Efficient Wait-Free Queue Algorithms with Multiple Enqueuers and Multiple Dequeuers

Colette Johnen ✉
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Adnane Khattabi ✉
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Alessia Milani ✉
Aix Marseille Univ, CNRS, LIS, UMR 7020, Marseille, France

Abstract

Despite the widespread usage of `FIFO` queues in distributed applications, designing efficient `wait-free` implementations of queues remains a challenge. The majority of wait-free queue implementations restrict either the number of dequeuers or the number of enqueuers that can operate on the queue, even when they use strong synchronization primitives, like the *Compare&Swap*. If we do not limit the number of processes that can perform enqueue and dequeue operations, the best-known upper bound on the worst case step complexity for a wait-free queue is given by Khanchandani and Wattenhofer [10]. In particular, they present an implementation of a multiple dequeuer multiple enqueuer wait-free queue whose worst case step complexity is in $O(\sqrt{n})$, where n is the number of processes. In this work, we investigate whether it is possible to improve this bound. In particular, we present a wait-free FIFO queue implementation that supports n enqueuers and k dequeuers where the worst case step complexity of an *Enqueue* operation is in $O(\log n)$ and of a *Dequeue* operation is in $O(k \log n)$.

Then, we show that if the semantics of the queue can be relaxed, by allowing concurrent *Dequeue* operations to retrieve the same element, then we can achieve $O(\log n)$ worst-case step complexity for both the *Enqueue* and *Dequeue* operations.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Theory of computation → Proof complexity

Keywords and phrases Distributed computing, distributed algorithms, FIFO queue, shared memory, fault tolerance, concurrent data structures, relaxed specifications, complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2022.4

Funding *Adnane Khattabi*: Adnane Khattabi is supported by UMI Relax.

1 Introduction

1.1 Context

Shared FIFO queues are an important building block for the design of many concurrent applications. Many implementations of concurrent FIFO queues have been proposed using shared objects provided by multiprocessor architectures, e.g. *Compare&Swap*, registers, *Fetch&Add*, and so on. In this paper, we are interested in `wait-free` implementations of shared queues where any operation by a correct process is guaranteed to terminate after a finite number of steps.

The design of efficient wait-free and linearizable concurrent queues is a difficult task even if the implementation is allowed to rely on strong synchronization primitives like *Compare&Swap*. Most implementations limit either the number of enqueuers or the number of dequeuers. In particular, David [3] presents a wait-free linearizable queue with a single enqueuer and multiple dequeuers with constant step complexity. Jayanti and Petrovic [9] provide an implementation of a multiple enqueuer, single dequeuer queue with $O(\log n)$ worst-case step complexity, where n is the number of processes. More recently, Khanchandani and Wattenhofer proposed a multiple enqueuer and multiple dequeuer wait-free queue implementation where both the enqueue and the dequeue operations have a worst-case step complexity of $O(\sqrt{n})$. In this paper, we investigate if this complexity represents the cost necessary in order to not limit the number of processes that can apply enqueue and dequeue operations on the concurrent queue.

By extension of algorithmic ideas from [9], we first show that a better complexity can be achieved even with multiple enqueuers and multiple dequeuers. In particular, we present a wait-free linearizable concurrent queue for n processes from which all n are enqueuers and $k \leq n$ are dequeuers. In our implementation, the step complexity of an *Enqueue* operation is in $O(\log n)$, while the complexity of a *Dequeue* operation is in $O(k \log n)$. Our implementation has logarithmic complexity as long as k is a constant. Also, it improves on the implementation by Khanchandani and Wattenhofer solution as long as $k \in O(\frac{\sqrt{n}}{\log n})$.

Then, we show that both *Enqueue* and *Dequeue* operations can have worst-case step complexity in $O(\log n)$, if we allow concurrent *Dequeue* operations to return the same element. This relaxed semantic denoted *multiplicity* has been formalized and introduced for the FIFO queue in [1]. Table 1 summarizes the state of the art and compares it to the contributions in this work.

Table 1 Comparing the contributions to state-of-the-art queue implementations (n is the number of processes and m is the number of enqueued elements).

