A Wait-free Queue with Polylogarithmic Step Complexity

Hossein Naderibeni[1] and Eric Ruppert[1*]

[1]York University, Toronto, Canada.

*Corresponding author(s). E-mail(s): ruppert@eecs.yorku.ca;

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.

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 [33], memory management (e.g.,[5]), synchronization [30],and sharing resources or tasks. We focus on shared queues that are *linearizable* [18], 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. We study the *amortized step complexity* per operation, which is measured by taking the maximum, over all possible executions, of the number of steps in the execution divided by the total number of enqueue and dequeue operations in the execution.

The lock-free *MS-queue* of Michael and Scott [34] 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. Numerous papers have suggested modifications to the MS-queue [19, 27, 28, 31, 35, 36, 42], 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 [37] called this the *CAS retry problem*. The same problem occurs in array-based implementations of queues [7, 15, 45, 49]. Solutions that tried to sidestep this problem using fetch&increment [37, 39, 40, 50] 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 [44]. For example, lock-free lists [14, 46], stacks [48] and search trees [10] 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 [3] 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 non-deterministically). 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 also *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. Since a queue is also a bag, our queue is the first lock-free bag with polylogarithmic step complexity.

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.

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, 21]), 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 reach the root in batches and are linearized in the order they reach the root. Since a batch can have up to p operations, explicitly recording the list of operations composing a batch or applying them one-by-one to the queue would be too costly. Instead, we use a novel implicit representation of batches of queue operations that allows us to quickly merge two batches from the children of a node, and quickly access any operation in a batch. A preliminary version of this work appeared in [38].

2 Related Work

List-based Queues.

The MS-queue [34] is a lock-free queue that has stood the test of time. The standard Java Concurrency Package includes a version of it. See [34] 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. [36] 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 [19] 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 [27] 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. [35] 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 [31] 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 [28] used Herlihy's helping technique [17] to make the MS-queue wait-free. Then, they introduced the fast-path slow-path methodology [29] 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 [42] added a different helping mechanism to the MS-queue. Although these approaches can perform well in practice, the amortized step complexity remains $\Omega(p)$.

Haas [16] described a queue implementation based on timestamping. To enqueue an item, a process adds the item, together with an associated timestamp, to the process's own single-enqueuer multi-dequeuer queue, which is implemented as a linked list. Items added by concurrent enqueues may get the same timestamp. Although enqueues are very efficient, dequeues look at all p lists to find and return an item with the oldest timestamp. This is compounded by the fact that dequeues may compete to claim the same item using a CAS, resulting in a CAS retry problem. As a result, the amortized step complexity for dequeue operations is $\Omega(p^2)$.

Array-Based Queues.

Arrays can be used to implement queues with bounded capacity [7, 41, 45, 49]. Dequeues and enqueues update indices of the front and back elements using CAS instructions. Gidenstam, Sundell, and Tsigas [15] 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 [37] 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 [43].) 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 [29]. Other array-based queues [39, 40, 50] 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 [17]. Afek, Dauber, and Touitou's universal construction [1] introduced the technique where a process must climb a binary tree while helping any other processes it sees along the way to reach the root, and this technique is the basis of our queue implementation. Jayanti [20] observed that their construction can be modified to use $O(\log p)$ steps per operation, assuming that words can store $\Omega(p \log p)$ bits. However, if more reasonably-sized $O(\log p)$-bit words words are used, the construction would take $\Omega(p \log p)$ steps per operation. There are two obstacles to making this construction more efficient: it is expensive to manipulate lists of up to p operations that must be propagated along a path up to the root, and when a batch of up to p operations reaches the root, it is expensive to perform all of them on the implemented data structure. In our work, we devise a novel way of doing this implicitly for batches of operations on a queue.

Fatourou and Kallimanis [13] used their own universal construction based on fetch&add and LL/SC instructions to implement a queue, but its step complexity is also $\Omega(p)$ when words are of reasonable size.

Restricted Queues.

David gave the first sublinear-time queue [8], 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 [21], but only for a single dequeuer. Like our queue, theirs uses the cooperative tree-climbing technique of Afek, Dauber and Touitou [1]. Concurrently with our work, which first appeared in [38], Johnen, Khattabi and Milani [23] built on [21] 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.

Li [32] implemented a non-blocking linearizable queue using only the weak primitives test&set, fetch&add and swap. This implementation's amortized step complexity is not bounded as a function of the number of processes and the size of the queue. (It depends on the total number of enqueues in the execution.)

Khanchandani and Wattenhofer [25] gave a wait-free queue with $O(\sqrt{p})$ step complexity using some strong 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 to argue that their primitives can be more efficient than CAS since previous CAS-based queues all required $\Omega(p)$ step complexity. Our new implementation undermines that argument.

Fetch&Increment Objects.

Ellen, Ramachandran and Woelfel [11] 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 [12] then gave an improved implementation of fetch&increment with better step complexity.

Lower Bounds.

As mentioned in the introduction, it follows from Attiya and Fouren's lower bound on bag data structures [3] that the amortized step complexity of operations on a queue is $\Omega(\min(c, \log\log p))$, where c is contention. Subsequently, Jayanti, Tarjan and Boix-Adserà [22] proved a $\Omega(\log c)$ lower bound on the amortized step complexity of queue operations. The combination of these lower bounds is shown schematically in Figure 1. If we are interested in the complexity only as a function of p, the second result gives us an $\Omega(\log p)$ lower bound, since c can be as high as p.

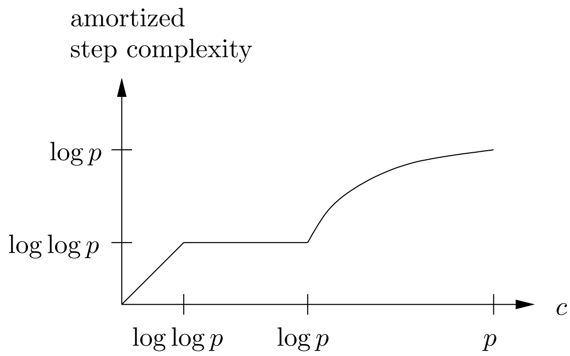

Fig. 1: Known lower bounds on amortized step complexity of queue operations when contention is c in a system of p processes.

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 2 for an example. For ease of presentation, we use an infinite array of blocks in each node in this section. Then, in Section 6, we describe 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 performs the following three steps. 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 [21], we call this three-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.

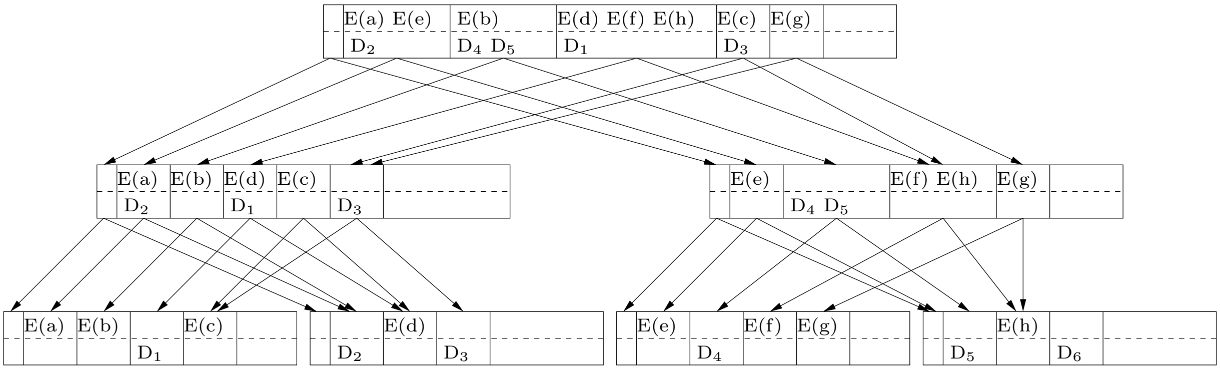

Fig. 2: An example ordering tree with four processes. E(x) denotes an Enqueue(x) operation and D_1 to D_6 denote Dequeue operations. We show explicitly the enqueue sequence and dequeue sequence represented by each block in the *blocks* arrays of the seven nodes. The leftmost element of each *blocks* 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 D_6 is still propagating towards the root. The linearization order for this tree is E(a) E(e) D_2 | E(b) D_4 D_5 | E(d) E(f) E(h) D_1 | E(c) D_3 | E(g), where vertical bars indicate boundaries of blocks in the root.

