A Wait-free Queue with Polylogarithmic Step Complexity

Hossein Naderibeni
York University
Toronto, Ontario, Canada

Eric Ruppert
York University
Toronto, Ontario, Canada

ABSTRACT

We present a novel linearizable wait-free queue implementation using single-word CAS instructions. Previous lock-free queue implementations from CAS all have amortized step complexity of $\Omega(p)$ per operation in worst-case executions, where p is the number of processes that access the queue. Our new wait-free queue takes $O(\log p)$ steps per enqueue and $O(\log^2 p + \log q)$ steps per dequeue, where q is the size of the queue. A bounded-space version of the implementation has $O(\log p \log(p+q))$ amortized step complexity per operation.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; **Concurrent algorithms**.

KEYWORDS

concurrent data structures, wait-free queues

ACM Reference Format:
Hossein Naderibeni and Eric Ruppert. 2023. A Wait-free Queue with Polylogarithmic Step Complexity. In *ACM Symposium on Principles of Distributed Computing (PODC '23), June 19–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3583668.3594565

1 INTRODUCTION

There has been a great deal of research in the past several decades on the design of shared queues. Besides being a fundamental data structure, queues are used in significant concurrent applications, including OS kernels [28], memory management [4], synchronization [26], and sharing resources or tasks. We focus on shared queues that are *linearizable* [16], meaning that operations appear to take place atomically, and *lock-free*, meaning that some operation on the queue is guaranteed to complete regardless of how asynchronous processes are scheduled to take steps.

The lock-free MS-queue of Michael and Scott [29] 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 in a system of p processes. (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 in the execution.) Numerous papers have suggested modifications to the MS-queue [17, 23, 24, 27, 30, 31, 37], 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 [32] called this the *CAS retry problem*. The same problem occurs in array-based implementations of queues [6, 14, 40, 44]. Solutions that tried to sidestep this problem using fetch&increment [32, 35, 36, 45] 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 [39]. For example, lock-free lists [13, 41], stacks [43] and search trees [9] 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 [2] proved that amortized $\Omega(c)$ steps per operation are indeed necessary for any CAS-based implementation of a lock-free bag data structure, which provides operations to insert an element or remove an arbitrary element (chosen nondeterministically). Since a queue trivially implements a bag, this lower bound also applies to queues. Although this might seem to settle the step complexity of lock-free queues, the lower bound holds only if c is $O(\log \log p)$ so it should be stated more precisely as an amortized bound of $\Omega(\min(c, \log \log p))$ steps per operation.

We exploit this loophole. We show it is, in fact, possible for a linearizable queue to have step complexity sublinear in p. Our queue is the first whose step complexity is polylogarithmic in p and in q, the number of elements in the queue. It is *wait-free*, meaning that every operation is guaranteed to complete within a finite number of its own steps. For ease of presentation, we first give an unbounded-space construction where enqueues take $O(\log p)$ steps and dequeues take $O(\log^2 p + \log q)$ steps, and then modify it to bound the space while having $O(\log p \log(p+q))$ amortized step complexity per operation. Moreover, each operation does $O(\log p)$ CAS instructions in the worst case, whereas previous lock-free queues use $\Omega(p)$ CAS instructions, even in an amortized sense. Both versions of our queue use single-word CAS on reasonably-sized words. We assume that a word is large enough to store an item to be enqueued (or at least a pointer to it). We also assume that the number of operations performed on the queue can be stored (in binary) in $O(1)$ words. This is analogous to the assumption for the classical RAM model that the number of bits per word is logarithmic in the problem size. For the space-bounded version, we unlink unneeded objects from our data structure. We do not address the orthogonal problem of reclaiming memory; we assume a safe garbage collector, such as the highly optimized one that Java provides.

PODC '23, June 19–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0121-4/23/06…$15.00
https://doi.org/10.1145/3583668.3594565

Our queue uses a binary tree, called the *ordering tree*, where each process has its own leaf. A process adds its operations to its leaf. As in previous work (e.g., [1, 19]), operations are propagated from the leaves up to the root in a cooperative way that ensures wait-freedom and avoids the CAS retry problem. Operations in the root are ordered, and this order is used to linearize the operations and compute their responses. Explicitly storing operations in the tree nodes would be too costly. Instead, we use a novel implicit representation of sets of operations that allows us to quickly merge two sets from the children of a node, and quickly access any operation in a set. A preliminary version of this work appeared in [33].

2 RELATED WORK

List-based Queues. The MS-queue [29] is a lock-free queue that has stood the test of time. The standard Java Concurrency Package includes a version of it. See [29] 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.

Many papers have described ways to reduce contention in the MS-queue. Moir et al. [31] added an elimination array that allows an enqueue to pass its enqueued value directly to a concurrent dequeue when the queue is empty. 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 [17] 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 order 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 [23] improved the MS-queue's performance using *futures*. Operations return future objects instead of responses. Later, when an operation's response is needed, it is evaluated using the 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. [30] extended this approach to allow batches to mix enqueues and dequeues. 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.

In the MS-queue, an enqueue requires two CAS steps. Ladan-Mozes and Shavit [27] presented an optimistic queue implementation that 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 necessary by traversing the list. This fixing is rare in practice, but it yields an amortized complexity of $\Omega(qp)$ steps per operation in the worst case.

Kogan and Petrank [24] used Herlihy's helping technique [15] to make the MS-queue wait-free. Then, they introduced the fast-path slow-path methodology [25] for making data structures wait-free: the fast path has good performance and the slow path guarantees termination. They applied their methodology to combine the MS-queue (as the fast path) with their wait-free queue (as the slow path). Ramalhete and Correia [37] added a different helping mechanism to the MS-queue. Although these approaches can perform well in practice, the amortized step complexity remains $\Omega(p)$.

Array-Based Queues. Arrays can be used to implement queues with bounded capacity [6, 40, 44]. Dequeues and enqueues update indices of the front and back elements using CAS instructions. Gidenstam, Sundell, and Tsigas [14] 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 [32] 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. (It was shown recently that this implementation can be modified to use single-word CAS instructions rather than double-width CAS [38].) 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. This approach is similar to the fast-path slow-path methodology [25]. Other array-based queues [35, 36, 45] also used this methodology. In worst-case executions that use the slow path, they 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 [18] 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 [12] used their own universal construction based on fetch&add and LL/SC instructions to implement a queue, but its step complexity is also $\Omega(p)$.

Restricted Queues. David gave the first sublinear-time queue [7], 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 gave a wait-free polylogarithmic queue [19], but only for a single dequeuer. Our ordering tree is similar to the tree structure they use to agree on a linearization ordering. Concurrently with our work, which first appeared in [33], Johnen, Khattabi and Milani [20] built on [19] to give a wait-free queue that achieves $O(\log p)$ steps for enqueue operations but fails to achieve polylogarithmic step complexity for dequeues: their dequeue operations take $O(k \log p)$ steps if there are k dequeuers.

Other Primitives. Khanchandani and Wattenhofer [21] gave a wait-free queue with $O(\sqrt{p})$ step complexity using non-standard synchronization primitives called half-increment and half-max, which can be viewed as 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. Ellen, Ramachandran and Woelfel [10] gave an implementation of fetch&increment objects that uses a polylogarithmic number of steps per operation. Like our queue, they also use a tree structure similar to the universal construction

of [1] to keep track of the operations that have been performed. However, our construction requires more intricate data structures to represent sets of operations, since a queue's state cannot be represented as succinctly as the single-word state of a fetch&increment object. Ellen and Woelfel [11] gave an improved implementation of fetch&increment with better step complexity.

3 QUEUE IMPLEMENTATION

3.1 Overview

Our *ordering tree* data structure is used to agree on a total ordering of the operations performed on the queue. It is a static binary tree of height $\lceil \log_2 p \rceil$ with one leaf for each process. Each tree node stores an array of *blocks*, where each block represents a sequence of enqueues and a sequence of dequeues. See Figure 1 for an example. In this section, we use an infinite array of blocks in each node. Section 6 describes how to replace the infinite array by a representation that uses bounded space.

To perform an operation on the queue, a process P appends a new block containing that operation to the *blocks* array in P's leaf. Then, P attempts to propagate the operation to each node along the path from that leaf to the root of the tree. We shall define a total order on all operations that have been propagated to the root, which will serve as the linearization ordering of the operations.