	Step complexity	Space complexity	Concurrency limit	CAS - LL/SC	Fetch&Inc - Swap
Khanchandani and Wattenhofer [10]	$O(\sqrt{n})$	$O(nm)$ of $O(max(\log n, \log m))$ registers	None	Y	Y
David [3]	$O(1)$	Unbounded	Single enqueuer	N	Y
Jayanti and Petrovic [9]	$O(\log n)$	$O(n+m)$	Single dequeuer	Y	N
Li [13]	$O(m)$	Unbounded	2 dequeuers	N	Y
Eisenstat [4]	$O(m)$	Unbounded	2 enqueuers	N	Y
Exact queue (this work)	$O(\log n)$ for Enq $O(k \log n)$ for Deq	Unbounded	k dequeuers	Y	Y
Relaxed queue (this work)	$O(\log n)$	Unbounded	None	Y	Y

1.2 Other Related Work

Several papers propose wait-free linearizable shared queue implementations that only use registers and `Common2` objects (a particular set of base objects with consensus number 2). All of them limit the concurrency. In particular, there are queues shared by one or two dequeuers and any number of enqueuers [8, 13] and a queue with a single enqueuer and any number of dequeuers [3]. In fact, it is a long-standing open problem if it is possible to implement a wait-free linearizable queue that supports at least three enqueuers and three dequeuers based only on registers and consensus 2 objects. Among all the aforementioned queue implementations, only the one by David [3] has sublinear step complexity.

Using *Compare&Swap*, some practical wait-free queue implementations that support multiple enqueuers and multiple dequeuers have been proposed [5, 12, 14, 16]. Some of these implementations are wait-free [5, 12, 16]; while some are only lock-free [14]. All these solutions have been evaluated empirically and do not have formal complexity analysis. Nonetheless, the worst-case step complexity of either the *Enqueue* or of the *Dequeue* operation is not sublinear.

More recently, relaxed queues have been proposed to overcome the complexity of implementing queues. For instance, in [6], Henzinger et al. formalize the definition of the `c-out-of-order` queue where an element at a distance up to $c-1$ from the element in the head of the queue, is allowed to be dequeued. A linearizable and lock-free c-out-of-order queue with no concurrency constraints is implemented in [11] using the CAS primitive. In [1], a lock-free implementation of a queue with `multiplicity` where only concurrent *Dequeue* operations can return the same element, is given under the coherence condition of set-linearizability. This implementation has no concurrency constraint and uses only $Read/Write$ primitives. In both these implementations, the *Dequeue* operation's worst-case step complexity is unbounded since it depends on the number of *Enqueue* operations executed. Regarding practical applications, [2] discusses possible applications of the multiplicity relaxation such as relaxed work-stealing for parallel SAT solvers.

Simply by considering an execution where a process only executes *Enqueue* operations, we can show a lower bound on space complexity in the number of elements present in the queue. However, besides this space requirement, there has been some work in optimizing the space complexity of queue implementations using memory reclamation (e.g. [3, 16]). We do not consider the issue of optimizing the space complexity and leave the question for future work.

Paper organization. In Section 2 we present the model. In Section 3, we describe our linearizable wait-free multiple enqueuer multiple dequeuer queue implementation together with its correctness proof. Finally, we present the relaxed queue implementation with multiplicity in Section 4.

2 Preliminaries

We consider a standard asynchronous shared memory model, consisting of a set $\mathcal{P}$ of n crash-prone processes with unique ids, where all n processes can be enqueuers and $k \leq n$ can be dequeuers. We also refer to this set of processes as a set of n enqueuers and k dequeuers.

Processes communicate by applying primitive operations to shared base objects. In particular, we consider *registers*, $Fetch\&Inc$, *Compare&Swap*, and *Max registers*. A *register* provides atomic $Read/Write$ primitives. The $Fetch\&Inc$ object provides a $Fetch\&Inc$ primitive that increments the value of the object by 1 and returns the previous value. The $Compare\&Swap$ object supports the *Read* and the CAS primitives. The *Read* simply returns the value of the object. The call to $CAS(old, new)$ writes *new* into the object only if the current value of the object is equal to *old* and in that case, it returns $True$, otherwise, it returns $False$.

The *max register* supports two primitives : $MaxWrite(v)$ that writes the value v into the register, and $MaxRead()$ that returns the largest value written so far. Modern architectures do not implement the max register object. However, our algorithm uses max registers in a restricted way (essentially, each new value written increments the previous value by one), thus we can easily implement the $MaxWrite(v)$ and $MaxRead()$ operations by applying a constant number of primitives on CAS objects.

The FIFO *queue* provides the two high-level operations $Enqueue(v)$ and $Dequeue()$. An $Enqueue(v)$ operation adds the element v at the tail of the queue, while the $Dequeue()$ operation removes the element at the head of the queue and returns its value, if the queue is not empty, otherwise it returns a special value ϵ.

An *implementation* of a shared object provides a specific data-representation for the object from a set of *base objects*, each of which is assigned an initial value; the implementation also provides algorithms for each process in $\mathcal{P}$ to apply each operation to the object being implemented. To avoid confusion, we call operations on the base objects *primitives* and reserve the term *operations* for the FIFO queue object being implemented.

