{\mathsf{enq}} = \left|E(n,b)\right|$.

Proof. We prove the claim by induction on the height of node n. For the base case when n is a leaf, statement (3) is trivial, and (1) and (2) are vacuously true. Supposing the claim is true for n's children, we prove the claim for n.

$$\begin{aligned}
B.\mathsf{num}_{\texttt{enq-left}} &= B.\mathsf{sum}_{\texttt{enq-left}} - B'.\mathsf{sum}_{\texttt{enq-left}}\\
&= B'.\mathsf{sum}_{\texttt{enq-left}} + n.\texttt{left.blocks[}B.\mathsf{end}_{\texttt{left}}\texttt{].sum}_{\mathsf{enq}}\\
&\quad - n.\texttt{left.blocks[}B'.\mathsf{end}_{\texttt{left}}\texttt{].sum}_{\mathsf{enq}} - B'.\mathsf{sum}_{\texttt{enq-left}}\\
&= n.\texttt{left.blocks[}B.\mathsf{end}_{\texttt{left}}\texttt{].sum}_{\mathsf{enq}} - n.\texttt{left.blocks[}B'.\mathsf{end}_{\texttt{left}}\texttt{].sum}_{\mathsf{enq}}\\
&= \sum_{i=B'.\mathsf{end}_{\texttt{left}}+1}^{B.\mathsf{end}_{\texttt{left}}} n.\texttt{left.blocks[}i\texttt{].num}_{\mathsf{enq}}\\
&= \left|E(n.\texttt{left}, B'.\mathsf{end}_{\texttt{left}}+1)\cdots E(n.\texttt{left}, B.\mathsf{end}_{\texttt{left}})\right|
\end{aligned}$$

The first line follows from the Definition of $\mathsf{num}_{\mathsf{enq}}$. The second line is similar to the way $\mathsf{sum}_{\texttt{enq-left}}$ is computed in the `CreateBlock` routine. Observation 3 implies the third line and the last line holds because of the induction hypothesis (3). (2) is similar to (1). Now we prove (3) starting from the definition of $E(n,b)$.

$$\begin{aligned}
E(n,b) = &E(n.\texttt{left}, n.\texttt{blocks[}b-1\texttt{].end}_{\texttt{left}}+1)\cdots E(n.\texttt{left}, n.\texttt{blocks[}b\texttt{].end}_{\texttt{left}})\cdot\\
&E(n.\texttt{right}, n.\texttt{blocks[}b-1\texttt{].end}_{\texttt{right}}+1)\cdots E(n.\texttt{right}, n.\texttt{blocks[}b\texttt{].end}_{\texttt{right}}).
\end{aligned}$$

By (1) and (2) we have $\left|E(n,b)\right| = B.\mathsf{num}_{\texttt{enq-left}} + B.\mathsf{num}_{\texttt{enq-right}} = B.\mathsf{num}_{\mathsf{enq}}$. □

The next claim is also true if we replace enq with deq and E with D.

m::sumX

Corollary 4.14. *Let B be n.*`blocks[`b`]`.

(1) *If* n *is an internal node then* $B.\texttt{sum}_{\texttt{enq-left}} = \left|E(n.\texttt{left}, 1) \cdots E(n.\texttt{left}, B.\texttt{end}_{\texttt{left}})\right|$.

(2) *If* n *is an internal node then* $B.\texttt{sum}_{\texttt{enq-right}} = \left|E(n.\texttt{right}, 1) \cdots E(n.\texttt{right}, B.\texttt{end}_{\texttt{right}})\right|$.

(3) $B.\texttt{sum}_{\texttt{enq}} = \left|E(n, 1) \cdot E(n, 2) \cdots E(n, b)\right|$.

Proof. Result (1) can be proved using the previous lemma.

$$
\begin{aligned}
B.\texttt{sum}_{\texttt{enq-left}} &= n.\texttt{blocks[1]}.\texttt{num}_{\texttt{enq-left}} + \cdots + n.\texttt{blocks[}b\texttt{]}.\texttt{num}_{\texttt{enq-left}} \\
&= \left|E(n.\texttt{left}, 1) \cdots E(n.\texttt{left}, n.\texttt{blocks[1]}.\texttt{end}_{\texttt{left}})\right| + \\
&\quad \vdots \\
&\quad + \left|E(n.\texttt{left}, n.\texttt{blocks[}b-1\texttt{]}.\texttt{end}_{\texttt{left}}) \cdots E(n.\texttt{left}, n.\texttt{blocks[}b\texttt{]}.\texttt{end}_{\texttt{left}})\right| \\
&= \left|E(n.\texttt{left}, 1) \cdots E(n.\texttt{left}, B.\texttt{end}_{\texttt{left}})\right|
\end{aligned}
$$

We can prove (2) and (3) the same as (1). □

4.3 Propagating Operations to the Root

This section explains why two Refreshes are enough to propagate a node's operations to its parent.

Definition 4.15. Let t^{op} be the time *op* is invoked, ${}^{op}t$ be the time *op* terminates, t_l^{op} be the time immediately before running Line l of operation *op* and ${}^{op}_{l}t$ be the time immediately after running Line l of operation *op*. We sometimes suppress *op* and write t_l or ${}_lt$ if *op* is clear from the context. In the text, v_l is the value of variable v immediately after line l for the process we are talking about and v_t is the value of variable v at time t.

Definition 4.16 (Successful Refresh). An instance of Refresh is *successful* if its CAS in Line $\overset{\texttt{cas}}{70}$ returns true.

In the next two results, we show that for every successful Refresh, all the operations established in the children before the Refresh are in the parent after the Refresh's successful CAS at Line $\overset{\texttt{cas}}{70}$.

Refresh

Lemma 4.17. *If* R *is a successful instance of* $n.$Refresh*, then we have* $EST^{t^R}_{n.\texttt{left}} \cup EST^{t^R}_{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{\overset{\texttt{cas}}{70}})$.

Proof. We show
$$
\begin{aligned}
EST^{t^R}_{n.\texttt{left}} &= ops(n.\texttt{left.blocks[0..n.left.head}_{t^R} - 1\texttt{]}) \\
&\subseteq ops(n.\texttt{blocks}_{\overset{\texttt{cas}}{70}}) = ops(n.\texttt{blocks[0..}n.\texttt{head}_{\overset{\texttt{cas}}{70}}\texttt{]}).
\end{aligned}
$$

In every node, blocks[0] is an empty block without any operations. Line $\overset{\texttt{cas}}{70}$ stores a block new in n that has $\texttt{end}_{\texttt{left}} = n.\texttt{left.head}_{\overset{\texttt{lastLine}}{83}} - 1$. Therefore, by Definition $\overset{\texttt{def::subblock}}{4.3}$, after the successful CAS in Line $\overset{\texttt{cas}}{70}$ we know all blocks in $n.\texttt{left.blocks[}1 \cdots n.\texttt{left.head}_{\overset{\texttt{lastLine}}{83}} - 1\texttt{]}$ are subblocks of $n.\texttt{blocks[}1 \cdots n.\texttt{head}_{\overset{\texttt{readHead}}{60}}\texttt{]}$. Because of Observation $\overset{\texttt{nonDecreasingHead}}{1}$ we have $n.\texttt{left.head}_{t^R} - 1 \le n.\texttt{left.head}_{\overset{\texttt{lastLine}}{83}} - 1$ and $n.\texttt{head}_{\overset{\texttt{readHead}}{60}} \le n.\texttt{head}_{\overset{\texttt{cas}}{70}}$. From Observation $\overset{\texttt{lem::blocksOrder}}{2}$ the claim follows. The proof for the right child is the same. □

Refresh

Corollary 4.18. *If* R *is a successful instance of* $n.$Refresh *that terminates, then we have*

$$EST^{t^R}_{n.\texttt{left}} \cup EST^{t^R}_{n.\texttt{right}} \subseteq EST^{{}^Rt}_{n}.$$

Proof. The left-hand side is the same as Lemma 4.17, so it is sufficient to show when R terminates the established blocks in n are a superset of $n.\texttt{blocks}_{70}$. Line 70 writes the block new in $n.\texttt{blocks}[h]$ where h is value of $n.\texttt{head}$ read at Line 60. Because of Lemma 4.2 we are sure that $n.\texttt{head} > h$ when R terminates. So the block new appended to n at Line 70 is established at ${}^{R}t$. □

In the next lemma, we show that if two consecutive instances of `Refresh` by the same process on node n fail, then the blocks established in the children of n before the first `Refresh` are guaranteed to be in n after the second `Refresh`.