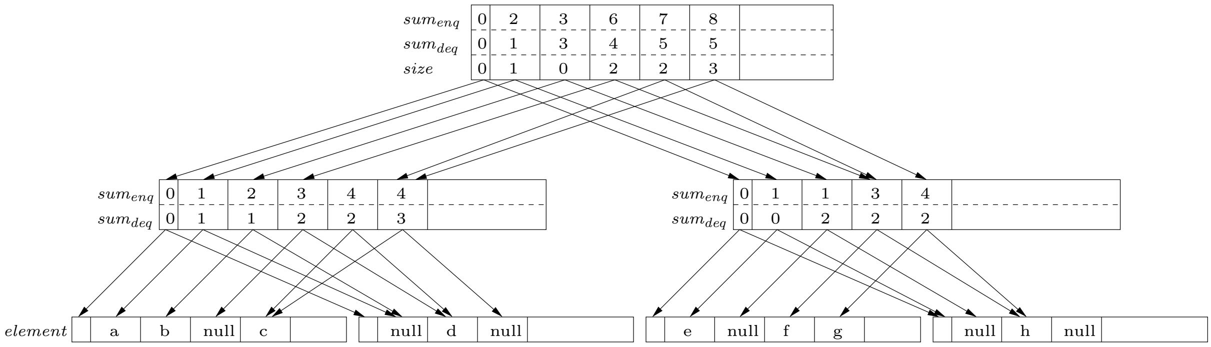

Fig. 3: The actual, implicit representation of the tree shown in Figure 2. The leaf blocks simply show the *element* field. Internal blocks show the sum_{enq} and sum_{deq} fields, and end_{left} and end_{right} fields are shown using arrows as in Figure 2. Root blocks also have the additional *size* field. The *super* field is not shown.

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 within 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

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

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

When linearizing operations within one of the root's blocks, we choose to put the 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 Representation of Blocks

In this section, we describe how to represent blocks so that tasks (T1) to (T4) can be done efficiently. 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.

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 3 for an example showing how the tree in Figure 2 is actually represented, and Figure 4 for the definitions of the fields of blocks and nodes.

▸ Node

- Node *left*, *right*, *parent* ▷ tree pointers initialized when creating the tree
- Block$[0..\infty]$ *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} ▷ # 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 end of this block

Fig. 4: Objects used in the ordering tree data structure.

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 the 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 of the sum_{enq} fields of 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 navigating upwards in the ordering tree for 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

$$B.sum_{enq} = v.left.blocks[B.end_{left}].sum_{enq} + v.right.blocks[B.end_{right}].sum_{enq}.$$

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:

$$B.size = \max(0, B'.size + (B.sum_{enq} - B'.sum_{enq}) - (B.sum_{deq} - B'.sum_{deq})).$$

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 Figures 5–7.

An **Enqueue**(e) appends a block to the process's leaf. The block's *element* field is e to indicate it represents an Enqueue(e) operation. It suffices to propagate the operation to the root and

```
▷ Shared variable
  • Node root                                  ▷ root of ordering tree
▷ Thread-local variable
  • Node leaf                                  ▷ process's leaf in the ordering tree

void Enqueue(Object e)
    let B be a new Block with fields element := e, sum_enq := leaf.blocks[leaf.head − 1].sum_enq + 1,
        sum_deq := leaf.blocks[leaf.head − 1].sum_deq
    Append(B)
end Enqueue

Object Dequeue()
    let B be a new Block with fields element := null, sum_enq := leaf.blocks[leaf.head − 1].sum_enq,
        sum_deq := leaf.blocks[leaf.head − 1].sum_deq + 1
    Append(B)
    ⟨b, i⟩ := IndexDequeue(leaf, leaf.head − 1, 1)
    return FindResponse(b, i)
end Dequeue

void Append(Block B)                           ▷ append block to leaf and propagate to root
    leaf.blocks[leaf.head] := B
    leaf.head := leaf.head + 1
    Propagate(leaf.parent)
end Append

void Propagate(Node v)                         ▷ propagate blocks from v's children to root
    if not Refresh(v) then                     ▷ double refresh
        Refresh(v)
    end if
    if v ≠ root then                           ▷ recurse up tree
        Propagate(v.parent)
    end if
end Propagate

boolean Refresh(Node v)                        ▷ try to append a new block B to v.blocks
    h := v.head
    for each dir in {left, right} do
        childHead := v.dir.head
        if v.dir.blocks[childHead] ≠ null then
            Advance(v.dir, childHead)
        end if
    end for
    B := CreateBlock(v, h)
    if B = null then return true
    else
        result := CAS(v.blocks[h], null, B)
        Advance(v, h)
        return result
    end if
end Refresh
```

Fig. 5: Queue implementation's main routines.

```
Block CreateBlock(Node v, int i)          ▷ create new block B for Refresh to install in v.blocks[i]
    let B be a new Block with fields end_left := v.left.head − 1, end_right := v.right.head − 1,
        sum_enq := v.left.blocks[B.end_left].sum_enq + v.right.blocks[B.end_right].sum_enq,
        sum_deq := v.left.blocks[B.end_left].sum_deq + v.right.blocks[B.end_right].sum_deq
    num_enq := B.sum_enq − v.blocks[i − 1].sum_enq
    num_deq := B.sum_deq − v.blocks[i − 1].sum_deq
    if v = root then B.size := max(0, v.blocks[i − 1].size + num_enq − num_deq)
    end if
    if num_enq + num_deq = 0 then return null ▷ no blocks need to be propagated to v
    else return B
    end if
end CreateBlock

void Advance(Node v, int h) ▷ set v.blocks[h].super and increment v.head from h to h + 1
    if v ≠ root then
        h_p := v.parent.head
        CAS(v.blocks[h].super, null, h_p)
    end if
    CAS(v.head, h, h+1)
end Advance

⟨int, int⟩ IndexDequeue(Node v, int b, int i)
    ▷ return ⟨b′, i′⟩ such that ith dequeue in D(v.blocks[b]) is (i′)th dequeue of D(root.blocks[b′])
    ▷ Precondition: v.blocks[b] is not null, was propagated to root, and contains at least i dequeues
    if v = root then return ⟨b, i⟩
    else                                  ▷ First, find the superblock of v.blocks[b]
        dir := (v.parent.left = v ? left : right)
        sup := v.blocks[b].super
        if b > v.parent.blocks[sup].end_dir then sup := sup + 1
        end if
        ▷ compute index i of dequeue in superblock
        i += v.blocks[b − 1].sum_deq − v.blocks[v.parent.blocks[sup − 1].end_dir].sum_deq
        if dir = right then
            i += v.blocks[v.parent.blocks[sup].end_left].sum_deq −
                v.blocks[v.parent.blocks[sup − 1].end_left].sum_deq
        end if
        return IndexDequeue(v.parent, sup, i)
    end if
end IndexDequeue

element FindResponse(int b, int i) ▷ find response to ith dequeue in D(root.blocks[b])
    ▷ Precondition: 1 ⩽ i ⩽ |D(root.blocks[b])|
    num_enq := root.blocks[b].sum_enq − root.blocks[b − 1].sum_enq
    if root.blocks[b − 1].size + num_enq < i then
        return null                       ▷ queue is empty when dequeue occurs
    else                                  ▷ response is the eth enqueue in the root
        e := i + root.blocks[b − 1].sum_enq − root.blocks[b − 1].size
        use binary search to find min b_e ⩽ b with root.blocks[b_e].sum_enq ⩾ e
        i_e := e − root.blocks[b_e − 1].sum_enq  ▷ find rank of enqueue within its block
        return GetEnqueue(root, b_e, i_e)
    end if
end FindResponse
```

Fig. 6: Queue implementation's helper routines.

element GetEnqueue(Node v, int b, int i) $\triangleright$ returns argument of ith enqueue in $E(v.blocks[b])$

  $\triangleright$ Preconditions: $i \geqslant 1$ and $v.blocks[b]$ is non-null and contains at least i enqueues

  if v is a leaf node **then return** $v.blocks[b].element$

  else

    $sum_{left} := v.left.blocks[v.blocks[b].end_{left}].sum_{enq}$ $\triangleright$ # enqueues in $v.blocks[1..b]$ from $v.left$

    $prev_{left} := v.left.blocks[v.blocks[b{-}1].end_{left}].sum_{enq}$ $\triangleright$ # enqueues in $v.blocks[1..b-1]$ from $v.left$

    $prev_{right} := v.right.blocks[v.blocks[b{-}1].end_{right}].sum_{enq}$ $\triangleright$ # enqueues in $v.blocks[1..b-1]$ from $v.right$

    if $i \leqslant sum_{left} - prev_{left}$ **then** $dir := left$ $\triangleright$ required enqueue is in $v.left$

    else $\triangleright$ required enqueue is in $v.right$

      $dir := right$

      $i := i - (sum_{left} - prev_{left})$

    end if

    $\triangleright$ find enqueue's block in $v.dir.blocks$ and its rank within block

    use binary search to find minimum b' in range $[v.blocks[b-1].end_{dir}{+}1..v.blocks[b].end_{dir}]$ such that $v.dir.blocks[b'].sum_{enq} \geqslant i + prev_{dir}$

    $i' := i - (v.dir.blocks[b'-1].sum_{enq} - prev_{dir})$

    return GetEnqueue($v.dir$, b', i')

  end if

end GetEnqueue

Fig. 7: Queue implementation's GetEnqueue routine.

then use its position in the linearization for future Dequeue operations.