An *execution* of an implementation of a shared object is a sequence of steps (possibly infinite), where a step is either the application of a primitive operation on a base object or an invocation/response of an operation of the high-level implemented object. An execution is *well-formed* if each process is sequential and if it invokes a new high-level operation only after it has completed the current one. The steps taken by a process during the execution of a high-level operation are defined by the algorithms provided by the implementation of the shared object.

If an operation op_1 returns before another operation op_2 is invoked, we say that op_1 precedes op_2 in real-time order, denoted $op_1 <_{ro} op_2$.

Roughly speaking, an implementation is *linearizable* [8] if each operation appears to take effect atomically at some point between its invocation and response; it is *wait-free* [7] if each process completes its operation if it performs a sufficiently large number of steps.

To define the relaxed FIFO queue, we consider the formalism of *set-linearizability* provided in [15]. Roughly speaking, set-linearizability allows for multiple concurrent operations to be linearized at the same point. Such a linearization point would fall within the execution interval of all the concurrent operations. The set-linearization of an execution E is defined by ordering different sets of the operations in E, such that the operations in a set are executed concurrently. The *FIFO queue with multiplicity* [1] is a relaxed FIFO queue such that its specification allows multiple concurrent *Dequeue*() operations to return the same value.

3 Wait-Free Linearizable Queue

3.1 Algorithm overview

We present hereafter a conceptual overview of the algorithm implementing the k-dequeuer n-enqueuer concurrent queue.

As depicted in Figure 1, the queue object can be seen as n different *sub-queues* such that when an *Enqueue*(v) operation is invoked by an enqueuer process p, the element v is enqueued in the corresponding p-th *sub-queue*. Each enqueued element is associated with a unique timestamp, used by the dequeuers to select the element to be returned (if any).

In particular, each enqueued element is associated to a pair (st, p) where st is the value of a shared counter, and p is the *id* of the process that invoked the corresponding *Enqueue* operation. Two processes executing concurrent *Enqueue*(v) operations can retrieve the same value from the shared counter, but the process *id* makes each timestamp unique. Timestamps are totally ordered according to the lexicographical order. The timestamps associated with the elements in a given sub-queue reflect the real-time order of *Enqueue*() operations by the same process. In particular, if an element e is enqueued in a sub-queue p before another element e', then e is associated with a smaller timestamp than e'. This also means that the head of the sub-queue has the smallest timestamp among the other elements in the same sub-queue.

For the sake of complexity, the timestamps are organized in a tree structure where the n leaves correspond to the timestamps of the elements at the head of the corresponding n sub-queues, and the root stores the smallest timestamp among the ones in the leaves. Our construction is similar to the one proposed by Jayanti et al. in [9].

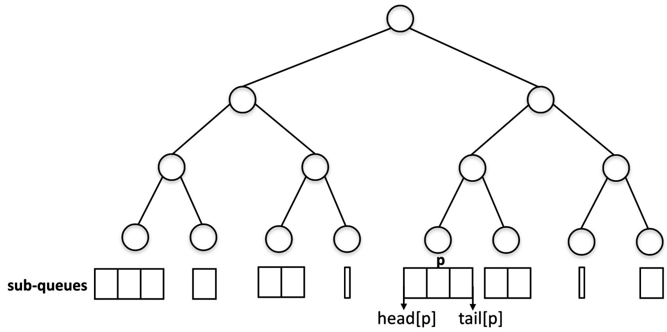

Figure 1 Data structure for the k-dequeuer n-enqueuer queue implementation.

Thus, a *Dequeue* operation simply reads the root of the tree and returns the corresponding element in the appropriate sub-queue in the same manner that this is done in the *single dequeuer* queue in [9]. However, to support k different dequeuer processes, we need to manage the concurrency between their operations. This is done by introducing a helping mechanism for the *Dequeue* operation. In particular, each *Dequeue* operation has a unique sequence number. Before executing its instance of *Dequeue* operation, a process will first ensure that the instances with smaller sequence numbers are not more pending. If they are, the process will execute the steps necessary for them to finish, and it will update the tree before executing its own instance of *Dequeue*. Since there are k dequeuer processes, during an instance of *Dequeue*, there could be at most $k-1$ other processes executing a *Dequeue* operation concurrently.

3.2 Algorithm Pseudocode

In the implementation of the multiple dequeuer and multiple enqueuer queue in Algorithms 1–2, we use two main data structures: a two-dimensional array of registers, called *items*, where each row p together with two integers $head[p]$ and $tail[p]$ represents the sub-queue of process p; and a balanced binary tree T with n leaves where each node is a *CAS* object used to stores the timestamps of enqueued elements.