Refresh Lemma 4.19. *Consider two consecutive terminating instances R_1, R_2 of* `Refresh` *by a process on an internal node n. If neither R_1 nor R_2 is a successful* `Refresh`*, then we have $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq EST^{R_2 t}_{n}$.*

Proof. Let R_1 read i from $n.\texttt{head}$ at Line 60. By Lemma 4.2, R_1 and R_2 cannot both read the same value i. By Observation 1, R_2 reads a larger value of $n.\texttt{head}$ than R_1.

Consider the case where R_1 reads i and R_2 reads $i+1$ from Line 60. As R_2's CAS in Line 70 returns `false`, there is another successful instance R'_2 of $n.\texttt{Refresh}$ that has done a CAS successfully into $n.\texttt{blocks}[i+1]$ before R_2 tries to CAS. R'_2 creates its block new after reading the value $i+1$ from $n.\texttt{head}$ (Line 60) and R_1 reads the value i from $n.\texttt{head}$. By Observation 1 we have ${}^{R_1}t < t^{R_1}_{60} < t^{R'_2}_{60}$ (see Figure 7). By Lemma 4.17 we have $EST^{n.\texttt{left}}_{t^{R'_2}_{60}} \cup EST^{n.\texttt{right}}_{t^{R'_2}_{60}} \subseteq ops(n.\texttt{blocks}_{t^{R'_2}_{70}})$. Also by Lemma 4.2 on R_2, the value of $n.\texttt{head}$ is more than $i+1$ after R_2 terminates, so the block appended by R'_2 into $n.\texttt{blocks}[i]$ is established by the time R_2 terminates. To summarize, ${}^{R_1}t$ is before R'_2's read of $n.\texttt{head}$ ($t^{R'_2}_{60}$), so we have $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{t^{R'_2}_{70}})$. R'_2's successful CAS ($t^{R'_2}_{70}$) is before R_2's termination (t^{R_2}), so by Lemma 4.2 $n.\texttt{head}$ has been incremented when R_2 terminates and the block R'_2 put into n is established by then. So we have $ops(n.\texttt{blocks}_{t^{R'_2}_{70}}) \subseteq EST^{R_2 t}_{n}$.

If R_2 reads some value greater than $i+1$ in Line 60 it means $n.\texttt{head}$ has been incremented at least two times since $^{R_1}_{60}t$. By Invariant 1, when $n.\texttt{head}$ is incremented from $i+1$ to $i+2$, $n.\texttt{blocks}[i+1]$ is non-null. Let R_3 be the `Refresh` on n that has put the block in $n.\texttt{blocks}[i+1]$. R_3 read $n.\texttt{head} = i+1$ at Line 60 and has put its block in $n.\texttt{blocks}[i+1]$ before R_2's read of n.head at Line 60. So we have $t^{R_1} < {}^{R_3}_{60}t < {}^{R_3}_{70}t < {}^{R_2}_{60}t$. From Observation 2 on the operations before and after R_3's CAS and Lemmas 4.17 and 4.2 on R_3 the claim holds. □

ateStep Corollary 4.20. $EST^{t}_{n.\texttt{left}_{52}} \cup EST^{t}_{n.\texttt{right}_{52}} \subseteq EST^{t_{53}}_{n}$

Proof. If the first `Refresh` in line 52 returns `true`, then by Corollary 4.18 the claim holds. If the first `Refresh` failed and the second `Refresh` succeeded, the claim still holds by Corollary 4.18. Otherwise, both failed, and the claim is implied by Lemma 4.19. □

Now we show that after `Append`(B) on a leaf finishes, the operation contained in B will be established in `root`.

:append Corollary 4.21. *For $A = l.$*`Append`*(B) we have $ops(b) \subseteq EST^{t^{A}}_{n}$ for each node n in the path from l to* `root`.

Proof. A adds B to the assigned leaf of the process, establishes it at Line 48 and then calls `Propagate` on the parent of the leaf where it appended B. For every node n, $n.$`Propagate` appends B to n, establishes it in n by Corollary 4.20 and then calls $n.\texttt{parent}.\texttt{Propagate}$ until n is `root`. □

tlyOnce Corollary 4.22. *After $l.$*`Append`*(B) finishes, B is a subblock of exactly one block in each node along the path from l to the* `root`.

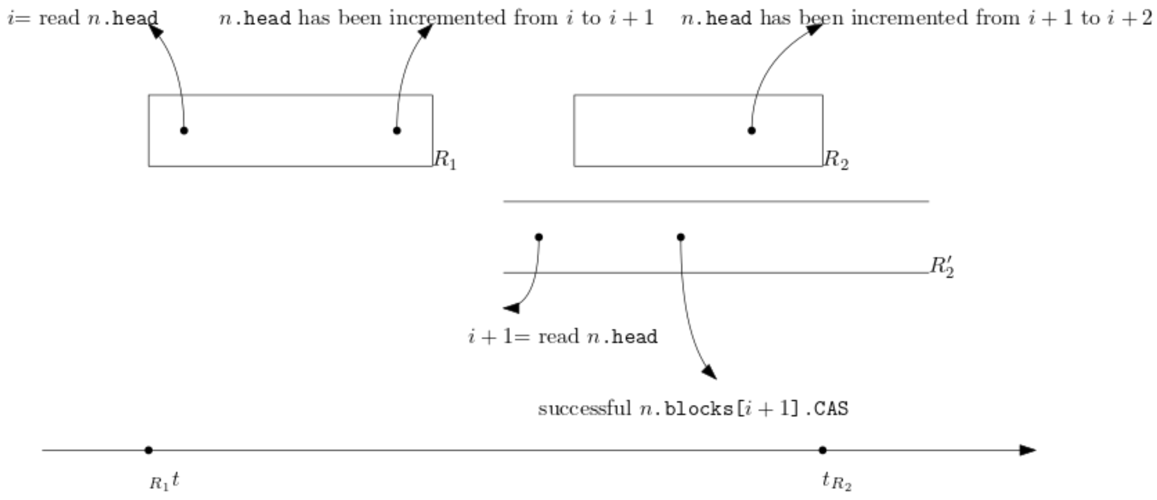

i= read n.head
n.head has been incremented from i to $i+1$
n.head has been incremented from $i+1$ to $i+2$
R_1
R_2
R_2'
$i+1$= read n.head
successful n.blocks[$i+1$].CAS
$_{R_1}t$
t_{R_2}

Fig. 7. $_{R_1}t < t^{R_1}_{\text{ReadHead}\,60} <$ incrementing n.head from i to $i+1$ $< t^{R_2'}_{\text{ReadHead}\,60} < t^{R_2'}_{\text{CAS}\,70} <$ incrementing n.head from $i+1$ to $i+2$ $< t_{R_2}$

Proof. By the previous corollary and Lemma 4.9 there is exactly one block in each node containing B. □

4.4 Correctness of GetEnqueue

First, we prove some claims about the size and operations of a block. These lemmas will be used later for the correctness and analysis of `GetEnqueue()`.