A **Dequeue** also appends a block to the process's leaf. The block's *element* field is null to indicate that it represents a Dequeue operation. After propagating the operation to the root, the Dequeue calls IndexDequeue to compute its position in the root and then calls FindResponse to compute its response.

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 calls Refresh on v twice in Lines 226 and 227. If both calls 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 226 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 B 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 B 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 new block B 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 $v.left.blocks[1..B.end_{left}]$ and $v.blocks[1..B.end_{right}]$, line 41 sets sum_{enq} to the sum of $v.left.blocks[B.end_{left}].sum_{enq}$ and $v.right.blocks[B.end_{right}].sum_{enq}$. Line 42 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 $B.sum_{enq}$. The values of $B.sum_{deq}$ and num_{deq} are computed similarly. Then, if B is going to be installed in the root, line 44 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 63–65 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 67–70 compute the dequeue's rank within the superblock's sequence of dequeues, using (3.1).

To compute the response of the ith Dequeue in the bth block of the root, **FindResponse**(b, i) determines at line 77 if the queue is empty. If not, line 80 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, $b_e \leqslant b$. To find the left end of the range for the binary search for b_e, we can first do a doubling search [6], 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 99) 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 53 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 55 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 55, $v.head$ 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 $0 \leqslant i < v.head$, $v.blocks[i] \neq$ 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, $v.head \leqslant h$ 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 55 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 55, the CAS at line 53 ensures that $v.blocks[i].super \neq$ null. $\square$

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*

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

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 41 of CreateBlock$(v, b-1)$ stores in $B'.end_{left}$ is less than or equal to the value that line 41 of CreateBlock(v, b) stores in $B.end_{left}$. Similarly, the values stored in $B'.end_{right}$ and $B.end_{right}$ at line 41 satisfy the claim. $\square$

Lemma 5. *If B and B' are two blocks in nodes at the same depth in the ordering tree, 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. $\square$

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. $\square$

The accuracy of the values stored in the sum_{enq} and sum_{deq} fields on lines 2, 6 and 41 follows easily from the definition of subblocks.

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

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

Proof. Initially, each *blocks* array contains only an empty block B_0 in location 0. By definition, $E(B_0)$ and $D(B_0)$ are empty sequences. Moreover, $B_0.sum_{enq} = B_0.sum_{deq} = 0$, so the claim is true.

We show that each installation of a block B into some location $v.blocks[i]$ preserves the claim, assuming the claim holds before this installation. We consider two cases.

If v is a leaf, B was created at line 2 or 6. For line 2, B represents a single enqueue, so $|E(B)| = 1$ and $|D(B)| = 0$. Since $B.sum_{enq}$ is set to $v.blocks[i-1].sum_{enq} + 1$ and $B.sum_{deq}$ is set to $v.blocks[i-1].sum_{deq}$, the claim follows from the hypothesis. The proof for line 6, where B represents a single dequeue, is similar.

Now suppose v is an internal node. By the definition of subblocks in (3.3) and Lemma 4, the subblocks of $v.blocks[1..i]$ are $v.left.blocks[1..B.end_{left}]$ and $v.right.blocks[1..B.end_{right}]$. Thus, the enqueues in $E(v.blocks[0]) \cdots E(v.blocks[i])$ are those in $E(v.left.blocks[0]) \cdots E(v.left.blocks[B.end_{left}])$ and those in $E(v.left.blocks[0]) \cdots E(v.left.blocks[B.end_{right}])$. By the hypothesis, the total number of these enqueues is $v.left.blocks[B.end_{left}].sum_{enq} + v.right.blocks[B.end_{right}].sum_{enq}$, which is the value that line 41 stored in $B.sum_{enq}$ when B was created. The proof for sum_{deq} (stored on line 41) is similar. $\square$

Invariant 7 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 46, $num_{enq} + num_{deq} \neq 0$. By Invariant 7, $num_{enq} = |E(B)|$ and $num_{deq} = |D(B)|$, so these sequences cannot both be empty. $\square$

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, $v.\mathit{left}$) contained an operation *op* when R executed line 25. Let i be the index such that the block $B_\ell = v.\mathit{left}.\mathit{blocks}[i]$ contains *op*. By Observation 1 and Lemma 4, the value of $\mathit{childHead}$ that R reads from $v.\mathit{left}.\mathit{head}$ in line 27 is at least i. If it is equal to i, R calls Advance at line 29, which ensures that $v.\mathit{left}.\mathit{head} > i$. Then, R creates a new block B by calling CreateBlock(v, h) at line 32, where h is the value R reads at line 25. CreateBlock reads a value greater than i from $v.\mathit{left}.\mathit{head}$ at line 41. Thus, $B.\mathit{end}_{\mathit{left}} \geqslant i$. We consider two cases.

Suppose R's CAS at line 35 installs B in $v.\mathit{blocks}$. Then, B_ℓ is a subblock of some block in v, since $B.\mathit{end}_{\mathit{left}}$ is greater than or equal to B_ℓ's index i in $v.\mathit{left}.\mathit{blocks}$. 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 41 is

$$\begin{aligned}
\mathit{num}_{\mathit{enq}} &= v.\mathit{left}.\mathit{blocks}[B.\mathit{end}_{\mathit{left}}].\mathit{sum}_{\mathit{enq}} \\
&\quad + v.\mathit{right}.\mathit{blocks}[B.\mathit{end}_{\mathit{right}}].\mathit{sum}_{\mathit{enq}} \\
&\quad - v.\mathit{blocks}[h-1].\mathit{sum}_{\mathit{enq}} \\
&= v.\mathit{left}.\mathit{blocks}[B.\mathit{end}_{\mathit{left}}].\mathit{sum}_{\mathit{enq}} \\
&\quad + v.\mathit{right}.\mathit{blocks}[B.\mathit{end}_{\mathit{right}}].\mathit{sum}_{\mathit{enq}} \\
&\quad - v.\mathit{left}.\mathit{blocks}[v.\mathit{blocks}[h-1].\mathit{end}_{\mathit{left}}].\mathit{sum}_{\mathit{enq}} \\
&\quad - v.\mathit{right}.\mathit{blocks}[v.\mathit{blocks}[h-1].\mathit{end}_{\mathit{right}}].\mathit{sum}_{\mathit{enq}}
\end{aligned}$$

It follows from Invariant 7 that $\mathit{num}_{\mathrm{enq}}$ is the total number of enqueues in $v.\mathit{left}.\mathit{blocks}[v.\mathit{blocks}[h-1].\mathit{end}_{\mathit{left}} + 1..B.\mathit{end}_{\mathit{left}}]$ and $v.\mathit{right}.\mathit{blocks}[v.\mathit{blocks}[h-1].\mathit{end}_{\mathit{right}} + 1..B.\mathit{end}_{\mathit{right}}]$. Similarly, $\mathit{num}_{\mathrm{deq}}$ is the total number of dequeues contained in these blocks. Since $\mathit{num}_{\mathrm{enq}} + \mathit{num}_{\mathrm{deq}} = 0$ at line 46, these blocks contain no operations. By Corollary 8, this means the ranges of blocks are empty, so that $v.\mathit{blocks}[h-1].\mathit{end}_{\mathit{left}} \geqslant B.\mathit{end}_{\mathit{left}} \geqslant i$. Hence, B_ℓ is already a subblock of some block in v, so v contains *op*. $\square$

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, $v.blocks[h_2] =$ 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

In this section, we show that the routines used to compute responses to Dequeue operations work correctly. We first prove the *super* field is accurate, since IndexDequeue uses it to trace superblocks up the tree. We do this 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 v's parent. In particular, $B.super$ is written after B is installed, but before B_s is installed.

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

Proof. We first show that $B.super \leqslant s$. Let R_s be the instance of Refresh$(v.parent)$ that installs B's superblock in $v.parent.blocks[s]$. By the definition of subblocks (3.3), R_s's read r of $v.head$ at line 41 obtains a value greater than b. By Invariant 3, $B.super$ is not null when r occurs, which means that $B.super$ was set (by line 53) to a value read from $v.parent.head$ before r. When r occurs, $v.parent.blocks[s] =$ null, since the later CAS by R_s at line 35 succeeds. So, by Invariant 3, $v.parent.head \leqslant s$ when r occurs. Since the value stored in $B.super$ was read from $v.parent.head$ before r and the *head* field is non-decreasing by Observation 1, it follows that $B.super \leqslant s$.