To propagate operations from a node v's children to v, P first observes the blocks in both of v's children that are not already in v, creates a new block by combining information from those blocks, and attempts to append this new block to v's *blocks* array using a CAS. Following [19], we call this a 3-step sequence a Refresh on v. A Refresh's CAS may fail if there is another concurrent Refresh on v. However, since a successful Refresh propagates multiple pending operations from v's children to v, we can prove that if two Refreshes by P on v fail, then P's operation has been propagated to v by some other process, so P can continue onwards towards the root.

Now suppose P's operation has been propagated all the way to the root. If P's operation is an enqueue, it has obtained a place in the linearization ordering and can terminate. 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 dequeue (since the dequeue may have been propagated to the root by some other process). P does this by finding the dequeue's location in each node along the path from the leaf to the root. Then, P determines whether the queue is empty when its dequeue is linearized. If so, it returns null and we call it a *null dequeue*. If not, P computes the rank r of its dequeue among all non-null dequeues in the linearization ordering. (We say that the rth element in a sequence has *rank* r within that sequence.) P then returns the value of the rth enqueue in the linearization.

We must choose what to store in each block so that the following tasks can be done efficiently.

(T1) Construct a block for node v that represents the operations in consecutive blocks in v's children, as required for a Refresh.
(T2) Given a dequeue in a leaf that has been propagated to the root, find that operation's position in the root's *blocks* array.
(T3) Given a dequeue's position in the root, decide if it is a null dequeue (i.e., if the queue is empty when it is linearized) or determine the rank r of the enqueue whose value it returns.
(T4) Find the rth enqueue in the linearization ordering.

Since these tasks depend on the linearization ordering, we describe that ordering next.

3.2 Linearization Ordering

Performing a double Refresh at each node along the path from the leaf to the root ensures a block containing the operation is appended to the root before the operation completes. 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 the root's blocks array. Thus, we linearize operations according to the block they belong to in the root's array. We can choose how to order operations in the same block, since they must be concurrent.

Each block in a leaf represents one operation. Each block B in an internal node v results from merging several consecutive blocks from each of v's children. The merged blocks in v's children are called the *direct subblocks* of B. A block B' is a *subblock* of B if it is a direct subblock of B or a subblock of a direct subblock of B. A block B represents the set of operations in all of B's subblocks in leaves of the tree. The operations propagated by a Refresh are all pending when the Refresh occurs, so there is at most one operation per process. Hence, a block represents at most p operations in total. Moreover, we never append empty blocks, so each block represents at least one operation and it follows that a block can have at most p direct subblocks.

As mentioned above, we are free to order operations within a block however we like. We order the enqueues and dequeues separately, and put the operations propagated from the left child before the operations from the right child. More formally, we inductively define sequences $E(B)$ and $D(B)$ of the enqueues and dequeues represented by a block B. If B is a block in a leaf representing an enqueue operation, its enqueue sequence $E(B)$ is that operation and its dequeue sequence $D(B)$ is empty. If B is a block in a leaf representing a dequeue, $D(B)$ is that single operation and $E(B)$ is empty. If B is a block in an internal node v with direct subblocks $B_1^L, \ldots, B_\ell^L$ from v's left child and $B_1^R, \ldots, B_r^R$ from v's right child, then B's operation sequences are defined by the concatenations

$$\begin{aligned} E(B) &= E(B_1^L) \cdots E(B_\ell^L) \cdot E(B_1^R) \cdots E(B_r^R) \text{ and} \\ D(B) &= D(B_1^L) \cdots D(B_\ell^L) \cdot D(B_1^R) \cdots D(B_r^R) \end{aligned} \tag{3.1}$$

We say the block B *contains* the operations in $E(B)$ and $D(B)$.

When linearizing the operations propagated to the root, we must choose how to order operations within a block. We choose to put each block's enqueues before its dequeues. Thus, if the root's blocks array contains blocks $B_1, \ldots, B_k$, the linearization ordering is

$$L = E(B_1) \cdot D(B_1) \cdot E(B_2) \cdot D(B_2) \cdots E(B_k) \cdot D(B_k). \tag{3.2}$$

3.3 Designing a Block Representation to Solve Tasks (T1) to (T4)

Each node of the ordering tree has an infinite array called *blocks*. To simplify the code, *blocks*[0] is initialized with an empty block B_0, where $E(B_0)$ and $D(B_0)$ are empty sequences. Each node's *head* index stores the position in the *blocks* array to be used for the next attempt to append a block.

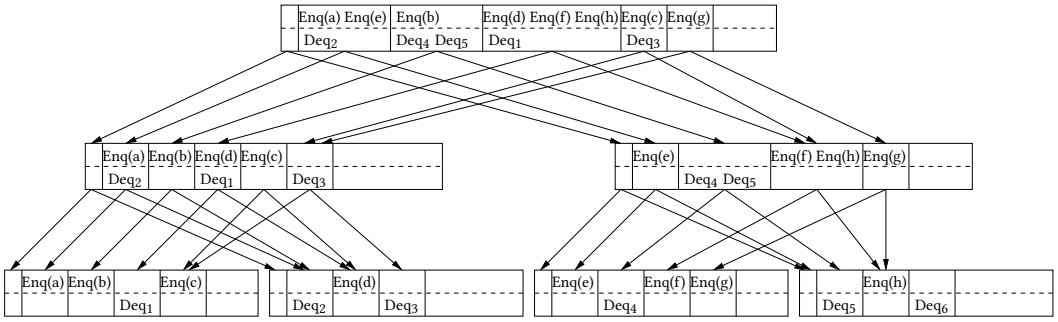

Figure 1: An example ordering tree with four processes. We show explicitly the enqueue sequence and dequeue sequence represented by each block in the *blocks* arrays of the seven nodes. The leftmost element of each *blocks* array is a dummy block. Arrows represent the indices stored in end_{left} and end_{right} fields of blocks (as described in Section 3.3). The fourth process's Deq_6 is still propagating towards the root. The linearization order for this tree is Enq(a) Enq(e) Deq_2 | Enq(b) Deq_4 Deq_5 | Enq(d) Enq(f) Enq(h) Deq_1 | Enq(c) Deq_3 | Enq(g), where vertical bars indicate boundaries of blocks in the root.

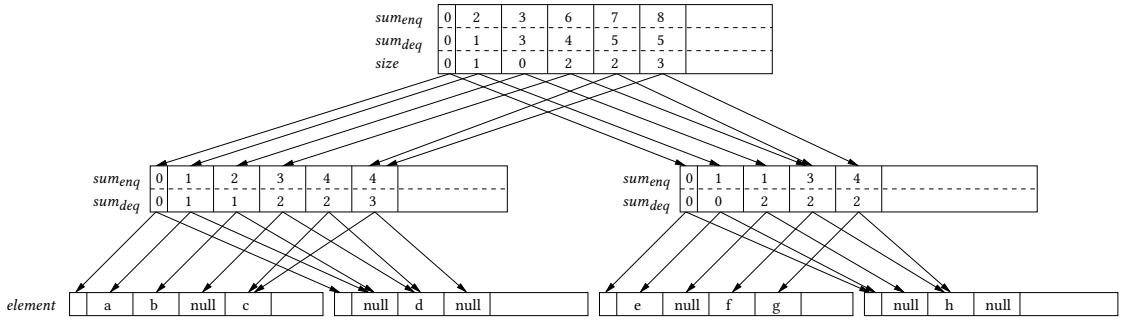

Figure 2: The actual, implicit representation of the tree shown in Figure 1. The leaf blocks simply show the *element* field. Internal blocks show the sum_{enq} and sum_{deq} fields, and end_{left} and end_{right} fields are shown using arrows as in Figure 1. Root blocks also have the additional *size* field. The *super* field is not shown.

```
▷ Shared variable
• Node root                    ▷ root of binary tree of Nodes with one leaf per process
▷ Thread-local variable
• Node leaf                    ▷ process's leaf in the tree
► Node
• Node left, right, parent     ▷ tree pointers initialized when creating the tree
• Block[0..∞] blocks           ▷ blocks that have been propagated to this node;
                               ▷ blocks[0] is empty block whose integer fields are 0
• int head                     ▷ position to append next block to blocks, initially 1
► Block
• int sum_enq, sum_deq         ▷ number of enqueues, dequeues in blocks array
                               ▷ up to this block (inclusive)
• int super                    ▷ approximate index of superblock in parent.blocks
▷ Blocks in internal nodes have the following additional fields
  • int end_left, end_right    ▷ index of last direct subblock in the left and right child
▷ Blocks in leaf nodes have the following additional field
  • Object element            ▷ x for Enqueue(x) operation; otherwise null
▷ Blocks in the root node have the following additional field
  • int size                   ▷ size of queue after performing all operations up
                               ▷ to the end of this block
```

Figure 3: Objects used in the ordering tree data structure.