Lemma 4.23. *Each block contains at most one operation of each process.*

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process P in block B in node n. Without loss of generality op_1 is invoked earlier than op_2. Process P cannot invoke more than one operation concurrently, so op_1 has to be finished before op_2 begins. By Corollary 4.22, before op_2 calls Append, op_1 exists in every node of the tree on the path from P's leaf to the root. Since b contains op_2, it must be created after op_2 is invoked. The fact that op_2.Append is invoked after op_1.Append terminated means that there is some block B' in n before B that contains op_1. The existence of op_1 in B and B' contradicts Lemma 4.9. □

Lemma 4.24. *Each block contains at most c operations, where c is the maximum number of concurrent operations at any time in the whole execution ($c \leq p$).*

Proof. There is a time that all the operations in a block are concurrent, because otherwise if there is an operation in a block that has ended before another operation in that block starts, then by Corollary 4.22 these two operations couldn't be in the same block. From the definition of c we know at any time in the execution there cannot be more than c concurrent operations, and from the previous lemma we know a process has at most one operation in a block, so there cannot be a block with more than c operations. □

Lemma 4.25. *Each block has at most c direct subblocks, where c is the maximum number of concurrent operations at any time in the whole execution ($c \leq p$).*

Proof. From Definition 4.8 we know the operations in a block are the union of the operations in the direct subblocks of the block. We can see that each block appended to an internal node contains at least one operation due to the test on Line 68. Also, blocks in the leaves contain only one `Enqueue` or `Dequeue` operation. By Lemma 4.24 each block in an internal node contains at most c operations and each one of its direct subblocks has at least one operation, so by pigeonhole principle the number of direct subblocks in a block is at most c. □

`DoublingSearch(`e`, `end`)` returns a pair <b, i> such that the ith `Enqueue` in the bth block of the root is the eth `Enqueue` in the sequence stored in the root.

dsearch Lemma 4.26 (`DoublingSearch` correctness). *If* $1 \leq e \leq$ `root.blocks[`end`].sum`$_{\texttt{enq}}$, *then* `DoublingSearch(`e`, `end`)` *returns* <b, i> *such that* $E_i(root, b) = E_e(root)$.

Proof. From Lines 86 and 87 we know the `sum`$_{\texttt{enq-left}}$, and `sum`$_{\texttt{enq-right}}$ fields of `blocks` in each node are sorted in non-decreasing order. Since `sum`$_{\texttt{enq}}$ = `sum`$_{\texttt{enq-left}}$ + `sum`$_{\texttt{enq-right}}$, the `sum`$_{\texttt{enq}}$ values of `root.blocks[0`$\cdots$$end$`]` are also non-decreasing. By Corollary 4.14 we know that the `sum`$_{\texttt{enq}}$ field in a block is the sum of the number of `Enqueue` operations in that block and the all blocks before that block in the node. Furthermore, since `root.blocks[0].sum`$_{\texttt{enq}}$ $= 0$ and `root.blocks[`end`].sum`$_{\texttt{enq}}$ $\geq e$, there is a b such that `root.blocks[`$b-1$`].sum`$_{\texttt{enq}}$ $< e$ and $e \leq$ `root.blocks[`b`].sum`$_{\texttt{enq}}$. Block `root.blocks[`b`]` contains $E_i(root, b)$. Lines 38–41 doubles the search range in Line 40 and will eventually reach `start` such that `root.blocks[start].sum`$_{\texttt{enq}}$ $\leq e \leq$ `root.blocks[end].sum`$_{\texttt{enq}}$. Then, in Line 42, the binary search finds the b such that `root.blocks[`$b-1$`].sum`$_{\texttt{enq}}$$< e \leq$`root.blocks[`$b$`].sum`$_{\texttt{enq}}$. By Corollary 4.14, `root.blocks[`b`]` is the block that contains $E_e(root)$. Finally, i is computed using the definition of `sum`$_{\texttt{enq}}$ and Corollary 4.14. □

get Lemma 4.27 (`GetEnqueue` correctness). *If* $1 \leq i \leq n$`.blocks[`b`].num`$_{\texttt{enq}}$ *then* n`.GetEnqueue(`b, i`)` *returns* $E_i(n, b)$`.element`.

Proof. We will prove this lemma by induction on the node n's height. For the base case, suppose n is a leaf. Leaf blocks each contain exactly one operation, n`.blocks[`b`].sum`$_{\texttt{enq}}$ ≤ 1, which means only n`.GetEnqueue(b,1)` can be called when n is a leaf and n`.blocks[`b`]` must contain an `Enqueue` operation. Line 97 of n`.GetEnqueue(`b`, 1)` returns the `element` of the `Enqueue` operation stored in the bth block of leaf n, as required.

For the induction step, we prove if n`.dir.GetEnqueue(`b'`, `i`)` returns $E_i(n$`.dir`$, b')$ then n`.GetEnqueue(`b`, `i`)` returns $E_i(n, b)$. From Definition 4.11 of $E(n, b)$, we know that operations from the left subblocks come before the operations from the right subblocks in a block (see Figure 8). By Lemma 4.13, the `num`$_{\texttt{enq-left}}$ field in n`.blocks[`b`]` is the number of `Enqueue` operations from the blocks' subblocks in the left child of n. So the ith `Enqueue` operation in n`.blocks[`b`]` is propagated from the right child if and only if i is greater than n`.blocks[`b`].num`$_{\texttt{enq-left}}$. Line 98 decides whether the ith enqueue in the bth block of internal node n is in the left child or right child subblocks of n`.blocks[`b`]`. By Definitions 4.3 and 4.8, to find an operation in the subblocks of n`.blocks[`b`]` we need to search in the range

n`.left.blocks[`n`.blocks[`b`-1].end`$_{\texttt{left}}$`+1..`n`.blocks[`b`].end`$_{\texttt{left}}$`]` or

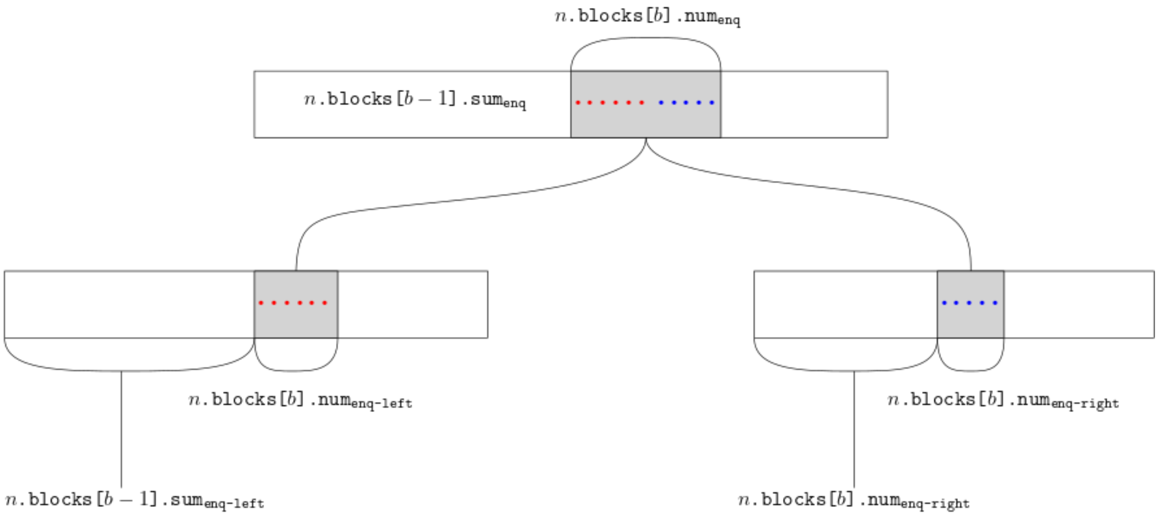
n`.right.blocks[`n`.blocks[`b`-1].end`$_{\texttt{right}}$`+1..`n`.blocks[`b`].end`$_{\texttt{right}}$`]`.