The sub-queue p contains the elements enqueued by process p that have not been dequeued, i.e. the current sub-queue p is defined by its values h and t of the max register $head[p]$ and the register $tail[p]$ respectively. If $h = t$, the sub-queue p is empty. Otherwise, it is the ordered list of $t-h$ elements : $items[p][h], \cdots, items[p][t-1]$.

Each *Enqueue* operation executed by process p is associated with a unique timestamp (st, p) where st is an integer obtained from the counter $enqCounter$, and p is the process id. The empty queue is associated with a special timestamp $(\epsilon, -1)$, and we consider that $\epsilon > i$ $\forall i \in \mathbb{N}$. $items[p][i] = (val, (st, p))$ means that the i-th *Enqueue* operation by p has enqueued the value val, and that this *Enqueue* has the timestamp (st, p).

The smallest timestamp of a sub-queue p is the timestamp value of $items[p][h]$ where h is the current value of the head of the sub-queue. This timestamp is stored in the p-th leaf of the tree T associated with p, called p-leaf. The following details the different functions of the implementation in Algorithms 1–2.

- *Enqueue*(v): when process p calls an instance of *Enqueue*(v), it starts by constructing the corresponding timestamp (st, p) by reading the value of $enqCounter$. Process p will then write $(v, (st, p))$ to $item[p][t]$ where t is the value of $tail[p]$. Then, it updates the value of $tail[p]$ to $t+1$. Afterward, the value $st+1$ is written to the max register $enqCounter$ to

Algorithm 1 Wait-free queue implementation (pseudo-code for process p).

```
Shared variables
    enqCounter : Max register object, initially 0.
    deqCounter :Fetch&Inc object, initially 1.
    head[n] : Array of Max register objects, initially 0.
    tail[n] : Array of registers where each register contains an integer, initially 0.
    items[n][···] : Two dimensional array of registers, each register contains the uplet
      (val, (st, it)) initially (⊥, (⊥, ⊥)).
    T : binary tree of CAS objects with n leaves, each node contains the pair (st, id), all
      initially (ϵ, −1).
    deqOps[···] : Array of CAS objects, initially (⊥, ⊥). deqOps[j] = (i, id) means that
      the j-th Dequeue operation returns items[id][i].val if id ≠ −1, otherwise the
      operation returns ϵ.

Function Enqueue(v)
    st ← enqCounter.MaxRead()
    t ← tail[p]
    items[p][t] ← (v, (st, p))
    tail[p] ← tail[p] + 1
    enqCounter.MaxWrite(st + 1)
    Propagate(p)
    return True

Function Dequeue()
    num ← deqCounter.Fetch&Inc()
    for (i ← max(1, num − k + 1); i ≤ num; i + +) do
        if deqOps[i].Read() = (⊥, ⊥) then
            if i > 1 then
                UpdateTree(i − 1)
            FinishDeq(i)
    (j, id) ← deqOps[num].Read()
    if id = −1 then
        return ϵ
    else
        (ret, −) ← items[id][j]
        return ret
```

ensure that all subsequent *Enqueue* operations will have a greater timestamp than (st, p). Finally, process p calls $Propagate(p)$ to update the timestamps in the nodes of the tree T from the p-leaf to the root, if necessary.

- *Refresh*(*node*, *isLeaf*): this function is invoked during the execution of an instance of *Propagate* to reset the timestamp stored in a *node*. If the boolean *isLeaf* is equal to $True$, the current node represents a leaf of the tree T. In this case, the operation computes the minimum timestamp in the corresponding sub-queue. This value is either (1) $(\epsilon, -1)$ if the sub-queue is empty (line 16 of Algorithm 2); or a timestamp (2) (st', i)(line 18 of Algorithm 2). If $isLeaf = False$ then *node* is not a leaf; the operation reads the

Algorithm 2 Auxiliary functions to the queue implementation.

```
Function Propagate(id)
    currentNode ← leaf(T, id)
    if !Refresh(currentNode, True) then
        Refresh(currentNode, True)
    do
        currentNode ← parent(currentNode)
        if !Refresh(currentNode, False) then
            Refresh(currentNode, False)
    while currentNode ≠ root(T)

Function Refresh(node, isLeaf)
    (st, id) ← node.Read()
    if isLeaf then
        h ← head[id].MaxRead()
        t ← tail[id]
        if h = t then
            ret ← node.CAS((st, id), (ϵ, −1))
        else
            (−, (st′, −)) ← items[id][h]
            ret ← node.CAS((st, id), (st′, id))
        return ret
    else
        (min_st, min_id) ←read minimum  timestamp in current node's children
        return node.CAS((st, id), (min_st, min_id))

Function FinishDeq(num)
    (−, id) ← root(T).Read()
    if id = −1 then
        deqOps[num].CAS((⊥, ⊥), (ϵ, −1))
    else
        h ← head[id].MaxRead()
        deqOps[num].CAS((⊥, ⊥), (h, id))

Function UpdateTree(num)
    (j, id) ← deqOps[num].Read()
    if id ≠ −1 then
        head[id].MaxWrite(j + 1)
        Propagate(id)
```

timestamps stored in the children of the current node to compute the minimal timestamp. Then, in both cases, the operation executes the *CAS* primitive on *node* to write the timestamp and returns the resulting boolean.