If a block contained an explicit representation of its sequences of enqueues and dequeues, it would take $\Omega(p)$ time to construct a block, which would be too slow for task (T1). Instead, the block stores an implicit representation of the sequences. We now explain how we designed the fields for this implicit representation. Refer to Figure 2 for an example showing how the tree in Figure 1 is actually represented, and Figure 3 for the definitions of the fields of blocks and nodes.

A block in a leaf represents a single enqueue or dequeue. The block's *element* field stores the value enqueued if the operation is an enqueue, or null if the operation is a dequeue.

Each block in an internal node v has fields end_{left} and end_{right} that store the indices of the block's last direct subblock in v's left and right child. Thus, the direct subblocks of $v.blocks[b]$ are

$$v.left.blocks[v.blocks[b-1].end_{left}+1..v.blocks[b].end_{left}] \text{ and}$$
$$v.right.blocks[v.blocks[b-1].end_{right}+1..v.blocks[b].end_{right}]. \quad (3.3)$$

The end_{left} and end_{right} fields allow us to navigate to a block's direct subblocks. Blocks also store some prefix sums: $v.blocks[b]$ has two fields sum_{enq} and sum_{deq} that store the total numbers of enqueues and dequeues in $v.blocks[1..b]$. We use these to search for a particular operation. For example, consider finding the rth enqueue E_r in the linearization. A binary search for r on sum_{enq} fields of the root's blocks finds the block containing E_r. If we know a block B in a node v contains E_r, we can use the sum_{enq} field again to determine which child of v contains E_r and then do a binary search among the direct subblocks of B in that child. Thus, we work our way down the tree until we find the leaf block that stores E_r

explicitly. We shall show that the binary search in the root can be done in $O(\log p + \log q)$ steps, and the binary search within each other node along the path to a leaf takes $O(\log p)$ steps, for a total of $O(\log^2 p + \log q)$ steps for task (T4).

A block is called the *superblock* of all of its direct subblocks. To facilitate task (T2), each block B has a field *super* that contains the (approximate) index of its superblock in the parent node's *blocks* array (it may differ from the true index by 1). This allows a process to determine the true location of the superblock by checking the end_{left} or end_{right} values of just two blocks in the parent node. Thus, starting from an operation in a leaf's block, one can use these indices to track the operation up the path to the root, and determine the operation's location in a root block in $O(\log p)$ time.

Now consider task (T3). To determine whether the queue is empty when a dequeue occurs, each block in the root has a *size* field storing the number of elements in the queue after all operations in the linearization up to that block (inclusive) have been done. We can determine which dequeues in a block B_d in the root are null dequeues using $B_{d-1}.size$, which is the size of the queue just before B_d's operations, and the number of enqueues and dequeues in B_d. Moreover, the total number of non-null dequeues in blocks $B_1, \ldots, B_{d-1}$ is $B_{d-1}.sum_{enq} - B_{d-1}.size$. We can use this information to determine the rank of a non-null dequeue in B_d among all non-null dequeues in the linearization, which is the rank (among all enqueues) of the enqueue whose value the dequeue should return.

Having defined the fields required for tasks (T2), (T3) and (T4), we can easily see how to construct a new block B during a Refresh in $O(1)$ time. A Refresh on node v reads the values h_ℓ and h_r of the *head* fields of v's children and stores $h_\ell - 1$ and $h_r - 1$ in $B.end_{left}$ and $B.end_{right}$. Then, we can compute

$$\begin{aligned} B.sum_{enq} &= v.left.blocks[B.end_{left}].sum_{enq} \\ &\quad + v.right.blocks[B.end_{right}].sum_{enq}. \end{aligned}$$

For a block B in the root, $B.size$ is computed using the *size* field of the previous block B' and the number of enqueues and dequeues in B:

$$\begin{aligned} B.size &= \max(0, B'.size + (B.sum_{enq} - B'.sum_{enq}) \\ &\quad - (B.sum_{deq} - B'.sum_{deq})). \end{aligned}$$

The only remaining field is $B.super$. When the block B is created for a node v, we do not yet know where its superblock will eventually be installed in v's parent. So, we leave $B.super$ blank. Soon after B is installed, some process will set $B.super$ to a value read from the *head* field of v's parent. We shall show that this happens soon enough that $B.super$ can differ from the true index of B' by at most 1.

3.4 Details of the Implementation

We now discuss the queue implementation in more detail. Pseudocode is provided in Figure 4.

An **Enqueue**(e) appends a block to the process's leaf. The block has $element = e$ to indicate it represents an Enqueue(e) operation. It suffices to propagate the operation to the root and then use its position in the linearization for future Dequeue operations.

A **Dequeue** also appends a block to the process's leaf. The block has *element* = null to indicate that it represents a Dequeue operation. After propagating the operation to the root, it computes its position in the root using IndexDequeue and then computes its response by calling FindResponse.

Append(B) first adds the block B to the invoking process's leaf. The leaf's *head* field stores the first empty slot in the leaf's *blocks* array, so the Append writes B there and increments *head*. Since Append writes only to the process's own leaf, there cannot be concurrent updates to a leaf. Append then calls Propagate to ensure the operation represented by B is propagated to the root.

Propagate(v) guarantees that any blocks that are in v's children when Propagate is invoked are propagated to the root. It uses the double Refresh idea described above and invokes two Refreshes on v in Lines 17 and 18. If both fail to add a block to v, it means some other process has done a successful Refresh that propagated blocks that were in v's children prior to line 17 to v. Then, Propagate recurses to $v.parent$ to continue propagating blocks up to the root.

A **Refresh** on node v creates a block representing the new blocks in v's children and tries to append it to $v.blocks$. Line 25 reads $v.head$ into the local variable h. Line 32 creates the new block to install in $v.blocks[h]$. If line 32 returns null instead of a new block, there were no new blocks in v's children to propagate to v, so Refresh can return true at line 33 and terminate. Otherwise, the CAS at line 35 tries to install the new block into $v.blocks[h]$. Either this CAS succeeds or some other process has installed a block in this location. Either way, line 36 then calls **Advance** to advance v's head index from h to $h + 1$ and fill in the *super* field of the most recently appended block. The boolean value returned by Refresh indicates whether its CAS succeeded. A Refresh may pause after a successful CAS before calling Advance at line 36, so other processes help keep *head* up to date by calling Advance, either at line 29 during a Refresh on v's parent or line 36 during a Refresh on v.

CreateBlock(v, i) is used by Refresh to construct a block to be installed in $v.blocks[i]$. The end_{left} and end_{right} fields store the indices of the last blocks appended to v's children, obtained by reading the *head* index in v's children. Since the sum_{enq} field should store the number of enqueues in $v.blocks[1..i]$ and these enqueues come from $v.left.blocks[1..new.end_{left}]$ and $v.blocks[1..new.end_{right}]$, line 45 sets sum_{enq} to the sum of $v.left.blocks[new.end_{left}].sum_{enq}$ and $v.right.blocks[new.end_{right}].sum_{enq}$. Line 47 sets num_{enq} to the number of enqueues in the new block by subtracting the number of enqueues in $v.blocks[1..i-1]$ from $new.sum_{enq}$. The values of $new.sum_{deq}$ and num_{deq} are computed similarly. Then, if *new* is going to be installed in the root, line 50 computes the *size* field, which represents the number of elements in the queue after the operations in the block are performed. Finally, if the new block contains no operations, CreateBlock returns null to indicate there is no need to install it.