First, we consider the case where the `Enqueue` we are looking for is in the left child. There are $eb = n$`.blocks[`$b-1$`].sum`$_{\texttt{enq-left}}$ `Enqueues` in the blocks of n`.left` before the left subblocks of n`.blocks[`b`]`, so $E_i(n, b)$ is $E_{i+eb}(n$`.left`$)$ which is $E_{i'}(n$`.left`$, b')$ for some b' and i'. We can compute b' and then search for the i'th `Enqueue` in n`.left.blocks[`b'`]`, where i' is $i + eb -$ n`.left.blocks[`$b'-1$`].sum`$_{\texttt{enq}}$. The parameters in Line 99 are for searching $E_{i+eb}(n$`.left`$)$ in

n.`left.blocks` in the range of left subblocks of n.`blocks`[b], so this `BinarySearch` returns the index of the subblock containing $E_i(n, b)$.

Otherwise, the `Enqueue` we are looking for is in the right child. Because `Enqueues` from the left subblocks are ordered before the ones from the right subblocks, there are `n.blocks[b].num`$_{\texttt{enq-left}}$ enqueues ahead of $E_i(n, b)$ from the left child. So we need to search for $i-n$.`blocks`[b]`.num`$_{\texttt{enq-left}}+$ n.`blocks`[$b-1$]`.sum`$_{\texttt{enq-right}}$ in the right child (Line rightChildGet 103). Other parameters for the right child are chosen similarly to the left child.

So, in both cases, the direct subblock containing $E_i(n, b)$ is computed in Line leftChildGet 99 or rightChildGet 103. *subblockIndex* is the index of the block in n.`dir` containing $E_i(n, b)$. Finally, n.`child.GetEnqueue(` *subblockIndex*, i) is invoked and it returns $E_i(n, b)$.`element` by the hypothesis of the induction. □


n.blocks[b].num$_{\text{enq}}$
n.blocks[$b-1$].sum$_{\text{enq}}$
n.blocks[b].num$_{\text{enq-left}}$
n.blocks[b].num$_{\text{enq-right}}$
n.blocks[$b-1$].sum$_{\text{enq-left}}$
n.blocks[b].num$_{\text{enq-right}}$

figGet Fig. 8. The number and order of the `Enqueue` operations propagated from the left and the right child to n.`blocks`[b]. Both n.`blocks`[b] and its subblocks are shown in grey. `Enqueue` operations from the left child (colored red), are ordered before the `Enqueue` operations from the right child (colored blue).

4.5 Correctness of IndexDequeue

The next few results show that the `super` field of a block is accurate within one of the actual index of the block's superblock in the parent node. Then we explain how it is used to compute the rank of a given `Dequeue` in the root.

Refresh *Definition 4.28.* If a `Refresh` instance R_1 does its `CAS` at Line cas 70 earlier than `Refresh` instance R_2 we say R_1 has *happened before* R_2.

IncHead Observation 4. *After* n.`blocks`[i]`.CAS(null,` B) *succeeds,* n.`head` *cannot increase from* i *to* $i+1$ *until* B.`super` *is set.*

Proof. From Observation nonDecreasingHead 1 we know that n.`head` changes only by the increment on Line incrementHead 77. Before an instance of `Advance` increments `n.head` on Line incrementHead 77, Line setSuper1 76 ensures that `n.blocks[head].super` was set at Line setSuper1 76. □

Corollary 4.29. *If* n.`blocks`[i]`.super` *is* `null`, *then* n.`head` $\leq i$ *and* n.`blocks`[$i+1$] *is* `null`.

Proof. By Invariant 1 and Observation 4. □

Now let us consider how the `Refresh`es that took place on the parent of node n after block B was stored in n will help to set B.`super` and propagate B to the parent.

ockHead

Observation 5. *If the block created by an instance R_p of* n.`parent.Refresh` *contains block* $B = n$.`blocks[`b`]` *then R_p reads a value greater than b from* n.`head` *in Line 83.*

Lemma 4.30. *If* $B = n$.`blocks[`b`]` *is a direct subblock of* n.`parent.blocks[`*superblock*`]` *then* B.`super` $\leq$ *superblock.*

Proof. Let R_p be the instance of n.`parent.Refresh` that does a successful `CAS`(Line 70) and puts the superblock of B which is n.`parent.blocks[`*superblock*`]` into n.`parent`. By Observation 5 if R_p propagates B it has to read a greater value than b from n.`head`, which means n.`head` was incremented from b to $b+1$ in Line 77. By Observation 4 B.`super` was already set in Line 76. The value written in B.`super`, was read in Line 75 before the `CAS` that sets B.`super` in Line 76. From Observation 1 we know n.`parent.head` is non-decreasing so B.`super` $\leq$ *superblock*, since n.`parent.head` is still equal to *superblock* when R_p executes its `CAS` at Line 70 by Lemma 4.4. The reader may wonder when the case b.`super` $=$ *superblock* happens. This can happen when n.`parent.blocks[`B.`super]` = `null` when B.`super` is written and R_p puts its created block into n.`parent.blocks[`B.`super]` afterwards. □

shSuper

Lemma 4.31. *Let R_n be a* `Refresh` *that puts B in* n.`blocks[`b`]` *at Line 70. Then, the block created by one of the next two successful* n.`parent.Refresh`*es according to Definition 4.28 contains B and* B.`super` *is set when the second successful* n.`parent.Refresh` *reaches Line 67.*

Proof. Let R_{p1} and R_{p2} be the next two successful n.`parent.Refresh`es after R_n. To derive a contradiction assume B was neither propagated to n.`parent` by R_{p1} nor by R_{p2}.

Since R_{p2}'s created block does not contain B, by Observation 5 the value R_{p2} reads from n.`head` in Line 83 is at most b. From Observation 1 the value R_{p2} reads in Line 62 is also at most b.

R_n puts B into n.`blocks[`b`]` so R_n reads the value b from n.head. Since R_{p2}'s `CAS` into n.`parent.blocks` is successful there should be a `Refresh` instance R'_p on n.`parent` that increments n.`parent.head` (Line 77) after R_{p1}'s Line 70 and before R_{p2}'s Line 60. We assumed $t^{R_n}_{70} < t^{R_{p1}}_{70} < t^{R_{p2}}_{70}$ by Definition 4.28. Finally, Line 62 is after Line 60 and R_{p2}'s Line 60 is after R'_p's Line 77, which is after R_n's n.`blocks.CAS`.

$$t^{R_n}_{70} < \left.\begin{array}{l} t^{R_{p1}}_{70} < t^{R_{p2}}_{70} \\ t^{R'_p}_{77} < t^{R_{p2}}_{60} \\ t^{R_{p2}}_{60} < t^{R_{p2}}_{62} \end{array}\right\} \Longrightarrow t^{R_n}_{70} < t^{R_{p2}}_{62}$$

So R_{p2} reads a value greater than or equal to b for n.head by Observation 1.

Therefore R_{p2} reads n.`head` $= b$. R_{p2} calls n.`Advance` at Line 64, which ensures n.`head` is incremented from b. So the value R_{p2} reads in Line 83 of `CreateBlock` is greater than b and R_{p2}'s created block contains B. This is in contradiction with our hypothesis.