- *Propagate*(id): updates the nodes of the tree T in the path from the id-leaf node to the root. Specifically, the function relies on calls to *Refresh* while traversing the path to update each individual *node*. To ensure that the value written into a node is up to date, the call to the function *Refresh*($node$, $-$) is repeated if the first call fails because a concurrent instance r_1 of *Refresh*$(node, -)$ might have written an outdated value since r_1 started before the call to *Refresh*$(node, -)$ in *Propagate*(id). However, after the second call to *Refresh*$(node, -)$, we are certain that the value written is up to date because it can only be written by an instance invoked after *Propagate*(id). This technique is used in the implementation of the single dequeuer multiple enqueuer queue in [9].

- *Dequeue*: First, an instance of the *Dequeue* operation executed by a process p, computes its unique sequence number *num* by applying a $Fetch\&Inc$ primitive on $deqCounter$. Then, p executes the helping mechanism to assist any pending *Dequeue* operation with a sequence number $i \in [max(1, num - k + 1), num])$ in increasing order of i. If the operation with the index i is still pending (i.e. $deqOps[i]$ is still set to its initial value), p executes $UpdateTree(i - 1)$ if $i > 1$, to ensure that the root of the tree is updated to an accurate value. Then, p executes $FinishDeq(i)$ to decide on the operation's return value in $deqOps[i]$. After the return values have been decided for all *Dequeue* operations with indexes in $[max(1, num - k + 1), num])$, p reads $deqOps[num] = (i, j)$ and returns $items[j][i].val$, otherwise p returns ϵ.
- *FinishDeq*(num): The array *DeqOps* stores the information regarding the return values of each *Dequeue* operation. A call to *FinishDeq* with the parameter *num* decides a value and attempts to write it to *DeqOps*$[num]$ using a CAS primitive.FinishDeq(num) reads the timestamp at the root of the tree $T : (-, id)$. And if $id = -1$ (i.e. the queue is empty), then $(\epsilon, -1)$ is written to *DeqOps*$[num]$. Otherwise, the value (h, id) is written to *DeqOps*$[num]$ where h is the value of the head of the sub-queue *id*. In either scenario, if the CAS instruction fails, another process has succeeded in executing a CAS instruction on *DeqOps*$[num]$ and the return value for the corresponding *Dequeue* has been decided.
- *UpdateTree*(num): A simple function call that encapsulates the steps necessary before executing the *Dequeue* operation with the sequence number $num + 1$. If the *Dequeue* operation with the sequence number *num* returns ϵ, then there are no additional steps necessary. Otherwise, it is necessary to update the head of the sub-queue *id* from which the return value was retrieved; followed by a call to the function *Propagate*(id) to update the tree accordingly.

3.3 Proof

In this section, we establish that Algorithms 1–2 are a wait-free implementation of a k-dequeuer multi-enqueuer queue. We also establish that an *Enqueue* operation has a worst-case step complexity of $O(\log n)$ and a *Dequeue* operation has a worst-case step complexity of $O(k \log n)$.

3.3.1 Algorithm properties

Each *Dequeue* operation is associated with a unique sequence number that is the value obtained by applying the $Fetch\&Inc$ primitive on $deqCounter$ at line 18 of Algorithm 1.

▶ **Lemma 1.** *A total order between Dequeue operations is provided by their sequence number. This order respects the real-time order.*

Proof. Let deq_1 and deq_2 be two *Dequeue* operations by process p_1 and p_2 respectively. Let seq_1 be the sequence number of deq_1 and seq_2 be the sequence number of deq_2. We prove that if deq_1 precedes deq_2 in real-time order, then $seq_1 < seq_2$.

deq_1 completes before deq_2 is invoked, thus p_1 executes line 18 of Algorithm 1 before the invocation of deq_2 by p_2. The proof follows from the fact that $deqCounter$ is a linearizable $Fetch\&Inc$ object. ◀

The *Dequeue* operation with the sequence number i is complete at a given configuration C if $DeqOps[i] \neq (\perp, \perp)$ (i.e.; the value of $DeqOps[i]$ at C is not the initial value). Otherwise, it is incomplete at C.