Once a dequeue is appended to a block of the process's leaf and propagated to the root, the **IndexDequeue** routine finds the dequeue's location in the root. More precisely, IndexDequeue(v, b, i) computes the block in the root and the rank within that block of the ith dequeue of the block B stored in $v.blocks[b]$. Lines 72–74 compute the location of B's superblock in v's parent, taking into account the fact that $B.super$ may differ from the superblock's true index by one. The arithmetic in lines 76–79 compute the dequeue's rank within the superblock's sequence of dequeues, using (3.1).

```
1:  void Enqueue(Object e)
2:      let B be a new Block with element := e,
            sum_enq := leaf.blocks[leaf.head − 1].sum_enq + 1,
            sum_deq := leaf.blocks[leaf.head − 1].sum_deq
3:      Append(B)
4:  end Enqueue
5:  Object Dequeue()
6:      let B be a new Block with element := null,
            sum_enq := leaf.blocks[leaf.head − 1].sum_enq,
            sum_deq := leaf.blocks[leaf.head − 1].sum_deq + 1
7:      Append(B)
8:      ⟨b, i⟩ := IndexDequeue(leaf, leaf.head − 1, 1)
9:      return FindResponse(b, i)
10: end Dequeue
11: void Append(Block B) ▷ append block to leaf and propagate to root
12:     leaf.blocks[leaf.head] := B
13:     leaf.head := leaf.head + 1
14:     Propagate(leaf.parent)
15: end Append
16: void Propagate(Node v) ▷ propagate blocks from v's children to root
17:     if not Refresh(v) then                          ▷ double refresh
18:         Refresh(v)
19:     end if
20:     if v ≠ root then                                ▷ recurse up tree
21:         Propagate(v.parent)
22:     end if
23: end Propagate
24: boolean Refresh(Node v) ▷ try to append a new block to v.blocks
25:     h := v.head
26:     for each dir in {left, right} do
27:         childHead := v.dir.head
28:         if v.dir.blocks[childHead] ≠ null then
29:             Advance(v.dir, childHead)
30:         end if
31:     end for
32:     new := CreateBlock(v, h)
33:     if new = null then return true
34:     else
35:         result := CAS(v.blocks[h], null, new)
36:         Advance(v, h)
37:         return result
38:     end if
39: end Refresh
40: Block CreateBlock(Node v, int i)
41:     ▷ create new block for a Refresh to install in v.blocks[i]
42:     let new be a new Block
43:     new.end_left := v.left.head − 1
44:     new.end_right := v.right.head − 1
45:     new.sum_enq := v.left.blocks[new.end_left].sum_enq +
                       v.right.blocks[new.end_right].sum_enq
46:     new.sum_deq := v.left.blocks[new.end_left].sum_deq +
                       v.right.blocks[new.end_right].sum_deq
47:     num_enq := new.sum_enq − v.blocks[i − 1].sum_enq
48:     num_deq := new.sum_deq − v.blocks[i − 1].sum_deq
49:     if v = root then
50:         new.size := max(0, v.blocks[i − 1].size + num_enq − num_deq)
51:     end if
52:     if num_enq + num_deq = 0 then
53:         return null          ▷ no blocks need to be propagated to v
54:     else
55:         return new
56:     end if
57: end CreateBlock
58: void Advance(Node v, int h) ▷ set v.blocks[h].super and increment v.head from h to h + 1
59:     if v ≠ root then
60:         h_p := v.parent.head
61:         CAS(v.blocks[h].super, null, h_p)
62:     end if
63:     CAS(v.head, h, h+1)
64: end Advance
65: ⟨int, int⟩ IndexDequeue(Node v, int b, int i) ▷ return ⟨b′, i′⟩ such that ith dequeue in
66:     ▷ D(v.blocks[b]) is (i′)th dequeue of D(root.blocks[b′])
67:     ▷ Precondition: v.blocks[b] is not null, was propagated to root, and contains at least
68:     ▷ i dequeues
69:     if v = root then return ⟨b, i⟩
70:     else
71:         dir := (v.parent.left = v ? left : right)
72:         sup := v.blocks[b].super
73:         if b > v.parent.blocks[sup].end_dir then sup := sup + 1
74:         end if
75:         ▷ compute index i of dequeue in superblock
76:         i += v.blocks[b − 1].sum_deq − v.blocks[v.parent.blocks[sup − 1].end_dir].sum_deq
77:         if dir = right then
78:             i += v.blocks[v.parent.blocks[sup].end_left].sum_deq −
                     v.blocks[v.parent.blocks[sup − 1].end_left].sum_deq
79:         end if
80:         return IndexDequeue(v.parent, sup, i)
81:     end if
82: end IndexDequeue
83: element FindResponse(int b, int i) ▷ find response to ith dequeue in D(root.blocks[b])
84:     ▷ Precondition: 1 ≤ i ≤ |D(root.blocks[b])|
85:     num_enq := root.blocks[b].sum_enq − root.blocks[b − 1].sum_enq
86:     if root.blocks[b − 1].size + num_enq < i then
87:         return null                                 ▷ queue is empty when dequeue occurs
88:     else ▷ response is the eth enqueue in the root
89:         e := i + root.blocks[b-1].sum_enq - root.blocks[b-1].size
90:         ▷ compute enqueue's block using binary search
91:         find min b_e ≤ b with root.blocks[b_e].sum_enq ≥ e
92:         ▷ find rank of enqueue within its block
93:         i_e := e − root.blocks[b_e − 1].sum_enq
94:         return GetEnqueue(root, b_e, i_e)
95:     end if
96: end FindResponse
97: element GetEnqueue(Node v, int b, int i)    ▷ returns argument of ith enqueue in E(v.blocks[b])
98:     ▷ Preconditions: i ≥ 1 and v.blocks[b] is non-null and contains at least i enqueues
99:     if v is a leaf node then return v.blocks[b].element
100:    else
101:        sum_left := v.left.blocks[v.blocks[b].end_left].sum_enq
102:        ▷ sum_left is the number of enqueues in v.blocks[1..b] from v's left child
103:        prev_left := v.left.blocks[v.blocks[b − 1].end_left].sum_enq
104:        ▷ prev_left is the number of enqueues in v.blocks[1..b − 1] from v's left child
105:        prev_right := v.right.blocks[v.blocks[b − 1].end_right].sum_enq
106:        ▷ prev_right is the number of enqueues in v.blocks[1..b − 1] from v's right child
107:        if i ≤ sum_left − prev_left then            ▷ required enqueue is in v.left
108:            dir := left
109:        else                                        ▷ required enqueue is in v.right
110:            dir := right
111:            i := i − (sum_left − prev_left)
112:        end if
113:        ▷ Use binary search to find enqueue's block in v.dir and its rank within block
114:        find minimum b′ in range [v.blocks[b − 1].end_dir+1..v.blocks[b].end_dir] such that
                v.dir.blocks[b′].sum_enq ≥ i + prev_dir
115:        i′ := i − (v.dir.blocks[b′ − 1].sum_enq − prev_dir)
116:        return GetEnqueue(v.dir, b′, i′)
117:    end if
118: end GetEnqueue
```

Figure 4: Queue implementation.

To compute the response of the ith Dequeue in the bth block of the root, **FindResponse**(b, i) determines at line 86 if the queue is empty. If not, line 89 computes the rank e of the Enqueue whose argument is the Dequeue's response. A binary search on the sum_{enq} fields of $root.blocks$ finds the index b_e of the block that contains the eth enqueue. Since the enqueue is linearized before the dequeue, $b_e \leq b$. To find the left end of the range for the binary search for b_e, we can first do a doubling search [5], comparing e to the sum_{enq} fields at indices $b-1, b-2, b-4, b-8, \ldots$. Then, GetEnqueue traces down through the tree to find the required enqueue in a leaf.

GetEnqueue(v, b, i) returns the argument of the ith enqueue in the bth block B of Node v. It recursively finds the location of the

enqueue in each node along the path from v to a leaf, which stores the argument explicitly. GetEnqueue first determines which child of v contains the enqueue, and then finds the range of blocks within that child that are subblocks of B using information stored in B and the block that precedes B in v. GetEnqueue finds the exact subblock containing the enqueue using a binary search on the sum_{enq} field (line 114) and proceeds recursively down the tree.

4 PROOF OF CORRECTNESS

After proving some basic properties in Section 4.1, we show in Section 4.2 that a double refresh at each node suffices to propagate an operation to the root. In Section 4.3 we show GetEnqueue and IndexDequeue correctly navigate through the tree. Finally, we prove linearizability in Section 4.4.