Next, we show that $B.super \geqslant s-1$. The value stored in $B.super$ at line 53 is read from $v.parent.head$ at line 52 and $head$ is always at least 1, so $B.super \geqslant 1$. So, if $s \leqslant 2$, the claim is trivial. Assume $s > 2$ for the rest of the proof. By Lemma 4, $v.parent.blocks[s-1]$ is not null. Let R_{s-1} be the call to Refresh($v.parent$) that installed the block in $v.parent.blocks[s-1]$. Let r' be the step when R_{s-1} reads $s-1$ in $v.parent.head$ at line 25. This read r' must be before B is installed in v; otherwise, Lemma 9 would imply that B is a subblock of one of $v.parent.blocks[1..s-1]$, contrary to the hypothesis. Now, consider the call to Advance(v, b) that writes $B.super$. It is invoked either at line 29 after seeing $v.blocks[b] \neq$ null at line 28 or at line 36 after ensuring $v.blocks[b] \neq$ null at line 35. Either way, the Advance is invoked after B is installed, and therefore after r'. By Observation 1, $v.parent.head$ is non-decreasing, so the value this Advance reads in $v.parent.head$ and writes in $B.super$ is greater than or equal to the value $s-1$ that r' reads in $v.parent.head$. $\square$

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 $1 \leqslant i \leqslant |D(v.blocks[b])|$, 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 $D(root.blocks[b'])$.*

Proof. We prove the claim by induction on the depth of node v. The base case where v is the root is trivial (see Line 60). Assuming the claim holds for v's parent, we prove it for v. Let $B = v.blocks[b]$ and B' be the superblock of B. IndexDequeue(v, b, i) first computes the index sup of B' in $v.parent$. By Lemma 12, this index is either $B.super$ or $B.super + 1$. The correct index is determined by testing on line 64 whether B is a subblock of $v.parent.blocks[B.super] + 1$.

Next, the position of the required dequeue in $D(B')$ (as defined by Equation (3.1)) is computed in lines 67–70. We first add the number of dequeues in the subblocks of B' in v that precede B on line 67. If v is the right child of its parent, then all of the subblocks of B' from v's left sibling also precede the required dequeue, so we add the number of dequeues in those subblocks in line 69.

Finally, IndexDequeue is called recursively on v's parent. Since B has been propagated to the root, so has its superblock B'. Thus, all preconditions of the recursive call are met. By the induction hypothesis, the recursive call returns the location of the required dequeue in the root. $\square$

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

Proof. We prove the claim by induction on the height of node v. If v is a leaf, the hypothesis implies that $i = 1$ and the block $v.blocks[b]$ represents an enqueue whose argument is stored in $v.blocks[b].element$. GetEnqueue returns the argument of this enqueue at line 88.

Assuming the claim holds for v's children, we prove it for v. Let B be $v.blocks[b]$. By Equation (3.1), $E(B)$ is obtained by concatenating the enqueue sequences of the direct subblocks of B, which are listed in (3.3). By Invariant 7, $sum_{left} - prev_{left}$ is the number of enqueues in $E(B)$ that come from B's subblocks in v's left child. Thus, dir is set to the direction for the child of v that contains the required enqueue operation. Moreover, when line 97 is reached, i is the position of the required enqueue within the portion E' of $E(B)$ that comes from that child. Thus, line 99 finds the index b' of the subblock B' containing the required enqueue. By Invariant 7, $v.dir.blocks[b'-1].sum_{enq} - prev_{dir}$ is the number of enqueues in E' before the enqueues of block B', so the value i' computed on line 100 is the position of the required enqueue within $E(B')$. Thus, the recursive call on line 101 satisfies its precondition, and returns the required result, by the induction hypothesis. □

4.4 Linearizability

We show that the linearization ordering L defined in Equation (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 each completed dequeue returns the same result as it would 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. □

We next show 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.*

Proof. We prove the claim by induction on b. The base case when $b = 0$ is trivially true, since the queue is initially empty and $root.blocks[0]$ contains an empty block whose *size* field is 0. Assuming the claim holds for $b - 1$, we prove it for b. The *size* field of the block B installed in $root.blocks[b]$ is computed at line 44 of a call to CreateBlock(*root, b*). By the induction hypothesis, $root.blocks[b-1].size$ gives the size of the queue before the operations of block B are performed. By Invariant 7, the values of num_{enq} and num_{deq} are the number of enqueues and dequeues contained in B. Hence, the size of the queue after the operations of B are performed (with enqueues before dequeues as specified by L) is $\max(0, root.blocks[b-1].size + num_{enq} - num_{deq})$. □

Next, we show each operation returns the same response as it 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 Deq terminates, it is contained in some block in the root, by Lemma 11. By Lemma 13, Deq's call to IndexDequeue on line 8 returns a pair $\langle b, i\rangle$ such that Deq is the ith dequeue in the block $B = root.blocks[b]$. Deq 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 76 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 Deq returns null on line 78 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 80 sets e to the rank of Deq among all the non-null dequeues in L. Thus, in the sequential execution L, Deq 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 Deq computes on line 81 and 82. By Lemma 14, the call to GetEnqueue returns the argument of the required enqueue. □

Combining Lemmas 15 and 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* $\mathsf{FindResponse}(b, i)$ *does at line 81 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 $b - b_e \leqslant 2 \cdot B_{b_e}.size + B_{b-1}.size + 1$. If $b \leqslant b_e + 1$, 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 $B_{b-1}.size \geqslant B_{b_e}.size + |E(B_{b_e+1}) \cdots E(B_{b-1})| - |D(B_{b_e+1}) \cdots D(B_{b-1})|$. Thus, $|E(B_{b_e+1}) \cdots E(B_{b-1})| \leqslant B_{b-1}.size + |D(B_{b_e+1}) \cdots D(B_{b-1})| \leqslant B_{b-1}.size + B_{b_e}.size$. 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, $b - 1 - b_e \leqslant B_{b-1}.size + 2 \cdot B_{b_e}.size$. ☐

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. B cannot contain any later operations by p, since B is created before those operations are invoked.

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 $c \leqslant p$, 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 81 and call GetEnqueue at line 83. 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 Bounded-Space Implementation

In the implementation described in Section 3, operations remain in the *blocks* arrays forever. Thus, the space used continues to grow as operations are 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 ensuring the (amortized) step complexity is still polylogarithmic. 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 81 and 99 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 [10, 26]. 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. [9]: 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 adds only 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 [4] used persistent RBTs in a similar way for a snapshot data structure.

To prevent RBTs from growing without bound, we must 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 $\mathsf{Enqueue}(x)$ operation must be kept until x is dequeued.

To maintain good amortized time, we periodically do a garbage collection (GC) phase. If a $\mathsf{Refresh}$ 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 [47, 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.

6.1 Detailed Description

To avoid confusion, we use nodes to refer to the nodes of the ordering tree, and blocks to refer to the nodes of a RBT (since the RBT stores blocks). The space-bounded implementation uses two shared arrays: the *leaf* array allows processes to access one another's leaves to perform helping, and the *last* array is used to determine which blocks are safe to discard.

The *blocks* field of each node in the ordering tree is implemented as a pointer to the root of a RBT of Blocks rather than an infinite array. Each RBT is initialized with an empty block with index 0. Any access to an entry of the *blocks* array is replaced by a search in the RBT. The node's *head* field, which previously gave the next position to insert into the *blocks* array is no longer needed; we can instead simply find the maximum *index* of any block in the RBT. To facilitate this, MaxBlock is a query operation on the RBT that returns the block with the maximum *index*. We can store, in the root of the RBT, a pointer to the maximum block so that MaxBlock can be done in constant time, without affecting the time of other RBT operations. Similarly, a MinBlock query finds the block with the minimum *index* in a RBT.

Blocks no longer require the *super* field. It was used to quickly find a block's superblock in the parent node's *blocks* array, but this can now be done efficiently by searching the parent's *blocks* RBT instead. Each Block has an additional field.

- int *index*

that represents the position this block would have in the *blocks* array. To facilitate helping, each Block in a leaf has one more additional field

- Object *response*

which is used only for blocks that store a dequeue operation and stores the response of the dequeue in the block.

Pseudocode for the space-bounded implementation appears in Figures 8 to 10. New or modified code appears in blue. The Propagate and GetEnqueue routines are unchanged. A few lines have been added to FindResponse to update the *last* array to ensure that it stores the value described above. Minor modifications have also been made to Enqueue, Dequeue, CreateBlock, Refresh and Append to accommodate the switch from an array of blocks to a RBT of blocks (and the corresponding disappearance of the *head* field). In addition, the second half of the Dequeue routine is now in a separate routine called CompleteDeq so that it can also be used by other processes helping to complete the operation. The Refresh routine no longer needs to set the *super* field of blocks since that field has been removed. The IndexDequeue routine, which must trace the location of a dequeue along a path from its leaf to the root has a minor modification to search the *blocks* RBT at each level instead of using the *super* field.