▶ **Observation 2.** *Let deq denote a Dequeue operation with the sequence number i. Any call to $FinishDeq(i)$ is executed after the invocation of deq.*

▶ **Lemma 3.** *Fix an execution E and let C be any configuration of E. $\forall h > 0$ and $\forall i \geq 1$, if the $h+i$-th Dequeue operation exists and it is complete at C, then the i-th Dequeue operation is complete at C.*

Proof. Consider the first configuration C where the $h+i$-th *Dequeue* operation is complete, i.e.; $deqOps[i+h] \neq (\perp, \perp)$. Assume by contradiction that $deqOps[i]$ has its initial value at C.

The value of $deqOps[i]$ is only set during the execution of $FinishDeq(i)$ at line 30 or 27 of Algorithm 2. According to the condition in the for-loop (line 19 of Algorithm 1), only a *Dequeue* operation with a sequence number $i + h \leq l \leq i + h + k - 1$ may change the value of $deqOps[i+h]$.

According to Lemma 1, the *Dequeue* operations with a sequence number smaller than or equal to l, and in particular $\in [i, l]$, have started at the configuration immediately before the value of $deqOps[i+h]$ is changed by the l-th *Dequeue* operation. Also, the *Dequeue* operations with a sequence number $num \in [i, i+k-1]$ could not have returned at C otherwise $deqOps[i] \neq (\perp, \perp)$ at C (contradicting our assumption). This is trivially true for $num = i$. For $num \in [i+1, i+k-1]$, and since the condition at line 20 of Algorithm 1 is *true* for $deqOps[i]$, the *Dequeue* operation with sequence number num will execute the $FinishDeq(i)$ function and set $deqOps[i] \neq (\perp, \perp)$ before it returns.

Thus, l should be greater than $i + k - 1$. But this means that there are $k + 1$ pending *Dequeue* operations, which contradicts the fact that we can have at most k pending *Dequeue* operations. There is a contradiction. ◀

As $deqOps[num]$ is updated only during the execution of the function $FinishDeq(num)$; the following observation is a consequence of Lemma 3.

▶ **Observation 4.** *Before the first execution of $FinishDeq(i+h)$, $FinishDeq(i)$ has been executed.*

Each *Enqueue* operation *op* has a unique timestamp composed of an integer obtained by reading the Max register *enqCounter* during the execution of line 10, and the *id* of the process that executed the operation *op*.

▶ **Observation 5.** *For each p, the timestamps of the elements written in the sub-array $items[p]$ are monotonically increasing in accordance with their index in the array. In other terms, we have $items[p][i].ts < items[p][i+1].ts$.*

At any given configuration, the sub-queue of process p is the sub-array of $items[p]$ in the range $items[p][head[p].MaxRead()], ..., items[p][tail[p]-1]$.

▶ **Lemma 6.** *Let enq_1 and enq_2 be two Enqueue operations such that enq_1 ends before enq_2 is invoked. Let (st_1, id_1) be the timestamp of enq_1 and (st_2, id_2) be the time stamp of enq_2. We have $st_1 < st_2$.*

Proof. After the execution of line 14 of Algorithm 1 during enq_1, any value returned by a `enqCounter`.$MaxRead$ is greater or equal to $st_1 + 1$. The claim follows from the fact that enq_2 executes line 10 of Algorithm 1 after enq_1 returned. ◀

We say that the i-th *Enqueue* operation by a process p matches the *Dequeue* operation with sequence number j, if $deqOps[j] = (i, p)$ at some point in the execution.

Meaning, if the *Dequeue* operation returns, it returns the element enqueued by the i-th *Enqueue* operation of process p (i.e. `items[p][i]`).

▶ **Lemma 7.** *An Enqueue operation has at most a single matching Dequeue operation.*

Proof. Let enq be the i-th *Enqueue* operation by a process p. Assume by contradiction that there are two *Dequeue* operations, deq_1 and deq_2 that match enq. Let j_1 and j_2 be their corresponding sequence numbers. Then, $deqOps[j_1] = deqOps[j_2] = (i, p)$. By Lemma 1 and without loss of generality, let $j_1 < j_2$. Because of the Observation 4, $FinishDeq(j_1)$ returned before $FinishDeq(j_2)$ is invoked. According to lines 22 to 23 of Algorithm 1, $UpdateTree(j_1)$ is executed before $FinishDeq(j_1 + 1)$. This means that the value $i + 1$ is written in the Max register $head[p]$ at line 34 before that a process read it during the $FinishDeq(j_1 + 1)$. And since $j_2 \geq j_1 + 1$, the claim follows. ◀

▶ **Lemma 8.** *Let enq denote the i-th Enqueue operation by a process p. Let $ts = (st, p)$ be the timestamp of enq. Let s be any node in the tree T in the path from the p-th leaf to the root of the tree. At any configuration C after enq ends and such that $deqOps[j] \neq (i, p)$ for each $j \geq 0$, we have that the timestamp stored at s is smaller than or equal to ts at C.*

Proof. After enq, we have that $tail[p] \geq i + 1$, because enq is the i-th *Enqueue* operation executed by p.