4.1 Basic Properties

A Block object's fields, except for *super*, are immutable: they are written only when the block is created. Moreover, only a CAS at line 61 modifies *super* (from null to a non-null value), so it is changed only once. Similarly, only a CAS at line 35 modifies an element of a node's *blocks* array (from null to a non-null value), so blocks are permanently added to nodes. Only a CAS at line 63 can update a node's *head* field by incrementing it, which implies the following.

Observation 1. *For each node v, $v.head$ is non-decreasing over time.*

Observation 2. *Let R be an instance of* Refresh(v) *whose call to* CreateBlock *returns a non-*null *block. When R terminates, $v.head$ is strictly greater than the value R reads from it at line 25.*

Proof. After R's CAS at line 63, $v.head$ is no longer equal to the value h read at line 25. The claim follows from Observation 1. □

Now we show $v.blocks[v.head]$ is either the last non-null block or the first null block in node v.

Invariant 3. *For $0 \le i < v.head$, $v.blocks[i] \neq$ null. For $i > v.head$, $v.blocks[i] =$ null. If $v \neq root$, $v.blocks[i].super \neq$ null for $0 < i < v.head$.*

Proof. Initially, $v.head = 1$, $v.blocks[0] \neq$ null and $v.blocks[i] =$ null for $i > 0$, so the claims hold.

Assume the claims hold before a change to $v.blocks$, which can be made only by a successful CAS at line 35. The CAS changes $v.blocks[h]$ from null to a non-null value. Since $v.blocks[h]$ is null before the CAS, $v.head \le h$ by the hypothesis. Since h was read from $v.head$ earlier at line 25, the current value of $v.head$ is at least h by Observation 1. So, $v.head = h$ when the CAS occurs and a change to $v.blocks[v.head]$ preserves the invariant.

Now, assume the claim holds before a change to $v.head$, which can only be an increment from h to $h+1$ by a successful CAS at line 63 of Advance. For the first two claims, it suffices to show that $v.blocks[head] \neq$ null. Advance is called either at line 29 after testing that $v.blocks[h] \neq$ null at line 28, or at line 36 after the CAS at line 35 ensures $v.blocks[h] \neq$ null. For the third claim, observe that prior to incrementing $v.head$ to $i+1$ at line 63, the CAS at line 61 ensures that $v.blocks[i].super \neq$ null. □

It follows that blocks accessed by the Enqueue, Dequeue and CreateBlock routines are non-null.

The following two lemmas show that no operation appears in more than one block of the root.

Lemma 4. *If $b > 0$ and $v.blocks[b] \neq$ null, then*

$$v.blocks[b-1].end_{left} \le v.blocks[b].end_{left} \text{ and}$$
$$v.blocks[b-1].end_{right} \le v.blocks[b].end_{right}.$$

Proof. Let B be the block in $v.blocks[b]$. Before creating B at line 32, the Refresh that installed B read b from $v.head$ at line 25. At that time, $v.blocks[b-1]$ contained a block B', by Invariant 3. Thus, the CreateBlock$(v, b-1)$ that created B' terminated before the CreateBlock(v, b) that created B started. It follows from Observation 1 that the value that line 43 of CreateBlock$(v, b-1)$ stores in $B'.end_{left}$ is less than or equal to the value that line 43 of CreateBlock(v, b) stores in $B.end_{left}$. Similarly, the values stored in $B'.end_{right}$ and $B.end_{right}$ at line 44 satisfy the claim. □

Lemma 5. *If B and B' are two blocks in nodes at the same depth, their sets of subblocks are disjoint.*

Proof. We prove the lemma by reverse induction on the depth. If B and B' are in leaves, they have no subblocks, so the claim holds. Assume the claim holds for nodes at depth $d+1$ and let B and B' be two blocks in nodes at depth d. Consider the direct subblocks of B and B' defined by (3.3). If B and B' are in different nodes at depth d, then their direct subblocks are disjoint. If B and B' are in the same node, it follows from Lemma 4 that their direct subblocks are disjoint. Either way, their direct subblocks (at depth $d+1$) are disjoint, so the claim follows from the induction hypothesis. □

It follows that each block has at most one superblock. Moreover, we can now prove each operation is contained in at most one block of each node, and hence appears at most once in the linearization L.

Corollary 6. *For $i \neq j$, $v.blocks[i]$ and $v.blocks[j]$ cannot both contain the same operation.*

Proof. A block B contains the operations in B's subblocks in leaves of the tree. An operation by process P appears in just one block of P's leaf, so an operation cannot be in two different leaf blocks. By Lemma 5, $v.blocks[i]$ and $v.blocks[j]$ have no common subblocks, so the claim follows. □

The accuracy of the values stored in the sum_{enq} and sum_{deq} fields on lines 2, 6, 45 and 46 follows easily from the definition of subblocks. See the full version [34] for a detailed proof of Invariant 7.

Invariant 7. *If B is a block stored in $v.blocks[i]$, then*

$$B.sum_{enq} = |E(v.blocks[0]) \cdots E(v.blocks[i])| \text{ and}$$
$$B.sum_{deq} = |D(v.blocks[0]) \cdots D(v.blocks[i])|.$$

This allows us to prove that every block a Refresh installs contains at least one operation.

Corollary 8. *If a block B is in $v.blocks[i]$ where $i > 0$, then $E(B)$ and $D(B)$ are not both empty.*

Proof. The Refresh that installed B got B as the response to its call to CreateBlock on line 32. Thus, at line 52, $num_{enq} + num_{deq} \neq 0$. By Invariant 7, $num_{enq} = |E(B)|$ and $num_{deq} = |D(B)|$, so these sequences cannot both be empty. □

4.2 Propagating Operations to the Root

In the next two lemmas, we show two Refreshes suffice to propagate operations from a child to its parent. We say that node v *contains* an operation op if some block in $v.blocks$ contains op. Since blocks are permanently added to nodes, if v contains op at some time, v contains op at all later times too.

Lemma 9. *Let R be a call to* Refresh*(v) that performs a successful* CAS *on line 35 (or terminates at line 33). After that CAS (or termination, respectively), v contains all operations that v's children contained when R executed line 25.*

Proof. Suppose v's child (without loss of generality, $v.left$) contained an operation op when R executed line 25. Let i be the index such that the block $B = v.left.blocks[i]$ contains op. By Observation 1 and Lemma 4, the value of $childHead$ that R reads from $v.left.head$ in line 27 is at least i. If it is equal to i, R calls Advance at line 29, which ensures that $v.left.head > i$. Then, R calls CreateBlock(v, h) in line 32, where h is the value R reads at line 25. CreateBlock reads a value greater than i from $v.left.head$ at line 43. Thus, $new.end_{left} \geq i$. We consider two cases.

Suppose R's call to CreateBlock returns the new block B' and R's CAS at line 35 installs B' in $v.blocks$. Then, B is a subblock of some block in v, since $B'.end_{left}$ is greater than or equal to B's index i in $v.left.blocks$. Hence v contains op, as required.

Now suppose R's call to CreateBlock returns null, causing R to terminate at line 33. Intuitively, since there are no operations in v's children to promote, op is already in v. We formalize this intuition. The value computed at line 45 is

$$\begin{aligned} num_{enq} &= v.left.blocks[new.end_{left}].sum_{enq} \\ &\quad + v.right.blocks[new.end_{right}].sum_{enq} \\ &\quad - v.blocks[h-1].sum_{enq} \\ &= v.left.blocks[new.end_{left}].sum_{enq} \\ &\quad + v.right.blocks[new.end_{right}].sum_{enq} \\ &\quad - v.left.blocks[v.blocks[h-1].end_{left}].sum_{enq} \\ &\quad - v.right.blocks[v.blocks[h-1].end_{right}].sum_{enq}. \end{aligned}$$

It follows from Invariant 7 that num_{enq} is the total number of enqueues in $v.left.blocks[v.blocks[h-1].end_{left}+1..new.end_{left}]$ and $v.right.blocks[v.blocks[h-1].end_{right}+1..new.end_{right}]$. Similarly, num_{deq} is the total number of dequeues contained in these blocks. Since $num_{\text{enq}} + num_{\text{deq}} = 0$ at line 52, these blocks contain no operations. By Corollary 8, this means the ranges of blocks are empty, so that $v.blocks[h-1].end_{left} \geq new.end_{left} \geq i$. Hence, B is already a subblock of some block in v, so v contains op. □

We now show a double Refresh propagates blocks as required.