The new routines, AddBlock, SplitBlock, Help and Propagated are used to implement the garbage collection (GC) phase. When a Refresh or Append wants to add a new block to a node's *blocks* RBT, it calls the new **AddBlock** routine. Before attempting to add the block to a node's RBT, AddBlock triggers a GC phase on the RBT if the new block's *index* is a multiple of the constant G, which we choose to be $p^2\lceil\log p\rceil$. This ensures that obsolete blocks are removed from the RBT once every G times a new block is added to it. The GC phase uses SplitBlock to determine the index s of the oldest block to keep, calls Help to help all pending dequeues that have been propagated

```
▷ Shared variables
Node root                                        ▷ root of ordering tree
Node[] leaf[1..p]                                ▷ leaf[k] is the leaf assigned to process k
int[] last[1..p]                                 ▷ last[k] is max index of a root block that process k saw
                                                 ▷ contains either a dequeued enqueue or a null dequeue
void Enqueue(Object e)
    h := MaxBlock(leaf[id].blocks).index+1
    let B be a new Block with fields element := e, sum_enq := leaf[id].blocks[h − 1].sum_enq + 1,
        sum_deq := leaf[id].blocks[h − 1].sum_deq, index :=h
    Append(B)
end Enqueue
Object Dequeue()
    h := MaxBlock(leaf[id].blocks).index+1
    let B be a new Block with fields element := null, sum_enq := leaf[id].blocks[h − 1].sum_enq,
        sum_deq := leaf[id].blocks[h − 1].sum_deq + 1, index :=h
    Append(B)
    return CompleteDeq(leaf[id], h)
end Dequeue
Object CompleteDeq(Node leaf, int h)             ▷ finish propagated dequeue in leaf.blocks[h]
    ⟨b, i⟩ := IndexDequeue(leaf, h, 1)
    return FindResponse(b, i)
end CompleteDeq
void Append(Block B)                             ▷ append block to leaf and propagate to root
    leaf[id].blocks := AddBlock(leaf[id], leaf[id].blocks, B)
    Propagate(leaf[id].parent)
end Append
void Propagate(Node v)                           ▷ propagate blocks from v's children to root
    if not Refresh(v) then                       ▷ double refresh
        Refresh(v)
    end if
    if v ≠ root then                             ▷ recurse up tree
        Propagate(v.parent)
    end if
end Propagate
boolean Refresh(Node v)                          ▷ try to append a new block B to v.blocks
    T := v.blocks
    h := MaxBlock(T).index + 1
    B := CreateBlock(v, h)
    if B = null then return true
    else
        T' := AddBlock(v, T, B)
        return CAS(v.blocks, T, T')
    end if
end Refresh
RBT AddBlock(Node v, RBT T, Block B)             ▷ add block B ≠ null to T; do GC if necessary
    if B.index is a multiple of G then           ▷ do garbage collection
        s := SplitBlock(v).index
        Help
        T' := Split(T, s)                        ▷ Split removes blocks with index < s
        return Insert(T', B)
    else return Insert(T, B)
    end if
end AddBlock
```

Fig. 8: Bounded-space queue implementation's main routines for process number *id*.

```
Block SplitBlock(Node v)                    ▷ figure out where to split v's RBT
    if v = root then
        m := 0
        for k := 1..p do m := max(m, v.last[k])
        end for
        B := root.blocks[m − 1]
    else
        B_p := SplitBlock(v.parent)
        dir := (v = v.parent.left ? left : right)
        B := v.blocks[B_p.end_dir]
    end if
    return (B = null ? MinBlock(v.blocks) : B) ▷ If B was discarded, use leftmost block instead
end SplitBlock

void Help                                   ▷ help pending operations
    for ℓ in leaf[1..k] do
        B := MaxBlock(ℓ.blocks)
        if B.element = null and B.index > 0 and Propagated(ℓ, B.index) then
            ▷ operation is a propagated dequeue
            B.response := CompleteDeq(ℓ, B.index)
        end if
    end for
end Help

boolean Propagated(Node v, int b)           ▷ check if v.blocks[b] has propagated to root
    ▷ Precondition: v.blocks[b] exists
    if v = root then return true
    else
        T := v.parent.blocks
        dir := (v.parent.left = v ? left : right)
        if MaxBlock(T).end_dir < b then return false
        else
            B_p := minimum index block in T with end_dir ⩾ b
            return Propagated(v.parent, B_p.index)
        end if
    end if
end Propagated

Block CreateBlock(Node v, int i) ▷ create new block B to install in v.blocks[i]
    let B be a new Block with fields end_left := MaxBlock(v.left.blocks).index,
        end_right := MaxBlock(v.right.blocks).index, index := i
        sum_enq := v.left.blocks[B.end_left].sum_enq + v.right.blocks[B.end_right].sum_enq
        sum_deq := v.left.blocks[B.end_left].sum_deq + v.right.blocks[B.end_right].sum_deq
    num_enq := B.sum_enq − v.blocks[i − 1].sum_enq
    num_deq := B.sum_deq − v.blocks[i − 1].sum_deq
    if v = root then B.size := max(0, v.blocks[i − 1].size + num_enq − num_deq)
    end if
    if num_enq+num_deq = 0 then return null ▷ no blocks need to be propagated to v
    else return B
    end if
end CreateBlock
```

Fig. 9: Bounded-space queue implementation's routines for GC and creating new Block.

⟨int, int⟩ IndexDequeue(Node v, int b, int i)
    ▷ return $\langle b', i' \rangle$ such that ith dequeue in $D(v.blocks[b])$ is (i')th dequeue of $D(root.blocks[b'])$
    ▷ Precondition: $v.blocks[b]$ exists and has propagated to root and $|D(v.blocks[b])| \geqslant i$
    if $v = root$ **then return** $\langle b,\ i \rangle$
    else ▷ First, find the superblock B_p of $v.blocks[b]$
        $dir := (v.parent.left = v\ ?\ left : right)$
        $T := v.parent.blocks$
        $B_p :=$ min block in T with $end_{dir} \geqslant b$
        $B'_p :=$ max block in T with $end_{dir} < b$ ▷ predecessor of B_p
        ▷ compute index i of dequeue in superblock B_p
        $i \mathrel{+}= v.blocks[b-1].sum_{deq} - v.blocks[B'_p.end_{dir}].sum_{deq}$
        if $dir = right$ **then**
            $i \mathrel{+}= v.blocks[B_p.end_{left}].sum_{deq} - v.blocks[B'_p.end_{left}].sum_{deq}$
        end if
        return IndexDequeue($v.parent$, $B_p.index$, i)
    end if
end IndexDequeue

element FindResponse(int b, int i) ▷ find response to ith dequeue in $D(root.blocks[b])$
    ▷ Precondition: $1 \leqslant i \leqslant |D(root.blocks[b])|$
    $num_{enq} := root.blocks[b].sum_{enq} - root.blocks[b-1].sum_{enq}$
    if $root.blocks[b-1].size + num_{enq} < i$ **then** ▷ queue is empty when dequeue occurs
        if $b > last[id]$ **then** $last[id] := b$
        end if
        return null
    else ▷ response is the eth enqueue in the root
        $e := i + root.blocks[b-1].sum_{enq} - root.blocks[b-1].size$
        use BST search in $root.blocks$ to find minimum index b_e of a Block with $sum_{enq} \geqslant e$
        $i_e := e - root.blocks[b_e - 1].sum_{enq}$ ▷ find rank of enqueue within its block
        $res :=$ GetEnqueue($root$, b_e, i_e)
        if $b_e > last[id]$ **then** $last[id] := b_e$
        end if
        return res
    end if
end FindResponse

element GetEnqueue(Node v, int b, int i) ▷ returns argument of ith enqueue in $E(v.blocks[b])$
    ▷ Preconditions: $i \geqslant 1$ and $v.blocks[b]$ exists and contains at least i enqueues
    if v is a leaf node **then return** $v.blocks[b].element$
    else
        $sum_{left} := v.left.blocks[v.blocks[b].end_{left}].sum_{enq}$ ▷ # enqueues in $v.blocks[1..b]$ from $v.left$
        $prev_{left} := v.left.blocks[v.blocks[b-1].end_{left}].sum_{enq}$ ▷ # enqueues in $v.blocks[1..b-1]$ from $v.left$
        $prev_{right} := v.right.blocks[v.blocks[b-1].end_{right}].sum_{enq}$ ▷ # enqueues in $v.blocks[1..b-1]$ from $v.right$
        if $i \leqslant sum_{left} - prev_{left}$ **then** $dir := left$ ▷ required enqueue is in $v.left$
        else ▷ required enqueue is in $v.right$
            $dir := right$
            $i := i - (sum_{left} - prev_{left})$
        end if
        ▷ find enqueue's block in $v.dir.blocks$ and its rank within block
        use BST search in $v.dir.blocks$ to find minimum index b' of a Block with $sum_{enq} \geqslant i + prev_{dir}$
        $i' := i - (v.dir.blocks[b'-1].sum_{enq} - prev_{dir})$
        return GetEnqueue($v.dir$, b', i')
    end if
end GetEnqueue

Fig. 10: Bounded-space queue implementation's routines to compute responses to operations.

to the root (to ensure that all blocks before s can safely be discarded), and then uses the standard RBT Split routine [47] to remove all blocks with *index* less than s. Then AddBlock inserts the new block at line 248 or 249 and returns the root of the resulting RBT. The Append or Refresh then stores this root into the node's *blocks* field at line 222 or 240.