Furthermore, if B.`super` was not set earlier, it is set by R_{p2}'s call to n.`Advance` invoked from Line 64. □

elation

Corollary 4.32. *If* $B = n$.`blocks[`b`]` *is propagated to* n.`parent`*, then* B.`super` *is equal to or one less than the index of the superblock of B.*

Proof. Let R_n be the n.`Refresh` that put B in n.`blocks` and let R_{p1} be the first successful n.`parent.Refresh` after R_n and R_{p2} be the second next successful n.`parent.Refresh`. Before B

can be propagated to n's parent, n.head must be greater than b, so by Observation 4 B.super is set. From Lemma 4.31 we know that B is propagated by the second next successful Refresh's CAS on n.parent.blocks. To summarize, we have n.parent.head$_{c_{p_2}^{t}}$ = n.parent.head$_{c_{p_1}^{t}}$ + 1 and n.parent.head$_{c_{p_1}^{t}}$ ≤ n.parent.head$_{c_{n}^{t}}$ from Definition 4.28 and Observation 1. The value that is set in B.super is read from n.parent.head after c_n^t. So B.super is equal to or one less than the index of the superblock of B. □

We prove IndexDequeue's correctness using Corollary 4.32 on each step of the IndexDequeue.

Lemma 4.33 (IndexDequeue correctness). *If* $1 \le i \le n$.blocks[b].num$_{\text{deq}}$ *then* n.IndexDequeue(b, i) *returns* $< x, y >$ *such that* $D_i(n, b) = D_y(\text{root}, x)$.

Proof. We will prove this by induction on the distance of n from the root. The base case where n is root is trivial (see Line 109). For the non-root nodes n.IndexDequeue(b, i) computes *superblockIndex*, the index of the superblock of the bth block in n, in Line 112 by Corollary 4.32. After that, the position of $D_i(n, b)$ in $D(n.\text{parent}, \textit{superblockIndex})$ is computed in Lines 113–118. By Definition 4.11, Dequeues in a block are ordered based on the order of its subblocks from left to right. If $D_i(n, b)$ was propagated from the left child, the number of dequeues in the left subblocks of n.parent.blocks[*superblockIndex*] before n.blocks[b] is considered in Line 114 (see Figure 9). Otherwise, if $D_i(n, b)$ was propagated from the right child, the number of dequeues in the subblocks from the left child is considered to be ahead of the computed index (Line 115) (see Figure 10). Finally, IndexDequeue is called on n.parent recursively, and it returns the correct response by the induction hypothesis. □

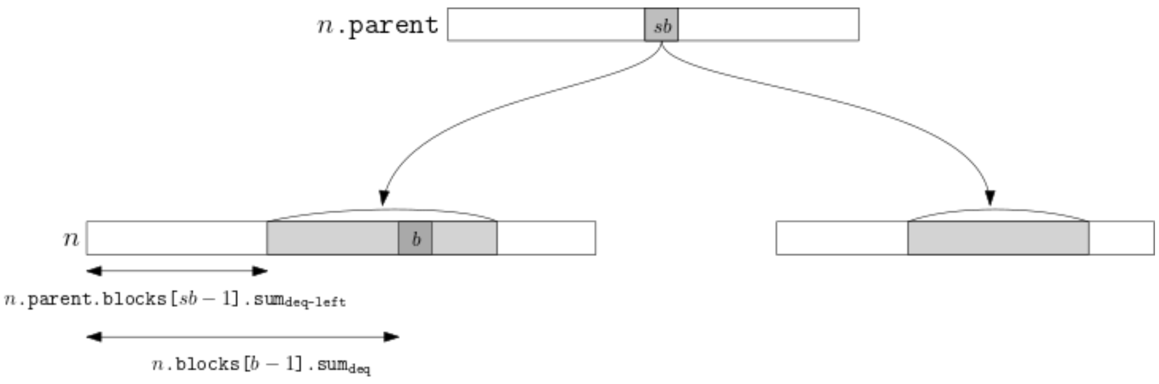


Fig. 9. The number of Dequeue operations before $D_i(n, b)$ shown in the case where n is a left child. The index of the superblock is shown with sb.

4.6 Linearizability

We now prove the two properties needed for linearizability.

Lemma 4.34. *L is a legal linearization ordering.*

Proof. We must show for any execution that every operation that terminates is in L exactly once. Also, if op_1 terminates before op_2 in starts in the execution, then op_1 is before op_2 in the linearization. The first claim is directly reasoned from Corollary 4.22. For the latter, if op_1 terminates

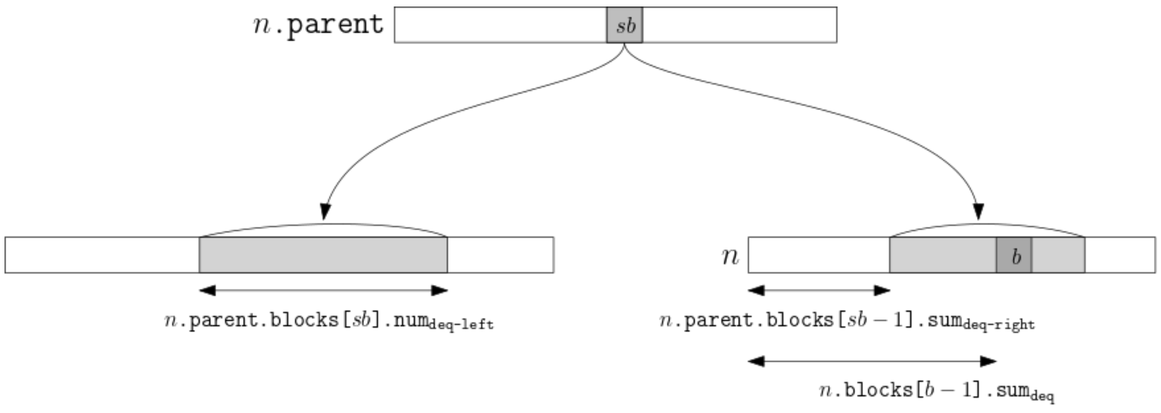

n.parent
sb
n
b
n.parent.blocks[sb].num$_{\text{deq-left}}$
n.parent.blocks[$sb-1$].sum$_{\text{deq-right}}$
n.blocks[$b-1$].sum$_{\text{deq}}$

Fig. 10. The number of Dequeue operations before $D_i(n, b)$ shown in the case where n is a right child. The index of the superblock is shown with sb.

eqRight

before op_2 starts, op_1.Append has terminated before op_2.Append started. From Corollary lem::append 4.21, op_1 is in root.blocks before op_2 starts to propagate. By definition of L, op_1 is linearized before op_2. □

Once some operations are aggregated in one block, they will get propagated up to the root together, and they can be linearized in any order among themselves. We have chosen to put Enqueues in a block before Dequeues (see Definition ordering 4.11).

NullDeq

Definition 4.35. If a Dequeue operation returns null it is called a *null* Dequeue, otherwise it is called *non-null* Dequeue.

Next, we define the responses that Dequeues should return, according to the linearization.

Definition 4.36. Assume the operations in root.blocks are applied sequentially on an empty queue in the order of L. $Resp(d) = e$.element if the element of Enqueue e is the response to Dequeue d. Otherwise if d is a null Dequeue then $Resp(d) =$ null.

In the next lemma, we show that the size field in each root block is computed correctly.

ectness

Lemma 4.37. root.blocks[b].size *is the size of the queue after the operations in* root.blocks[$0 \cdots b$] *are applied in the order of* L.