Lemma 10. *Consider two consecutive terminating calls R_1, R_2 to* Refresh*(v) by the same process. All operations contained v's children when R_1 begins are contained in v when R_2 terminates.*

Proof. If either R_1 or R_2 performs a successful CAS at line 35 or terminates at line 33, the claim follows from Lemma 9. So suppose both R_1 and R_2 perform a failed CAS at line 35. Let h_1 and h_2 be the values R_1 and R_2 read from $v.head$ at line 25. By Observation 2, $h_2 > h_1$. By Lemma 4, $v.blocks[h_2] = \text{null}$ when R_1 executes line 25. Since R_2 fails its CAS on $v.blocks[h_2]$, some other Refresh R_3 must have done a successful CAS on $v.blocks[h_2]$ before R_2's CAS. R_3 must have executed line 25 after R_1, since R_3 read the value h_2 from $v.head$ and the value of $v.head$ is non-decreasing, by Observation 1. Thus, all operations contained in v's children when R_1 begins are also contained in v's children when R_3 later executes line 25. By Lemma 9, these operations are contained in v when R_3 performs its successful CAS, which is before R_2's failed CAS. □

Lemma 11. *When an* Append*(B) terminates, B's operation is contained in exactly one block in each node along the path from the process's leaf to the root.*

Proof. Append adds B to the process's leaf and calls Propagate, which does a double Refresh on each internal node on the path P from the leaf to the root. By Lemma 10, this ensures a block in each node on P contains B's operation. There is at most one such block in each node, by Corollary 6. □

4.3 Correctness of GetEnqueue and IndexDequeue

See the full version [34] for detailed proofs for this section.

We first show the *super* field is accurate, since IndexDequeue uses it to trace superblocks up the tree. This is proved by showing that the *super* field of a block B in node v is read from $v.parent$'s *head* field close to the time that B's superblock B_s is installed in the parent node. On one hand, $B.super$ is written before B_s is installed: Advance writes $B.super$ before advancing $v.head$ past B's index, which must happen before the CreateBlock that creates B_s gets the value of $B_s.end_{left}$ or $B_s.end_{right}$. On the other hand, $B.super$ cannot be written too long before B_s is installed: $B.super$ is written after B is installed, and Lemma 9 ensures that B is propagated to the parent soon after.

Lemma 12. *Let $B = v.blocks[b]$. If $v.parent.blocks[s]$ is the superblock of B then $s - 1 \leq B.super \leq s$.*

To show GetEnqueue and IndexDequeue work correctly, we just check that they correctly compute the index of the required block and the operation's rank within the block. For IndexDequeue, we use Lemma 12 each time IndexDequeue goes one step up the tree.

Lemma 13. *If $v.blocks[b]$ has been propagated to the root and $1 \leq i \leq |D(v.blocks[b])|$, then* IndexDequeue*$(v, b, i)$ returns $\langle b', i' \rangle$ such that the ith dequeue in $D(v.blocks[b])$ is the (i')th dequeue of $D(root.blocks[b'])$.*

Lemma 14. *If $1 \leq i \leq |E(v.blocks[b])|$ then* getEnqueue*(v, b, i) returns the argument of the ith enqueue in $E(v.blocks[b])$.*

4.4 Linearizability

We show that the linearization ordering L defined in (3.2) is a legal permutation of a subset of the operations in the execution, i.e., that it includes all operations that terminate and if one operation op_1 terminates before another operation op_2 begins, then op_2 does not precede op_1 in L. We also show the result each dequeue returns is the same as in the sequential execution L.

Lemma 15. *L is a legal linearization ordering.*

Proof. By Corollary 6, L is a permutation of a subset of the operations in the execution. By Lemma 11, each terminating operation is propagated to the root before it terminates, so it appears in L. Also, if op_1 terminates before op_2 begins, then op_1 is propagated to the root before op_2 begins, so op_1 appears before op_2 in L. □

A simple proof (in the full version [34]) shows that *size* fields are computed correctly.

Lemma 16. *If the operations of root.blocks[0..b] are applied sequentially in the order of L on an initially empty queue, the resulting queue has root.blocks[b].size elements.*

Next, we show operations return the same response as they would in the sequential execution L.

Lemma 17. *Each terminating dequeue returns the response it would in the sequential execution L.*

Proof. If a dequeue D terminates, it is contained in some block in the root, by Lemma 11. By Lemma 13, D's call to IndexDequeue on line 8 returns a pair $\langle b, i\rangle$ such that D is the ith dequeue in the block $B = root.blocks[b]$. D then calls FindResponse(b, i) on line 9. By Lemma 16, the queue contains $root.blocks[b-1].size$ elements after the operations in $root.blocks[1..b-1]$ are performed sequentially in the order given by L. By Invariant 7, the value of num_{enq} computed on line 85 is the number of enqueues in B. Since the enqueues in block B precede the dequeues, the queue is empty when the ith dequeue of B occurs if $root.blocks[b-1].size + num_{enq} < i$. So D returns null on line 87 if and only if it would do so in the sequential execution L. Otherwise, the size of the queue after doing the operations in $root.blocks[0..b-1]$ in the sequential execution L is $root.blocks[b-1].sum_{enq}$ minus the number of non-null dequeues in that prefix of L. Hence, line 89 sets e to the rank of D among all the non-null dequeues in L. Thus, in the sequential execution L, D returns the value enqueued by the eth enqueue in L. By Invariant 7, this enqueue is the i_eth enqueue in $E(root.blocks[b_e])$, where b_e and i_e are the values D computes on line 91 and 93. By Lemma 14, the call to GetEnqueue returns the argument of the required enqueue. □

Combining Lemma 15 and Lemma 17 provides our main result.

Theorem 18. *The queue implementation is linearizable.*

5 ANALYSIS

We now analyze the number of steps and the number of CAS instructions performed by operations.

Proposition 19. *Each* Enqueue *or* Dequeue *operation performs* $O(\log p)$ CAS *instructions.*

Proof. An operation invokes Refresh at most twice at each of the $\lceil \log_2 p \rceil$ levels of the tree. A Refresh does at most 5 CAS steps: one in line 35 and two during each Advance in line 29 or 36. □

Lemma 20. *The search that* FindResponse(b, i) *does at line 91 to find the index b_e of the block in the root containing the eth enqueue takes $O(\log(root.blocks[b_e].size + root.blocks[b-1].size))$ steps.*

Proof. Let the blocks in the root be $B_1, \ldots, B_\ell$. The doubling search for b_e takes $O(\log(b - b_e))$ steps, so we prove $b - b_e \le 2 \cdot B_{b_e}.size + B_{b-1}.size + 1$. If $b \le b_e + 1$, then this is trivial, so assume $b > b_e + 1$. As shown in Lemma 17, the dequeue that calls FindResponse is in B_b and is supposed to return an enqueue in B_{b_e}. Thus, there can be at most $B_{b_e}.size$ dequeues in $D(B_{b_e+1}) \cdots D(B_{b-1})$; otherwise in the sequential execution L, all elements enqueued before the end of $E(B_{b_e})$ would be dequeued before $D(B_b)$. Furthermore, by Lemma 16, the size of the queue after the prefix of L corresponding to $B_1, \ldots, B_{b-1}$ is $B_{b-1}.size \ge B_{b_e}.size + |E(B_{b_e+1}) \cdots E(B_{b-1})| - |D(B_{b_e+1}) \cdots D(B_{b-1})|$. Thus, $|E(B_{b_e+1}) \cdots E(B_{b-1})| \le B_{b-1}.size + |D(B_{b_e+1}) \cdots D(B_{b-1})| \le B_{b-1}.size + B_{b_e}.size$. So, the total number of operations in $B_{b_e+1}, \ldots, B_{b-1}$ is at most $B_{b-1}.size + 2 \cdot B_{b_e}.size$. Each of these $b - 1 - b_e$ blocks contains at least one operation, by Corollary 8. So, $b - 1 - b_e \le B_{b-1}.size + 2 \cdot B_{b_e}.size$. □

The following lemma helps bound the time for GetEnqueue.

Lemma 21. *Each block B in each node contains at most one operation of each process. If c is the execution's maximum point contention, B has at most c direct subblocks.*

Proof. Suppose B contains an operation of process p. Let op be the earliest operation by p contained in B. When op terminates, op is contained in B by Lemma 11. Since B is created before those operations are invoked, B cannot contain any later operations by p.