To determine the oldest block in a node v to keep, the **SplitBlock** routine first uses the *last* array to find the most recent block B_{root} in the root that contains either an enqueue that has been dequeued or a null dequeue. By the FIFO property of queues, all enqueues in blocks before B_{root} are either dequeued or will be dequeued by a dequeue that is currently in progress. Once those pending dequeues have been helped to complete by line 246, it is safe to discard any blocks in the root older than B_{root}, as well as their subblocks.[1] The SplitBlock uses the end_{left} and end_{right} fields to find the last block in v that is a subblock of B_{root} (or any older block in the root, in case B_{root} has no subblocks in v). While SplitBlock is in progress, it is possible that some block that it needs in a node v' along the path from v to the root is discarded by another GC phase. In this case, SplitBlock uses the last subblock in v of the oldest block in v' instead (since a GC phase on v' determined that all blocks older than that are safe to discard anyway).

The **Help** routine is fairly straightforward: it loops through all leaves and helps the dequeue that is in progress there if it has already been propagated to the root. The **Propagated** function is used to determine whether the dequeue has propagated to the root.

In the code, we use $v.blocks[i]$ to refer to the block in the RBT stored in $v.blocks$ with index i. A search for this block may sometimes not find it, if it has already been discarded by another process's GC phase. As mentioned above, if this happens to an enqueue operation, the enqueue can simply terminate because the fact that the block is gone means that another process has helped the enqueue reach the root of the ordering tree. Similarly, if a dequeue operation performs a failed search on a RBT, the dequeue can return the value written in the *response* field of the leaf block that represents the dequeue and terminate, since some other process will have written the *response* there before discarding the needed block. We do not explicitly write this early termination in the pseudocode every time we do a lookup in an RBT. There is one exception to this rule: if an RBT lookup for block B returns null on line 257 or 261 of SplitBlock because the required block has been discarded, we continue doing GC, since we do not want a GC phase on one node to be prevented from cleaning up its RBT because a GC phase on

[1] If we used the more conservative approach of discarding blocks whose indices are smaller than the *minimum* entry of *last* instead of the maximum, helping would be unnecessary, but then one slow process could prevent GC from discarding any blocks, so the space would not be bounded.

a different node threw away some blocks that were needed. Line 263 says what to do in this case.

6.2 Correctness

There are enough changes to the algorithm that a new proof of correctness is required. Its structure mirrors the proof of the original algorithm, but requires additional reasoning to ensure GC does not interfere with other routines.

6.2.1 Basic Properties

The following observation describes how the set of blocks in a node's RBT can be modified.

Lemma 24. *Suppose a step of the algorithm changes $v.blocks$ from a non-empty tree T to T'. If the set of index values in T is I, then the set of index values in T' is $(I \cap [m-1, \infty)) \cup \{\max(I)+1\}$ for some m.*

Proof. The RBT of a node is updated only at line 222 or 240.

If line 222 of an Append operation modifies $v.blocks$, then v is a leaf node, and no other process ever modifies $v.blocks$. T' was obtained from T by calling AddBlock(v, T, B). B was created either by the Enqueue or Dequeue that called Append. Either way, $B.index = \max(I) + 1$. The AddBlock that creates T' may optionally Split the RBT at line 247 and then add B to it. So the claim is satisfied.

If line 240 of a Refresh modifies $v.blocks$, then v is an internal node. After reading T from $v.blocks$ at line 234, the Refresh then creates the block B, and calls AddBlock(v, T, B) to create T'. Line 288 sets $B.index = \max(I) + 1$. The AddBlock that creates T' may optionally Split the RBT and then add B to it. So the claim is satisfied. □

Since each RBT starts with a single block with *index* 0, the following is an easy consequence of Lemma 24.

Corollary 25. *The RBT stored in each node v is never empty and always stores a set of blocks with consecutive indices. Moreover, its maximum index can only increase over time.*

Since RBTs are always non-empty, calls to MaxBlock have well-defined answers. Throughout the proof, we use $v.blocks[b]$ to refer to the block with *index* b that appeared in v's tree at some time during the execution. It follows from Lemma 24 and Corollary 25 that each time a new block appears in v's RBT, its *index* is greater than any block that has appeared in v's RBT earlier. Thus, $v.blocks[b]$ is unique, if it exists. We also use this notation in the code to indicate that a search of the RBT $v.blocks$ should be performed for the block with *index* b.

We now establish that Definition (3.3) of a block's subblocks still makes sense by proving the analogue of Lemma 4.

Lemma 4′. *If v is an internal node and a block with index $h > 0$ has been inserted into $v.blocks$*

then $v.blocks[h-1].end_{left} \leqslant v.blocks[h].end_{left}$ *and* $v.blocks[h-1].end_{right} \leqslant v.blocks[h].end_{right}$.

Proof. The block B with index h was installed into v's RBT by the CAS at line 240. Suppose that CAS changed the tree from T to T'. Before this CAS, line 234 read the tree T from $v.blocks$, line 235 found a block B' with $index\ h-1$ in T, and then line 236 created the block B with $index = h$. Since B' was already in T before B was created, the CreateBlock$(v, b-1)$ that created B' terminated before the CreateBlock(v, b) that created B started. By Corollary 25, the value that line 288 of CreateBlock$(v, b-1)$ stores in $B'.end_{left}$ is less than or equal to the value that line 288 of CreateBlock(v, b) stores in $B.end_{left}$. Similarly, the values stored in $B'.end_{right}$ and $B.end_{right}$ at line 288 satisfy the claim. □

Lemma 4′ implies that the nodes of an in-order traversal of any RBT have non-decreasing values of end_{left} (and of end_{right}). Thus, the searches for a block based on end_{left} or end_{right} values at lines 282, 304 and 305, which are used to look for the superblock of a node or its predecessor, can be done using an ordinary BST search.

Lemma 5, Corollary 6, Invariant 7 and Corollary 8 all hold for the modified algorithm. Their proofs are identical to those given in Section 4.1 since they depend only on Lemma 4 (which can be replaced by Lemma 4′) and the definition of subblocks given in (3.3). In particular, Invariant 7 says that nodes in an in-order traversal of a RBT have non-decreasing values of sum_{enq} so the searches for a block based on sum_{enq} values in lines 323 and 344 can be done using an ordinary BST search.

6.2.2 Propagating Operations to the Root

Next, we prove an analogue of Lemma 9. We say a node v *contains* an operation if some block containing the operation has previously appeared in the RBT $v.block$ (even if the block has been removed from the RBT by a subsequent Split during garbage collection).

Lemma 9′. *Let R be a call to* Refresh*(v) that performs a successful* CAS *on line 240 (or terminates at line 237). In the configuration after that CAS (or termination, respectively), v contains all operations that v's children contained when R executed line 234.*

Proof. Suppose v's child (without loss of generality, $v.left$) contained an operation op when R executed line 234. Let i be the index of the block B_ℓ containing op that was in $v.left$'s RBT before R executed line 234. We consider two cases.

Suppose R's call to CreateBlock returns a new block B that is installed in $v.blocks$ by R's CAS at line 240. The CreateBlock set $B.end_{left}$ to the maximum $index$ in $v.left$'s RBT at line 288. By

Corollary 25, this maximum *index* is bigger than i. By the definition of subblocks, some block in v contains B_ℓ as a subblock and therefore v contains *op*.

Now suppose R's call to CreateBlock returns null, causing R to terminate at line 237. Let h be the maximum *index* in T plus 1. By reasoning identical to the last paragraph of Lemma 9's proof, it follows from the fact that $num_{enq} + num_{deq} = 0$ at line 293 that the blocks $v.left.blocks[v.blocks[h-1].end_{left}+1..B.end_{left}]$ and $v.right.blocks[v.blocks[h-1].end_{right}+1..B.end_{right}]$ contain no operations. By Corollary 8, each block contains at least one operation, so these ranges must be empty, and $v.blocks[h-1].end_{left} \geqslant B.end_{left} \geqslant i$. This implies that the block B_ℓ containing *op* is a subblock of some block that has appeared in v's RBT, so *op* is contained in v. ☐

This allows us to show that a double Refresh propagates operations up the tree, as in Lemma 10.

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 240 or terminates at line 237, the claim follows from Lemma 9′. So suppose both R_1 and R_2 perform a failed CAS at line 240. Then some other CAS on $v.blocks$ succeeds between the time each Refresh reads $v.blocks$ at line 234 and performs its CAS at line 240. Consider the Refresh R_3 that does this successful CAS during R_2. R_3 must have read $v.blocks$ after the successful CAS during R_1. The claim follows from Lemma 9′ applied to R_3. ☐

Lemma 11 can then be proved in the same way as in Section 4.2.

6.2.3 GC Keeps Needed Blocks

The correctness of GetEnqueue and IndexDequeue, which are very similar to the original implementation, are dependent only on the fact that GC does not discard blocks needed by those routines. The following results are used to show this.