Proof. We prove the claim by induction on b. The base case when $b = 0$ is trivial since the queue is initially empty and root.blocks[0] contains an empty block with size field equal to 0. We are going to show the correctness when $b = i$ assuming correctness when $b = i - 1$. By Definition ordering 4.11 Enqueue operations come before Dequeue operations in a block in L. By Lemma lem::numX 4.13 num$_{\text{enq}}$ and num$_{\text{deq}}$ fields in a block show the number of Enqueue and Dequeue operations in it. If there are more than root.blocks[$i-1$].size + root.blocks[i].num$_{\text{enq}}$ dequeue operations in root.blocks[i] then the queue would become empty after root.blocks[i]. Otherwise, the size of the queue after the bth block in the root is root.blocks[$b-1$].size + root.blocks[b].num$_{\text{enq}}$ − root.blocks[b].num$_{\text{deq}}$. In both cases, this is the same as the assignment on Line computeLength 91. □

The next lemma is useful to compute the number of non-null dequeues.

erOfNND

Lemma 4.38. *If operations in the root are applied in the order of* L*, the number of non-null* Dequeues *in* root.blocks[$0 \cdots b$] *is* root.blocks[b].sum$_{\text{enq}}$ − root.blocks[b].size.

Proof. There are root.blocks[b].sum$_{\text{enq}}$ Enqueue operations in root.blocks[$0\cdots b$] by Corollary lem::sumX 4.14. The size of the queue after doing root.blocks[$0\cdots b$] in the order of L is the number of *enqueues* in root.blocks[$0\cdots b$] minus the number of *non-null* Dequeue*s* in root.blocks[$0\cdots b$]. By the correctness of the size field from Lemma sizeCorrectness 4.37 and sum$_{\text{enq}}$ field from Lemma lem::numX 4.13, the number of *non-null* Dequeue*s* is root.blocks[b].sum$_{\text{enq}}$ $-$ root.blocks[b].size. □

eqBlock

Corollary 4.39. *If operations in the root are applied in the order of L, the number of non-null dequeues in* root.blocks[b] *is* root.blocks[b].num$_{\text{enq}}$ $-$ root.blocks[b].size $+$ root.blocks[$b-$1].size.

lReturn

Lemma 4.40. *Resp*(D_i(root, b)) *is* null *iff* root.blocks[$b-$1].size + root.blocks[b].num$_{\text{enq}}$$-i$ < 0.

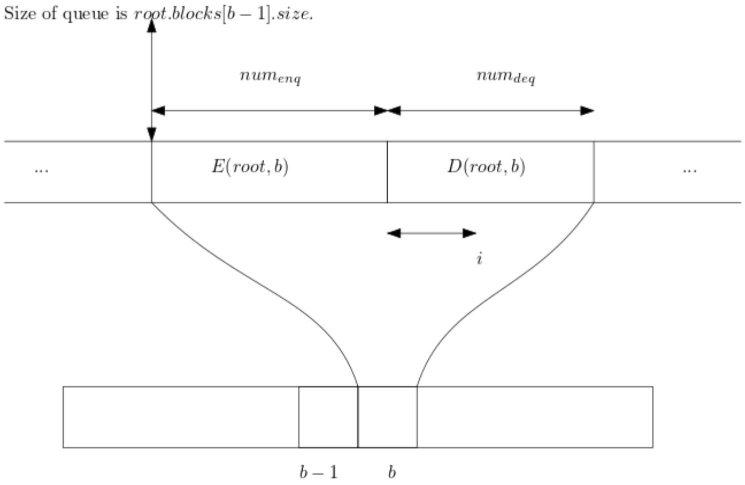
Proof. The claim follows immediately from Corollary numNullDeqBlock 4.39 and Lemma lem::numX 4.13. □

uteHead

Lemma 4.41. FindResponse(b, i) *returns Resp*(D_i(*root*, b)).

Proof. $D_i(root, b)$ is $D_{\text{root.blocks[}b-1\text{].sum}_{\text{deq}}+i}(root)$ by Definition ordering 4.11 and Lemma lem::sumX 4.14. $D_i(root, b)$ returns null at Line returnNull 20 if root.blocks[$b-1$].size + root.blocks[b].num$_{\text{enq}}$ $- i < 0$ and *Resp*($D_i(root, b)$) = null in this case by Lemma nullReturn 4.40. Otherwise, if $D_i(root, b)$ is the eth non-null Dequeue in L it should return the eth enqueued value. By Lemma numberOfNND 4.38 there are root.blocks[$b-$1].sum$_{\text{enq}}$ $-$ root.blocks[$b-1$].size non-null Dequeue operations in root.blocks[$0\cdots b-$1]. The Dequeues in root.blocks[b] before $D_i(root, b)$ are non-null Dequeues. So $D_i(root, b)$ is the eth non-null Dequeue where $e = i +$ root.blocks[$b-1$].sum$_{\text{deq}}$ $-$ root.blocks[$b-1$].size (Line computeE 22). See Figure computeResponseDetail 11.

After computing e at Line computeE 22, the code finds b, i such that $E_i(root, b) = E_e(root)$ using DoublingSearch and then finds its element using GetEnqueue (Line findAnswer 23). Correctness of DoublingSearch and GetEnqueue routines are shown in Lemmas dsearch 4.26 and get 4.27. □



eDetail

Fig. 11. The position of $D_i(root, b)$.

Correct

Lemma 4.42. *The responses to operations in our algorithm are the same as in the sequential execution in the order given by L.*

computeHead

Proof. Enqueue operations do not return any value. By Lemma 4.41, the response of a Dequeue in our algorithm is the same as its response in the sequential execution of L. □

Theorem 4.43 (Main). *The queue implementation is linearizable.*

linearSat linearCorrect

Proof. The theorem follows from Lemmas 4.34 and 4.42. □

DBLP:conf/stoc/GolabHW11 ordering

Remark. In fact our algorithm is strongly linearizable as defined in [14]. By Definition 4.11 the linearization ordering of operations will not change as blocks containing new operations are appended to the root.

5 ANALYSIS

In this section, we analyze the number of CAS invocations and the time complexity of our algorithm.

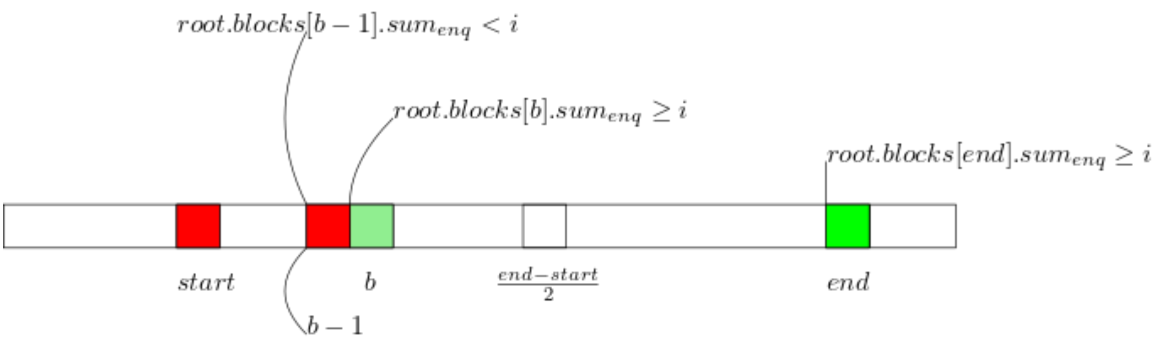
Proposition 5.1. *An* Enqueue *or* Dequeue *operation does at most* $14 \log p$ CAS *operations.*

Proof. In each level of the tree Refresh is invoked at most two times, and every Refresh invokes at most seven CASes, one in Line 70 and two from each Advance in Line 64 or 71. ☐

rchTime Lemma 5.2 (DoublingSearch Analysis). *If the* element *enqueued by* $E_i(root, b) = E_e(root)$ *is the response to some* Dequeue *operation in* root.blocks[end], *then* DoublingSearch(e, end) *takes* $O\big(\log($root.blocks[b].size + root.blocks[end].size$)\big)$ *steps.*

Proof. First we show $end - b - 1 \leq 2\times$root.blocks[$b-1$].size+root.blocks[end].size. There can be at most root.blocks[b].size Dequeues in root.blocks[$b+1 \cdots end-1$]; otherwise all elements enqueued by root.blocks[b] would be dequeued before root.blocks[end]. Furthermore, in the execution of queue operations in the linearization ordering, the size of the queue becomes root.blocks[end].size after the operations of root.blocks[end]. The final size of the queue after root.blocks[$1 \cdots end$] is root.blocks[end].size. After an execution on a queue, the *size* of the queue is greater than or equal to *#enqueues* − *#dequeues* in the execution. We know the number of dequeues in root.blocks[b+1 ⋯ end−1] is less than root.blocks[b].size, therefore in root.blocks[$b+1 \cdots end-1$] there cannot be more than root.blocks[b].size + root.blocks[end].size Enqueues. Overall there can be at most $2 \times$ root.blocks[b].size + root.blocks[end].size operations in root.blocks[$b+1 \cdots end-1$] and since from Line 68 we know that the num field of every block in the tree is greater than 0, each block has at least one operation, so there are at most $2\times$ root.blocks[b].size + root.blocks[end].size blocks in between root.blocks[b] and root.blocks[end]. So, $end - b - 1 \leq 2\times$root.blocks[b].size+root.blocks[end].size.