Let t be the earliest termination of any operation contained in B. By Lemma 11, B is created before t, so all operations contained in B are invoked before t. Thus, all are running concurrently at t, so B contains at most c operations. By definition, the direct subblocks of B contain these c operations, and each operation is contained in exactly one of these subblocks, by Lemma 5. By Corollary 8, each direct subblock of B contains at least one operation, so B has at most c direct subblocks. □

We now bound step complexity in terms of the number of processes p, the maximum contention $c \le p$, and the size of the queue.

Theorem 22. *Each* Enqueue *and null Dequeue takes $O(\log p)$ steps and each non-null* Dequeue *takes $O(\log p \log c + \log q_e + \log q_d)$ steps, where q_d is the size of the queue when the* Dequeue *is linearized and q_e is the size of the queue when the* Enqueue *of the value returned is linearized.*

Proof. An Enqueue or null Dequeue creates a block, appends it to the process's leaf and propagates it to the root. The Propagate does $O(1)$ steps at each node on the path from the process's leaf to the root. A null Dequeue additionally calls IndexDequeue, which also does $O(1)$ steps at each node on this path. So, the total number of steps for either type of operation is $O(\log p)$.

A non-null Dequeue must also search at line 91 and call GetEnqueue at line 94. By Lemma 20, the doubling search takes $O(\log(q_e + q_d + p))$ steps, since the size of the queue can change by at most p within one block (by Lemma 21). GetEnqueue does a binary search within each node on a path from the root to a leaf. Each node v's search is within the subblocks of one block in v's parent. By Lemma 21, each such search takes $O(\log c)$ steps, for a total of $O(\log p \log c)$ steps. □

Corollary 23. *The queue implementation is wait-free.*

6 BOUNDING SPACE USAGE

Operations remain in the *blocks* arrays forever. This uses space proportional to the number of enqueues that have been invoked. Now, we modify the implementation to remove blocks that are no longer needed, so that space usage is polynomial in p and q, while (amortized) step complexity is still polylogarithmic. For lack of space, details are in [34]. We replace the *blocks* array in each node by a red-black tree (RBT) that stores the blocks. Each block has an additional *index* field that represents its position within the original *blocks* array, and blocks in a RBT are sorted by *index*. The attempt to install a new block in $blocks[i]$ on line 35 is replaced by an attempt to insert a new block with index i into the RBT. Accessing the block in $blocks[i]$ is replaced by searching the RBT for the index i. The binary searches for a block in line 91 and 114 can simply search the RBT using the sum_{enq} field, since the RBT is also sorted with respect to this field, by Invariant 7.

Known lock-free search trees have step complexity that includes a term linear in p [9, 22]. However, we do not require all the standard search tree operations. Instead of a standard insertion, we allow a Refresh's insertion to fail if another concurrent Refresh succeeds in inserting a block, just as the CAS on line 35 can fail if a concurrent Refresh does a successful CAS. Moreover, the insertion should succeed only if the newly inserted block has a larger index than any other block in the RBT. Thus, we can use a particularly simple concurrent RBT implementation. A sequential RBT can be made persistent using the classic node-copying technique of Driscoll et al. [8]: all RBT nodes are immutable, and operations on the RBT make a new copy of each RBT node x that must be modified, as well as each RBT node along the path from the RBT's root to x. The RBT reachable from the new copy of the root is the result of applying the RBT operation. This only adds a constant factor to the running time of any routine designed for a (sequential) RBT. Once a process has performed an update to the RBT representing the blocks of a node v in the ordering tree, it uses a CAS to swing v's pointer from the previous RBT root to the new RBT root. A search in the RBT can simply read the pointer to the RBT root and perform a standard sequential search on it. Bashari and Woelfel [3] used persistent RBTs in a similar way for a snapshot data structure.

To prevent RBTs from growing without bound, we would like to discard blocks that are no longer needed. Ensuring the size of the RBT is polynomial in p and q will also keep the running time of our operations polylogarithmic. Blocks should be kept if they contain operations still in progress. Moreover, a block containing an Enqueue(x) must be kept until x is dequeued.

To maintain good amortized time, we periodically do a garbage collection (GC) phase. If a Refresh on a node adds a block whose *index* is a multiple of $G = p^2\lceil \log p \rceil$, it does GC to remove obsolete blocks from the node's RBT. To determine which blocks can be thrown away, we use a global array $last[1..p]$ where each process writes the index of the last block in the root containing a null dequeue or an enqueue whose element it dequeued. To perform GC, a process reads $last[1..p]$ and finds the maximum entry m. Then, it helps complete every other process's pending dequeue by computing the dequeue's response and writing it in the block in the leaf that represents the dequeue. Once this helping is complete, it follows from the FIFO property of the queue that elements enqueued in $root.blocks[1..m-1]$ have all been dequeued, so GC can discard all subblocks of those. Fortunately, there is an RBT Split operation that can remove these obsolete blocks from an RBT in logarithmic time [42, Sec. 4.2].

An operation op's search of a RBT may fail to find the required block B that has been removed by another process's GC phase. If op is a dequeue, op must have been helped before B was discarded, so op can simply read its response from its own leaf. If op is an enqueue, it can simply terminate.

Our GC scheme ensures each RBT has $O(q + p^2 \log p)$ blocks, so RBT operations take $O(\log(p+q))$ time. Excluding GC, an operation does $O(1)$ operations on RBTs at each level of the tree for a total of $O(\log p \log(p+q))$ steps. A GC phase takes $O(p \log p \log(p+q))$ steps to help complete all pending operations. If all processes carry out this GC phase, it takes a total of $O(p^2 \log p \log(p+q)) = O(G \log(p+q))$ steps. Since there are at least G operations between two GC phases, each node contributes $O(\log(p+q))$ steps to each operation in an amortized sense. Adding up over all nodes an operation may have to do GC on, an operation spends $O(\log p \log(p+q))$ steps doing GC in an amortized sense. So the total amortized step complexity is $O(\log p \log(p+q))$ per operation.

7 FUTURE DIRECTIONS

Our focus was on optimizing step complexity for worst-case executions. However, our queue has a higher cost than the MS-queue in the best case (when an operation runs by itself). Perhaps our queue could be made adaptive by having an operation capture a starting node in the ordering tree (as in [1]) rather than starting at a statically assigned leaf. A possible application of our queue might be to use it as the slow path in the fast-path slow-path methodology [25] to get a queue that has good performance in practice while also having good worst-case step complexity.

A gap remains between our queue, which takes $O(\log^2 p + \log q)$ steps per operation, and Attiya and Fouren's $\Omega(\min(c, \log\log p))$ lower bound [2]. It would be interesting to determine how the true step complexity of lock-free queues (or, more generally, bags) depends on p. Since a queue is also a bag, our queue is the first lock-free bag we know of that has polylogarithmic step complexity.

We believe the approach used here to implement a lock-free queue could be applied to obtain other lock-free data structures with a polylogarithmic step complexity. For example, we can easily adapt our routines to implement a 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 ith element in the sequence, and Index(e) to compute the position of element e in the sequence. We are investigating whether a similar approach can be used for stacks, deques or even priority queues.

ACKNOWLEDGMENTS

We thank Franck van Breugel and the anonymous referees for their helpful comments. Funding was provided by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Yehuda Afek, Dalia Dauber, and Dan Touitou. 1995. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*. 538–547. https://doi.org/10.1145/225058.225271

[2] Hagit Attiya and Arie Fouren. 2017. Lower Bounds on the Amortized Time Complexity of Shared Objects. In *21st International Conference on Principles of Distributed Systems (LIPIcs, Vol. 95)*. 16:1–16:18. https://doi.org/10.4230/LIPIcs.OPODIS.2017.16

[3] Benyamin Bashari and Philipp Woelfel. 2021. An Efficient Adaptive Partial Snapshot Implementation. In *Proc. ACM Symposium on Principles of Distributed Computing*. 545–555. https://doi.org/10.1145/3465084.3467939

[4] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. 2021. Space and Time Bounded Multiversion Garbage Collection. In *Proc. 35th International Symposium on Distributed Computing (LIPIcs, Vol. 209)*. 12:1–12:20. https://doi.org/10.4230/LIPIcs.DISC.2021.12

[5] Jon Louis Bentley and Andrew Chi-Chih Yao. 1976. An almost optimal algorithm for unbounded searching. *Inform. Process. Lett.* 5, 3 (1976), 82–87. https://doi.org/10.1016/0020-0190(76)90071-5