We say a block is *finished* if

- it has been propagated to the root,
- the value of each enqueue contained in the block has either been returned by a dequeue or written in the *response* field of a dequeue, and
- each dequeue contained in the block has terminated or some process has written to the *response* field in the leaf block that represents it.

Intuitively, once a block is finished, operations no longer need the block to compute responses to operations. The following is an immediate consequence of the definition of finished and what it means for an operation to be contained in a block.

Observation 26. *A block is finished if and only all of its subblocks are finished.*

Invariant 27. *If the minimum index of any block in v's RBT is b_{min}, then each block with index at most b_{min} that was ever added to v is finished.*

Proof. The invariant is true initially, since the minimum *index* block in v's RBT is the empty block, which is (vacuously) finished.

We show that every step preserves the invariant. We need only consider a step *st* that modifies a node's RBT. The minimum *index* of v's RBT can only change when v's RBT changes, either at line 222 (if v is a leaf) or at line 240 (if v is an internal node). In either case, the step *st* changes $v.blocks$ from T to T', where T' is obtained by a call A to AddBlock(v, T, B). (In the case of a leaf v, this is true because only the process that owns the leaf ever writes to $v.blocks$.) If A does not do GC (lines 244–248), then T' is obtained by adding a new block to T, so by Lemma 24 the minimum *index* is unchanged and the invariant is trivially preserved. So consider the case where A performs GC. We must show that any block of v whose index is less than or equal to the minimum *index* in T' is finished when T' is installed in $v.blocks$. Since T' is obtained by discarding all blocks with *index* values less than or equal to s (and adding a block with a larger index), it suffices to show that all blocks that were ever added to v's RBT with *index* at most s are finished.

We must examine how A's call at line 245 to the recursive algorithm SplitBlock computes the value of s.

Claim 27.1. *If one of the recursive calls to* SplitBlock(x) *within A's call to* SplitBlock *returns a block B, then B and all earlier blocks in x are finished when st occurs.*

Proof of Claim. We prove this claim by induction on the depth of x.

For the base case, suppose x is the root. SplitBlock finds the maximum value m in *last*, which is the index of some block that contains an operation that is either a null dequeue or an enqueue whose value is the response for a dequeue that has been propagated to the root (since these are the only ways that an entry of *last* can be set to m). By the FIFO property of queues, the values enqueued by enqueues in $root.blocks[1..m-1]$ are all dequeued by operations that have already been installed in $root.blocks$ before the end of the SplitBlock. Between the termination of A's call to SplitBlock at line 245 and the CAS step *st* after A terminates, A helps all pending dequeues at line 246. Thus, after this helping (and before step *st*), $root.blocks[1..m-1]$ are all finished blocks. Since SplitBlock returns $root.blocks[m-1]$, the claim is true.

For the induction step, we assume the claim holds for x's parent, and prove it for x. We consider two cases.

If SplitBlock(x) returns the minimum block of x's RBT at line 263, then the claim follows from the assumption that Invariant 27 holds at all times before st.

Otherwise, SplitBlock(x) returns the block B at line 263. By the induction hypothesis, the block B_p computed at line 259 (and all earlier blocks of x) are finished when st occurs. By Observation 26, the block B in x indexed by $B_p.end_{left}$ or $B_p.end_{right}$ is also finished when st occurs. This completes the proof of Claim 27.1. $\diamondsuit$

If the Split at line 247 of A modifies the RBT, then it discards all the blocks older than the one returned by SplitBlock at line 245. By Claim 27.1, the minimum block in the new tree will satisfy the invariant when st installs the new tree in $v.blocks$. $\square$

We remark that Invariant 27 guarantees that GC keeps one block that is finished and discards all blocks with smaller indices. This is because the first unfinished block may still be traversed by an operation in the future, and when examining that block the operation may need information from the preceding block. For example, when FindResponse is called on a block with index b, line 316 looks up the block with index $b-1$.

Lemma 28. *If a* Dequeue *operation fails to find a block in an RBT, then it has been propagated to the root and its result has been written in the response field of the leaf block it created. If an* Enqueue *operation fails to find a block in an RBT, it has been propagated to the root.*

Proof. Any block that GC removes from an RBT is finished, by Invariant 27, so if an Enqueue or Dequeue fails to find a block while it is propagating itself up to the root (for example, during the CreateBlock routine), then a block containing the operation itself has been removed from a RBT, so the operation has propagated to the root, by the definition of finished. Moreover, by Invariant 27, if the operation is a dequeue, then its result is in its *response* field.

After propagation to the root, a Dequeue must access blocks that contain the dequeue, as well as the enqueue whose value it will return (if it is not a null dequeue). If any of those blocks have been removed, it follows from Invariant 27 that the Dequeue's result is written in its *response* field. $\square$

By Lemma 28, an operation that fails to find a block in an RBT can terminate. If it is a Dequeue, it can return the result written in its *response* field. Since no other process updates the RBT in a process's leaf, the block containing the *response* will be the last block in the leaf's RBT, and the last block of an RBT is never removed by GC.

Thus, the *response* field will still be there when a Dequeue needs it.

6.2.4 Linearizability

The correctness of the IndexDequeue and GetEnqueue operations can be proved in the same way as in Section 4.3, since they are largely unchanged (except for the simplification that IndexDequeue can simply search for a block's superblock instead of using the block's *super* field to calculate the superblock's position). They will give the correct response, provided none of the blocks they need to access have been removed by GC. But as we have seen above, if that happens, the Enqueue or Dequeue can simply terminate.

Similarly, the results of Section 4.4 can be reproved in exactly the same way as for the original algorithm to establish that the space-bounded algorithm is linearizable.

6.3 Analysis

Lemma 21 shows that each block contains at most one operation of each process. Its proof depends only on Lemma 5, Corollary 8 and Lemma 11, which are all still true for the space-bounded implementation. So the same proof of Lemma 21 still applies.

We first bound the size of RBTs. Let q_{max} be the maximum size of the queue at any time during the sequential execution given by the linearization L. Recall that GC is done on a node every G times its RBT is updated, and we chose G to be $p^2\lceil\log p\rceil$. Part of the proof of the following lemma is similar to the proof of Lemma 20.

Lemma 29. *If the maximum index in a node's RBT is a multiple of G, then it contains at most $3q_{max} + 5p + 1$ blocks.*

Proof. Consider the invocation A of AddBlock that updates a node v's RBT with the insertion of a block whose *index* is a multiple of G. Then, A performs a GC phase. Let C be the configuration before A invokes SplitBlock on line 245. That call to SplitBlock recurses up to the root, where it computes m by reading the *last* array. Let L_1 be the prefix of the linearization L corresponding to blocks $1..m$ of the root. Let L_2 be the next segment of the linearization corresponding to blocks $m+1..\ell$ of the root, where ℓ is the last block added to the root's RBT before C.

We first bound the number of operations in L_2.

The number of enqueues in L_2 whose values are still in the queue at the end of L_2 is at most q_{max}. If the value enqueued by any enqueue in L_2 is not still in the queue at the end of L_2, then the dequeue in L_2 that dequeued that value must still be in progress at C; otherwise the process that performed that dequeue would have set its *last* entry to the index of the root block that contains the enqueue, which is greater than m, before C, contradicting the fact that all values in the *last* array at C are less than or equal to m. So, there are at most p enqueues in L_2 whose values are still in the queue at the end of L_2. Thus, there are at most $q_{max} + p$ enqueues in L_2.

If a dequeue in L_2 returns a non-null value in the sequential execution L, then the value it returns was either in the queue at the end of L_1 or it was enqueued during L_2. Thus, there are at most $q_{max} + (q_{max} + p)$ dequeues in L_2 that return non-null values. Any dequeue in L_2 that returns a null value in the sequential execution L must still be in progress at C; otherwise the process that performed the dequeue would have set its *last* entry to a value greater than the index of the root block that contains the dequeue prior to C, contradicting the definition of m. So, there are at most p null dequeues in L_2. Thus, there are at most $2q_{max} + 2p$ dequeues in L_2.

A's call to SplitBlock determines the index s used to split v's RBT by following end_{left} and end_{right} pointers from the root down to v. So, the block returned is a subblock of the root block with index m, unless at some point along the path of subblocks the subblock has already been removed by a split, in which case SplitBlock returns a subblock of a root block with index $m' > m$.

Next, we bound the number of operations in v's blocks that are retained when A sets T':=Split(T, s). Since T was read before C, any operation in T is either in progress at C or has been propagated to the root before C, by Lemma 11. Thus, there are at most p operations in T that do not appear in $L_1 \cdot L_2$. All the rest of the operations in blocks of T' have been propagated to blocks $m..\ell$ of the root. By Lemma 21, There are at most p operations in block m of the root and we showed above that there are at most $3q_{max} + 3p$ in blocks $m+1..\ell$ of the root. Thus, there are at most $3q_{max} + 5p$ operations in blocks of T'. Since each block is non-empty by Corollary 8, T' contains at most $3q_{max} + 5p$ blocks, and one more block is inserted before A sets v's *blocks* to the resulting RBT. $\square$

Corollary 30. *At all times, the size of a node's RBT is* $O(q_{max} + p + G)$.