Thus, the doubling search reaches start such that the root.blocks[start].sum$_{\text{enq}}$ is less than e in $O\big(\log($root.blocks[b].size+root.blocks[end].size$)\big)$ steps. See Figure 12. After Line 41, the binary search that finds b also takes $O\big(\log($root.blocks[b].size+root.blocks[end].size$)\big)$. Next, i is computed via the definition of sum$_{\text{enq}}$ in constant time (Line 43). ☐



$root.blocks[b-1].sum_{enq} < i$
$root.blocks[b].sum_{enq} \geq i$
$root.blocks[end].sum_{enq} \geq i$
$start$
b
$\frac{end-start}{2}$
end
$b-1$

oubling Fig. 12. Distance relations between start, b, end.

DeqTime Lemma 5.3 (Worst Case Time Analysis). *The worst case number of steps for an* Enqueue *is* $O(\log^2 p)$ *and for a* Dequeue, *is* $O(\log^2 p + \log q_e + \log q_d)$, *where* q_d *is the size of the queue when the* Dequeue *is linearized and* q_e *is the size of the queue at the time the response of the* Dequeue *is linearized.*

Proof. Enqueue consists of creating a block and appending it to the tree. The first part takes constant time. To propagate the operation to the root the algorithm tries at most two Refreshes in each node of the path from the leaf to the root (Lines 52, 53). We can see from the code that each Refresh takes a constant number of steps and does $O(1)$ CASes. Since the height of the tree is $\Theta(\log p)$, Enqueue takes $O(\log p)$ steps.

A Dequeue creates a block whose element is null, appends it to the tree, computes its rank among non-null dequeues, finds the corresponding enqueue and returns the response. The first two parts are similar to an Enqueue operation and take $O(\log p)$ steps. To compute the rank of a Dequeue in $D(n)$, the Dequeue calls IndexDequeue(). IndexDequeue does $O(1)$ steps in each level which takes $O(\log p)$ steps. If the response to the Dequeue is null, FindResponse returns null in $O(1)$ steps. Otherwise, if the response to a dequeue in root.blocks[end] is in root.blocks[b] the DoublingSearch takes $\Theta(\log($root.blocks[b].size+root.blocks [end].size) by Lemma 5.2, which is $O(\log q_e + \log q_d)$. Each search in GetEnqueue() takes $O(\log p)$ steps since there are at most p subblocks in a block (Lemma 4.25), so GetEnqueue() takes $O(\log^2 p)$ steps. □

Lemma 5.4 (Amortized Worst-case Analysis). *The amortized number of steps for an* Enqueue *or* Dequeue *is $O(\log^2 p + \log q)$, where q is the size of the queue when the operation is linearized.*

Proof. If we split the DoublingSearch time cost between the corresponding Enqueue and Dequeue, each operation takes $O(\log^2 p + q)$ steps. □

Observation 6. *If the maximum number of concurrent processes at any time in an execution is c, then the amortized worst-case step complexity is $O(\log p \log c + \log q)$ per operations. Furthermore, in a sequential, execution where $c = 1$, the step complexity of our algorithm is $\Theta(\log p + \log q)$ per operation.*

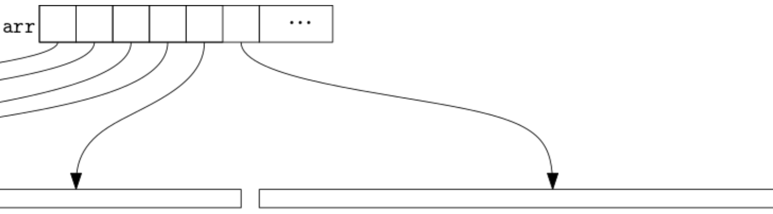
Proof. The analysis is similar to the two previous Lemmas, but by Lemma 4.25 each BinarySearch in each call of GetEnqueue takes $O(\log c)$ steps. □

Theorem 5.5. *The queue implementation is wait-free.*

Proof. To prove the claim, it is sufficient to show that every Enqueue and Dequeue operation terminates after a finite number of its own steps. This is directly concluded from Lemma 5.3. □

6 REDUCING SPACE

Reducing Space Usage. The blocks arrays defined in our algorithm are unbounded. To use $O(n)$ space in each node where n is the total number of operations, instead of unbounded arrays, we could use the memory model of the wait-free vector introduced by Feldman, Valera-Leon, and Damian [11]. We can create an array called arr of pointers to array segments (see Figure 13). When a process wishes to write into location head it checks whether arr[$\lfloor \log$ head$\rfloor$] points to an array or not. If not, it creates a shared array of size $2^{\lfloor \log \text{head} \rfloor}$ and tries to CAS a pointer to the created array into arr[$\lfloor \log$ head$\rfloor$]. Whether the CAS is successful or not, arr[$\lfloor \log$ head$\rfloor$] points to an array. When a process wishes to access the ith element it looks up arr[$\lfloor \log i \rfloor$][$i - 2^{\lfloor \log i \rfloor}$], which takes $O(1)$ steps. The CAS Retry Problem does not happen here because if n elements are appended to the array, then only $O(p \times \log n)$ CAS steps have happened on the array arr. Furthermore, at most p arrays with size $2^{\lfloor \log i \rfloor}$ are allocated by processes while processes try to do the CAS on arr[i]. Jayanti and Shun [19] present a way to initialize wait-free arrays in constant steps. The time taken to allocate arrays in an execution containing n operations is $O(\frac{p \log n}{n})$ per operation, which is negligible if $n >> p$. The vector implementation also has a mechanism for doubling arr when necessary, but this happens very rarely since increasing arr from s to $2s$ increases the capacity of the vector from 2^s to 2^{2s}.



ngArray

Fig. 13. Array segments.

Garbage Collection. We did not handle garbage collection: Enqueue operations remain in the nodes even after their elements have been dequeued. We can keep track of the blocks in the root whose operations are all terminated, i.e., all enqueues have been dequeued, and the responses of all dequeues have been computed. We call these blocks *finished blocks*. If we help the operations of all processes to compute their responses, then we can say if block B is finished, then all blocks before B are also finished. Knowing the most recent finished block in a node, we can reclaim the memory taken by finished blocks. We cannot use arrays (or vectors) to throw the garbage blocks away. We need a data structure that supports `tryAppend()`, `read(i)`, `write(i)` and `split(i)` operations in $O(\log n)$ time, where `split(i)` removes all the indices less than `i`. If each process tries to do the garbage collection once every p^2 operations on the queue, then the amortized complexity remains the same. We can use a concurrent implementation of a persistent red-black trees for this DBLP:conf/afp/Ok [32]. Bashari and Woelfel DBLP:conf/podc/BashariW21 [3] used persistent red-black trees in a similar way.