We first prove that after enq, $head[p]$ is smaller than or equal to i as long as $deqOps[l] \neq (i, p)$ for any $l \geq 0$.

The value of $head[p]$ is updated only during the execution of the function $UpdateTree$ (line 34 of Algorithm 2). In particular, the value of $head[p]$ is set to a value $j + 1$ where j is the value read from some $deqOps[num]$ at line 32. Also, the value of $deqOps[num]$ is updated only during the execution of the function $FinishDeq(num)$ with a value read from $head[p]$ (lines 29 and 30). We prove by induction on j that if the value written in $head[p]$ is j then, all values $0, \ldots j - 1$ have been previously written in $head[p]$ (in increasing order) and to some $deqOps[num]$. The base case is for $j = 1$. Consider the first MaxWrite() that writes 1 to $head[p]$ and let q be the process applying this primitive. According to line 34, q has read the value $(0, p)$ from some $deqOps[num]$, which has been updated with a value read from $head[p]$. The claim follows.

Suppose this is true for a value j, we show that the claim holds for $j + 1$. Consider the first process, denoted q, that writes $j + 1$ into $head[p]$. q has read (j, p) from some $deqOps[num]$ at line 32. By inductive hypothesis, and by the linearizability of $head[p]$ all the values $0, \ldots j$ have been written in $head[p]$ and all the values $0, \ldots j - 1$ have been written in some $deqOps[num]$. The claim follows.

Hence, $head[p] \leq i$ as long as for any $l \geq 0$, we have $deqOps[l] \neq (i, p)$. This is because to write the value $i + 1$ (and then any greater value), a process has to read $deqOps[l] = (i, p)$ for some l.

Base case $k = 0$. s is the p-th leaf. Since enq completes, there is at least one instance of $Propagate(p)$ performed after that process p has written the value i in $tail[p]$. The value of $head[p]$ is smaller than or equal to i, so any instance of $Propagate(p)$ that changes the value of s before C, will write a timestamp read in $items[p][j]$ for some $j \geq i$. By Observation 5, the timestamp read is smaller than or equal to $ts = (st, p)$.

It remains to prove that after an instance of $Propagate(p)$ completes, denoted *prop*, a value smaller than or equal to i has been written in the leaf corresponding to p. An instance of $Propagate(p)$ performs two $Refresh(s)$. Each $Refresh(s)$ reads the state of s, then the $head[p]$ and the corresponding timestamp ts and then applies a CAS to s to modify its value with ts. Suppose that both $Refresh(s)$ fail (and in particular the second one), otherwise the claim is trivial. The second $Refresh(s)$ fails because another an instance of $Propagate(p)$, denoted $prop'$ successfully applied a CAS on s. But $prop'$ has read $head[p]$ after $tail[p]$ is set to i. Meaning that it has read a value smaller than or equal to i and it writes in s the corresponding timestamp that is smaller than or equal to ts.

Induction case $k+1 \leq \log n$. Suppose that the claim holds for $j \leq \log n$: the timestamp stored at s_j is smaller than or equal to ts where s_j is in the path from the p-th leaf to the root at a height of $j \leq k$. We prove that the claim holds for the parent of s_j, denoted s_{j+1}.

Any instance of $Propagate(p)$ updates the nodes in the path from the p-th leaf to the root, one by one, starting from the leaf and following the path to the root. Also, immediately after *enq* completes, there is at least one $Propagate(p)$ instance that passed through all the nodes in this path. Consider, the first $Propagate(p)$ that updated node s_{j+1} after s_j has been updated, denoted *prop*.