Proof. Each update to a node's RBT adds at most one block to it, increasing its maximum *index* by 1. Thus, there are at most G updates since the last time its maximum *index* was a multiple of G. The claim follows from Lemma 29. $\square$

The following theorem bounds the space that is reachable (and therefore cannot be freed by the environment's garbage collector) at any time.

Theorem 31. *The queue data structure uses a maximum of* $O(pq_{max} + p^3 \log p)$ *words of memory at any time.*

Proof. There are $2p-1$ nodes in the ordering tree. Aside from the RBT, each node uses $O(1)$ memory words. Each process may hold pointers to $O(1)$ RBTs that are no longer current in local variables.

So the space bound follows from Corollary 30 and the fact that G is chosen to be $p^2\lceil\log p\rceil$. $\square$

Performing GC on a node takes $\Theta(p \log p \log(q_{max}+p))$ steps in the worst case (as explained in the following proof), so an individual operation can take up to $\Theta(p\log^2 p\log(q_{max}+p))$ steps if it helps at each node along the path from a leaf to the root. However, we show that operations still have polylogarithmic amortized step complexity.

Theorem 32. *The amortized step complexity of each operation is $O(\log p \log(p + q_{max}))$.*

Proof. It follows from Corollary 30 and our choice of $G = p^2\lceil\log p\rceil$ that all the RBT routines we use to perform Split, Insert and searches for blocks with a particular index or for a sum_{enq} value (in line 323 or 344) can be done in $O(\log(p + q_{max}))$ steps.

First, we bound the number of steps taken *excluding* the GC phase in line 244–248. An Enqueue or null Dequeue does $O(1)$ RBT operations and other work at each level of the tree during Propagate, for a total of $O(\log p \log(p + q_{max}))$ steps. A non-null Dequeue must also search for a block in the root at line 323 and call GetEnqueue. At each level of the tree, GetEnqueue does $O(1)$ RBT operations (including a search at line 344) and $O(1)$ other steps. Thus, a Dequeue also takes $O(\log p \log(p + q_{max}))$ steps.

Now we consider the additional steps a process takes while doing GC in line 244–248 and show that the amortized number of GC steps each operation performs is also $O(\log p \log(p + q_{max}))$. If a process does GC in a call to AddBlock(v, T, B) where B has *index* $r \cdot G$ for some integer r, we call this the process's *rth GC phase on v*.

We argue that each process P can do an rth GC phase on v at most once. Consider P's first call A to AddBlock that does an rth GC phase on v. Let v, T, B be the arguments of A. Any call to AddBlock on internal node v is from line 239 of Refresh, so $B.index$ is the maximum *index* in T plus 1. The Refresh that called A performed a CAS at line 240. Either the CAS succeeds or it fails because some other CAS changes $v.blocks$ from T to another tree. Either way, by Lemma 24, v's RBT's maximum *index* will be at least $r \cdot G$ at all times after this CAS. So if a subsequent Refresh by process P ever calls AddBlock on v again, the block it passes as the third argument will have $index > r \cdot G$, so P will not perform an rth GC phase on v again.

Each GC phase takes $O(p)$ steps for SplitBlock to read the *last* array and figure out where to split the *blocks* RBT, $O(p \log p \log(p + q_{max}))$ steps in Help, and $O(\log(p+q_{max}))$ steps to split and insert a new node into the RBT. Thus, for each integer r and each node v, a total of $O(p^2 \log p \log(p + q_{max}))$ steps are performed by all processes during their rth helping phase on v. We can amortize these steps over the operations that appear in $v.blocks[(r-1)G+1..rG]$. By Corollary 8, there are at least G such operations, so each operation's amortized number of steps for GC at each node along the path from its leaf to the root is $O(p^2 \log p \log(p+q_{max})/G) = O(\log(p+q_{max}))$. Hence each operation's amortized number of GC steps is $O(\log p \log(p+q_{max}))$. □

The implementation remains wait-free: the depth of recursion in each routine is bounded by the height of the tree and the only loop is the counted loop in the Help routine. Moreover, since each operation still does only two CAS instructions at each level of the tree (at line 240), the following proposition still holds for the space-bounded version of the queue.

Proposition 19′. *Each operation performs* $O(\log p)$ CAS *instructions in the worst case.*

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 [29] to get a queue that has good performance in practice while also having good worst-case step complexity.

It would be interesting to close the gap that remains between our queue, which takes $O(\log^2 p + \log q)$ steps per operation, and Jayanti, Tarjan and Boix-Adserà's $\Omega(\log p)$ lower bound [22]. For the more relaxed bag data structure the gap is larger between the $\Omega(\min(c, \log\log p))$ lower bound [3] and our upper bound of $O(\log^2 p + \log q)$. Could the complexity for either queues or bags be made polylogarithmic in p while being independent of the size q of the data structure?

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 the routines of the implementation in Section 3 to implement a restricted 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 ith element in the sequence, and Index(e) to compute the position of element e in the sequence. Only the Append operations need to be propagated to the root of the ordering tree since the other two operations do not affect the state of the object. An Append(e) is implemented like Enqueue(e) in $O(\log p)$ steps. A Get(i) is similar to GetEnqueue, taking $O(\log n + \log^2 p)$ steps when the vector has n elements. An Index is similar to IndexDequeue (except operating on enqueues instead of dequeues) and would take $O(\log p)$ steps if the argument is a pointer to the leaf block that contains the element e.

Building on the work described in this paper, Asbell and Ruppert [2] have designed a doubly-ended queue with polylogarithmic amortized step complexity, which also yields a stack as a special case. This required a substantially different representation of the data stored in the ordering tree. Whether the ordering tree could also be used to obtain a priority queue with polylogarithmic step complexity remains an open question.

Acknowledgements

We thank Franck van Breugel, Siddhartha Jayanti, Philipp Woelfel 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. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[2] Shalom Asbell and Eric Ruppert. Wait-free stacks and deques with polylogarithmic step complexity. Manuscript, 2023.

[3] 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.

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

[5] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and time bounded multiversion garbage collection. In *Proc. 35th International Symposium on Distributed Computing*, volume 209 of *LIPIcs*, pages 12:1–12:20, 2021.

[6] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[7] 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, 2005.

[8] 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.

[9] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer*

and System Sciences, 38(1):86–124, February 1989.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] Andreas Haas. *Fast Concurrent Data Structures Through Timestamping.* PhD thesis, University of Salzburg, 2015.

[17] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[18] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[19] 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.

[20] 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.

[21] 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.

[22] Siddhartha V. Jayanti, Robert E. Tarjan, and Enric Boix-Adserà. Randomized concurrent set union and generalized wake-up. In *Proc.*

ACM Symposium on Principles of Distributed Computing, pages 187–196, 2019.

[23] Colette Johnen, Adnane Khattabi, and Alessia Milani. Efficient wait-free queue algorithms with multiple enqueuers and multiple dequeuers. In *Proc. 26th International Conference on Principles of Distributed Systems*, volume 253 of *LIPIcs*, pages 4:1–4:19, February 2023.

[24] Michael Kenzel, Stefan Lemme, Richard Membarth, Matthias Kurtenacker, Hugo Dvillers, Markus Steinberger, and Philipp Slusallek. AnyQ: An evaluation framework for massively-parallel queue algorithms. In *Proc. 37th IEEE International Parallel and Distributed Processing Symposium*, 2023. To appear.

[25] 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.

[26] Jeremy Ko. The amortized analysis of a non-blocking chromatic tree. *Theoretical Computer Science*, 840:59–121, November 2020.

[27] 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.

[28] 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.

[29] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *ACM SIGPLAN Not.*, 47(8):141–150, 2012.

[30] Nikita Koval, Dan Alistarh, and Roman Elizarov. Fast and scalable channels in Kotlin coroutines. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 107–118, 2023.

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

[32] Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto, 2001. Available from https://tspace.library.utoronto.ca/bitstream/1807/16583/1/MQ62967.pdf.

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

[34] 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.

[35] 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):5:1–5:49, March 2022.

[36] 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.

[37] 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.

[38] Hossein Naderibeni. A wait-free queue with poly-logarithmic worst-case step complexity. Master's thesis, York University, Toronto, Canada, November 2022. Available from https://yorkspace.library.yorku.ca/xmlui/handle/10315/40975.

[39] 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.

[40] 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.

[41] Peter Pirkelbauer, Reed Milewicz, and Juan Felipe Gonzalez. A portable lock-free bounded queue. In *Proc. 16th International Conference on Algorithms and Architectures for Parallel Processing*, volume 10048 of *LNCS*, pages 55–73, 2016.

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

[43] Raed Romanov and Nikita Koval. The state-of-the-art LCRQ concurrent queue algorithm does NOT require CAS2. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 14–26, 2023.

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

[50] 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.