Tarjan Tar83 [37, Sec. 4.2] described a split algorithm for red-black trees that runs in logarithmic time.

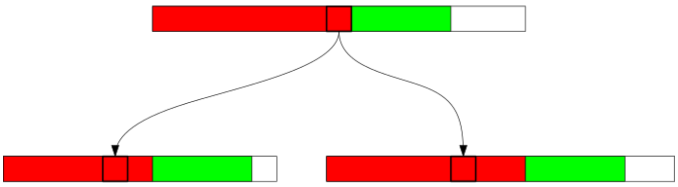


Fig. 14. Finished blocks are shown with red color and unfinished blocks are shown with green color. All the subblocks of a finished block are also finished.

edBlock

7 FUTURE DIRECTIONS

Our focus in constructing our queue was on optimizing step complexity for worst-case executions. However, our approach does have a higher cost in the best case (for example, when an operation runs in isolation). It may be possible to make our queue adaptive by having an operation capture a starting node in the tournament tree (as in ?? [?]) rather than having a statically assigned leaf as its starting point. Another way to make our queue more practical might be to use it as the slow path in the fast-path slow-path methodology of Kogan and Petrank 10.1145/2370036.2145835 [23] to get a queue that has good performance in practice while also having good worst-case step complexity.

There is a gap between our implementation, which takes $O(\log^2 p + \log q)$ steps per operation, and Attiya and Fouren's $\Omega(\min(c, \log\log p))$ lower bound DBLP:conf/opodis/AttiyaF17 [2]. It would be interesting to determine how the true step complexity of lock-free queues (or, more generally, bags) depends on p.

We believe the approach used here to implement a lock-free queue could also be applied to obtain other lock-free data structures with a polylogarithmic number of steps per operation. For example, we could implement a kind of vector data structure that stores a sequence and provides three operations: `append(e)` to add an element `e` to the end of the sequence, `get(i)` to read the `i`th element in the sequence, and `index(e)` to compute the position of element `e` in the sequence. **[[Can omit rest of this paragraph to save space]]** An `append(e)` is implemented like `Enqueue(e)` in $O(\log p)$ steps. A `get(i)` calls `DoublingSearch` but with a `BinarySearch` on the entire `root.blocks` array, taking $O(\log n + \log^2 p)$ when the vector has n elements. An `index(e)` is similar to `IndexDequeue` (except operating on enqueues instead of dequeues) and takes $O(\log p)$ steps. **[[doublecheck all preceding claims for accuracy]]**. As future work we would like to investigate whether a similar approach could be used for stacks or deques.

REFERENCES

fekDT95 [1] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

tiyaF17 [2] Hagit Attiya and Arie Fouren. Lower bounds on the amortized time complexity of shared objects. In *21st International Conference on Principles of Distributed Systems*, volume 95 of *LIPIcs*, pages 16:1–16:18, 2017.

hariW21 [3] Benyamin Bashari and Philipp Woelfel. An efficient adaptive partial snapshot implementation. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 545–555, 2021.

lvinG05 [4] Robert Colvin and Lindsay Groves. Formal verification of an array-based nonblocking queue. In *10th International Conference on Engineering of Complex Computer Systems*, pages 507–516. IEEE Computer Society, 2005.

David04 [5] Matei David. A single-enqueuer wait-free queue implementation. In *Proc. 18th International Conference on Distributed Computing*, volume 3274 of *LNCS*, pages 132–143. Springer, 2004.

DPDK [6] DPDK Project. Data plane development kit. www.dpdk.org.

EFHR14 [7] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 332–340, 2014.

ERW12 [8] Faith Ellen, Vijaya Ramachandran, and Philipp Woelfel. Efficient fetch-and-increment. In *Proc. International Symposium on Distributed Computing*, volume 7611 of *LNCS*, pages 16–30. Springer, 2012.

27-2_20 [9] Faith Ellen and Philipp Woelfel. An optimal implementation of fetch-and-increment. In *Proc. 27th International Symposium on Distributed Computing*, volume 8205 of *LNCS*, pages 284–298. Springer, 2013.

FK14 [10] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.

7073592 [11] Steven Feldman, Carlos Valera-Leon, and Damian Dechev. An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):654–667, 2016.

FR04 [12] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.

tamST10 [13] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proc. 14th International Conference on Principles of Distributed Systems*, volume 6490 of *LNCS*, pages 302–317. Springer, 2010.

labHW11 [14] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proc. 43rd ACM Symposium on Theory of Computing*, pages 373–382, 2011.

.102808 [15] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124âĂŞ149, 1991.

manSS07 [16] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In *Proc. 11th International Conference on Principles of Distributed Systems*, volume 4878 of *LNCS*, pages 401–414. Springer, 2007.

anti98a [17] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998.

antiP05 [18] Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *LNCS*, pages 408–419. Springer, 2005.

antiS21 [19] Siddhartha Jayanti and Julian Shun. Fast arrays: Atomic arrays with constant time initialization. In *Proc. 35th International Symposium on Distributed Computing*, volume 209 of *LIPIcs*, pages 25:1–25:19, 2021.

KW18 [20] Pankaj Khanchandani and Roger Wattenhofer. On the importance of synchronization primitives with low consensus numbers. In *Proc. 19th International Conference on Distributed Computing and Networking*, pages 18:1–18:10, 2018.

oganH14 [21] Alex Kogan and Maurice Herlihy. The future(s) of shared data structures. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 30–39, 2014.

oganP11 [22] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proc. 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 223–234, 2011.

2145835 [23] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141âĂŞ150, 2012.

ozesS08 [24] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free FIFO queues. *Distributed Computing*, 20(5):323–341, 2008.

MP91 [25] Henry Massalin and Carlton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, 1991.

MS98 [26] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

MKLLP22 [27] Gal Milman-Sela, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. BQ: A lock-free queue with batching. *ACM Trans. Parallel Comput.*, 9(1), March 2022.

irNSS05 [28] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *Proc. 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–262, 2005.

isonA13 [29] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 103–112, 2013.

Nik19 [30] Ruslan Nikolaev. A scalable, portable, and memory-efficient lock-free FIFO queue. In *Proc. 33rd International Symposium on Distributed Computing*, volume 146 of *LIPIcs*, pages 28:1–28:16, 2019.

3538572 [31] Ruslan Nikolaev and Binoy Ravindran. wCQ: A fast wait-free queue with bounded memory usage. In *Proc. 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 307–319, 2022.

asaki96 [32] Chris Okasaki. Functional data structures. In *Advanced Functional Programming, Second International School*, volume 1129 of *LNCS*, pages 131–158. Springer, 1996.

RC17 [33] Pedro Ramalhete and Andreia Correia. Poster: A wait-free queue with wait-free memory reclamation. *SIGPLAN Not.*, 52(8):453–454, January 2017.

Rup16 [34] Eric Ruppert. Analysing the average time complexity of lock-free data structures. Presented at BIRS Workshop on Complexity and Analysis of Distributed Algorithms, 2016. Available from http://www.birs.ca/videos/2016.

afiei09 [35] Niloufar Shafiei. Non-blocking array-based algorithms for stacks and queues. In *Proc. 10th International Conference on Distributed Computing and Networking*, volume 5408 of *LNCS*, pages 55–66. Springer, 2009.

Sha15 [36] Niloufar Shafiei. Non-blocking doubly-linked lists with good amortized complexity. In *Proc. 19th International Conference on Principles of Distributed Systems*, volume 46 of *LIPIcs*, pages 35:1–35:17, 2015.

Tar83 [37] Robert Endre Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, USA, 1983.

Tre86 [38] R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

igasZ01 [39] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143, 2001.

YangM16 [40] Chaoran Yang and John M. Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 16:1–16:13, 2016.