[6] Robert Colvin and Lindsay Groves. 2005. Formal Verification of an Array-Based Nonblocking Queue. In *10th International Conference on Engineering of Complex Computer Systems*. IEEE, 507–516. https://doi.org/10.1109/ICECCS.2005.49

[7] Matei David. 2004. A Single-Enqueuer Wait-Free Queue Implementation. In *Proc. 18th International Conference on Distributed Computing (LNCS, Vol. 3274)*. Springer, 132–143. https://doi.org/10.1007/978-3-540-30186-8_10

[8] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making Data Structures Persistent. *J. Comput. System Sci.* 38, 1 (Feb. 1989), 86–124. https://doi.org/10.1016/0022-0000(89)90034-2

[9] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. 2014. The Amortized Complexity of Non-blocking Binary Search Trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*. 332–340. https://doi.org/10.1007/978-3-642-25873-2_15

[10] Faith Ellen, Vijaya Ramachandran, and Philipp Woelfel. 2012. Efficient Fetch-and-Increment. In *Proc. International Symposium on Distributed Computing (LNCS, Vol. 7611)*. Springer, 16–30. https://doi.org/10.1007/978-3-642-33651-5_2

[11] Faith Ellen and Philipp Woelfel. 2013. An Optimal Implementation of Fetch-and-Increment. In *Proc. 27th International Symposium on Distributed Computing (LNCS, Vol. 8205)*. Springer, 284–298. https://doi.org/10.1007/978-3-642-41527-2_20

[12] Panagiota Fatourou and Nikolaos D. Kallimanis. 2014. Highly-Efficient Wait-Free Synchronization. *Theory of Computing Systems* 55, 3 (2014), 475–520. https://doi.org/10.1007/s00224-013-9491-y

[13] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free linked lists and skip lists. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing*. 50–59. https://doi.org/10.1145/1011767.1011776

[14] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. 2010. Cache-Aware Lock-Free Queues for Multiple Producers/Consumers and Weak Memory Consistency. In *Proc. 14th International Conference on Principles of Distributed Systems (LNCS, Vol. 6490)*. Springer, 302–317. https://doi.org/10.1007/978-3-642-17653-1_23

[15] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (1991), 124–149. https://doi.org/10.1145/114005.102808

[16] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[17] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The Baskets Queue. In *Proc. 11th International Conference on Principles of Distributed Systems (LNCS, Vol. 4878)*. Springer, 401–414. https://doi.org/10.1007/978-3-540-77096-1_29

[18] Prasad Jayanti. 1998. A Time Complexity Lower Bound for Randomized Implementations of Some Shared Objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*. 201–210. https://doi.org/10.1145/277697.277735

[19] Prasad Jayanti and Srdjan Petrovic. 2005. Logarithmic-Time Single Deleter, Multiple Inserter Wait-Free Queues and Stacks. In *Foundations of Software Technology and Theoretical Computer Science (LNCS, Vol. 3821)*. Springer, 408–419. https://doi.org/10.1007/11590156_33

[20] Colette Johnen, Adnane Khattabi, and Alessia Milani. 2023. Efficient Wait-Free Queue Algorithms with Multiple Enqueuers and Multiple Dequeuers. In *Proc. 26th International Conference on Principles of Distributed Systems (LIPIcs, Vol. 253)*. 4:1–4:19. https://doi.org/10.4230/LIPIcs.OPODIS.2022.4

[21] Pankaj Khanchandani and Roger Wattenhofer. 2018. On the Importance of Synchronization Primitives with Low Consensus Numbers. In *Proc. 19th International Conference on Distributed Computing and Networking*. 18:1–18:10. https://doi.org/10.1145/3154273.3154306

[22] Jeremy Ko. 2020. The amortized analysis of a non-blocking chromatic tree. *Theoretical Computer Science* 840 (Nov. 2020), 59–121. https://doi.org/10.1016/j.tcs.2020.07.007

[23] Alex Kogan and Maurice Herlihy. 2014. The future(s) of shared data structures. In *Proc. ACM Symposium on Principles of Distributed Computing*. 30–39. https://doi.org/10.1145/2611462.2611496

[24] Alex Kogan and Erez Petrank. 2011. Wait-free queues with multiple enqueuers and dequeuers. In *Proc. 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 223–234. https://doi.org/10.1145/1941553.1941585

[25] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-Free Data Structures. *ACM SIGPLAN Not.* 47, 8 (2012), 141–150. https://doi.org/10.1145/2370036.2145835

[26] Nikita Koval, Dan Alistarh, and Roman Elizarov. 2023. Fast and Scalable Channels in Kotlin Coroutines. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*. 107–118. https://doi.org/10.1145/3572848.3577481

[27] Edya Ladan-Mozes and Nir Shavit. 2008. An optimistic approach to lock-free FIFO queues. *Distributed Computing* 20, 5 (2008), 323–341. https://doi.org/10.1007/s00446-007-0050-0

[28] Henry Massalin and Carlton Pu. 1991. *A Lock-Free Multiprocessor OS Kernel*. Technical Report CUCS-005-91. Department of Computer Science, Columbia University.

[29] Maged M. Michael and Michael L. Scott. 1998. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *J. Parallel and Distrib. Comput.* 51, 1 (1998), 1–26. https://doi.org/10.1006/jpdc.1998.1446

[30] Gal Milman-Sela, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2022. BQ: A Lock-Free Queue with Batching. *ACM Trans. Parallel Comput.* 9, 1, Article 5 (March 2022), 49 pages. https://doi.org/10.1145/3512757

[31] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using elimination to implement scalable and lock-free FIFO queues. In *Proc. 17th ACM Symposium on Parallelism in Algorithms and Architectures*. 253–262. https://doi.org/10.1145/1073970.1074013

[32] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 103–112. https://doi.org/10.1145/2442516.2442527

[33] Hossein Naderibeni. 2022. *A Wait-free Queue with Poly-logarithmic Worst-case Step Complexity*. Master's thesis. York University, Toronto, Canada. Available from https://yorkspace.library.yorku.ca/xmlui/handle/10315/40975.

[34] Hossein Naderibeni and Eric Ruppert. 2023. A Wait-free Queue with Polylogarithmic Step Complexity. (2023). Full version of this paper available from https://arxiv.org/abs/2305.07229.

[35] Ruslan Nikolaev. 2019. A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue. In *Proc. 33rd International Symposium on Distributed Computing (LIPIcs, Vol. 146)*. 28:1–28:16. https://doi.org/10.4230/LIPIcs.DISC.2019.28

[36] Ruslan Nikolaev and Binoy Ravindran. 2022. wCQ: A Fast Wait-Free Queue with Bounded Memory Usage. In *Proc. 34th ACM Symposium on Parallelism in Algorithms and Architectures* (Philadelphia, PA, USA). 307–319. https://doi.org/10.1145/3490148.3538572

[37] Pedro Ramalhete and Andreia Correia. 2017. Poster: A Wait-Free Queue with Wait-Free Memory Reclamation. *ACM SIGPLAN Not.* 52, 8 (Jan. 2017), 453–454. https://doi.org/10.1145/3155284.3019022

[38] Raed Romanov and Nikita Koval. 2023. The state-of-the-art LCRQ concurrent queue algorithm does NOT require CAS2. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*. 14–26. https://doi.org/10.1145/3572848.3577485

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

[40] Niloufar Shafiei. 2009. Non-blocking Array-Based Algorithms for Stacks and Queues. In *Proc. 10th International Conference on Distributed Computing and Networking (LNCS, Vol. 5408)*. Springer, 55–66. https://doi.org/10.1007/978-3-540-92295-7_10

[41] Niloufar Shafiei. 2015. Non-blocking doubly-linked lists with good amortized complexity. In *Proc. 19th International Conference on Principles of Distributed Systems (LIPIcs, Vol. 46)*. 35:1–35:17. https://doi.org/10.4230/LIPIcs.OPODIS.2015.35

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

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

[44] Philippas Tsigas and Yi Zhang. 2001. 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*. 134–143. https://doi.org/10.1145/378580.378611

[45] Chaoran Yang and John M. Mellor-Crummey. 2016. A wait-free queue as fast as fetch-and-add. In *Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 16:1–16:13. https://doi.org/10.1145/2851141.2851168