Observe that any process that executes the $Refresh$ function on node s_{j+1} writes the minimum timestamp it reads from the children of s_{j+1}. And that the second $Refresh(s_{j+1})$ fails only if another $Propagate(p)$ has modified the state of this node with a value smaller than or equal to the value at s_j read by *prop*. ◀

▶ **Lemma 9.** *Let enq be an Enqueue operation with the timestamp ts that enqueued* $items[p][i]$*. If* (i,p) *was written to* $deqOps[j]$ *by a process q, then the execution of line 25 of Algorithm 2 to read ts by q was executed after the invocation of enq.*

Proof. *enq* is the i-th enqueue operation by p. Let *deq* be the *Dequeue* operation executed by q that retrieves ts from the root of the tree (Line 25 of Algorithm 2) before writing (i,p) to $deqOps[j]$. *enq* must execute the line 13 of Algorithm 1 before ts can be propagated in the tree according to the code of function $Refresh$. The claim follows. ◀

▶ **Lemma 10.** *Let* enq_1 *and* enq_2 *be two Enqueue operations such that* enq_1 *ends before* enq_2 *is invoked. If* enq_2 *has a matching Dequeue operation* deq_2*, then* enq_1 *also has a matching Dequeue operation* deq_1*.*

Proof. By contradiction, we suppose that deq_2 exists and deq_1 does not. We denote ts_1 and ts_2 the timestamps associated with enq_1 and enq_2 respectively and num_2 the sequence number of deq_2. From Lemma 6, $ts_1 < ts_2$ because enq_1 ends before enq_2 begins.

And since enq_1 does not have a matching *Dequeue*, there is no $j \geq 0$ such that $deqOps[j] = (i,p)$ where $items[i][p]$ is enqueued by enq_1. Therefore, from Lemma 8, for any node s in the path in T from the p-th leaf to the root, the timestamp stored at s is smaller than or equal to ts_1. In particular, for the root of the tree, the timestamp stored is smaller or equal to ts_1. From Lemma 9, the step of line 25 of Algorithm 2 to read the root of the tree before writing $deqOps[num_2]$ is executed after the invocation of enq_2 which is after the invocation of enq_1. Meaning that during this step, the timestamp at the root was smaller or equal to ts_1 contradicting the fact that $ts_1 < ts_2$. ◀

▶ **Lemma 11.** *Let* enq_1 *and* enq_2 *be two Enqueue operations such that* enq_1 *ends before* enq_2 *is invoked and let* deq_1 *and* deq_2 *be the matching Dequeue operations to* enq_1 *and* enq_2 *respectively. We have that* deq_1 *has a lower sequence number than* deq_2*.*

Proof. We denote num_1 and num_2 the sequence numbers of deq_1 and deq_2 respectively, and ts_1 and ts_2 the timestamps of enq_1 and enq_2 respectively. By contradiction, we suppose that $num_1 > num_2$. Since enq_1 ends before enq_2 begins we have that $ts_1 < ts_2$ (Lemma 6).

And since $deqOps[i]$ are written in an increasing order of i according to Lemma 3, we have that $deqOps[num_2]$ is written before $deqOps[num_1]$. However, from Lemma 8, as long as $deqOps[num_1]$ has its initial value, then the timestamp stored at the root is smaller than or equal to ts_1. At the execution of line 25 of Algorithm 2 to compute the final value of $deqOps[num_2]$, the root has a timestamp smaller or equal to ts_1; contradicting the fact that $ts_1 < ts_2$. ◀

▶ **Lemma 12.** *Let deq be a Dequeue operation and let enq be an Enqueue operation that ends before deq is complete. Let C be a configuration of E where enq does not have a matching Dequeue operation deq' or deq' is not complete at C. If deq is complete at C, then deq does not return ϵ.*

Proof. By contradiction, we suppose that deq returns ϵ. Let i denote the sequence number of deq and ts denote the timestamp of enq. Since deq returns ϵ, deq reads the value $(\epsilon, -1)$ in $deqOps[i]$ at line 24 of Algorithm 1. Therefore, during the execution of $FinishDeq(i)$, the process that writes $deqOps[i]$, reads $(\epsilon, -1)$ at the root of the tree (line 27 of Algorithm 2). However, By Lemma 8, the timestamp at the root of the tree after the end of enq is smaller than or equal to ts. Meaning that during the execution of line 25 of Algorithm 2 during the instance $FinishDeq(i)$ that writes $deqOps[i]$, the timestamp at the root of the tree was smaller than or equal to ts. We reach a contradiction because $(\epsilon, -1)$ is larger than any timestamp $(h, -)$ $\forall h \in \mathbb{N}$. ◀

3.3.2 Linearizability

First, we construct a permutation L of some of the *Dequeue*() and *Enqueue*() operations invoked such that L contains all operations that have terminated. Then, we prove that L preserves the real order as well as the semantics of a queue.

3.3.2.1 Linearization definition

Let E denote a given execution of the wait-free queue implemented in Algorithm 1 and Algorithm 2. We classify every *Dequeue*() operation deq that appears in E to exactly one of the following types :

1. deq does not execute line 18 of Algorithm 1 in E. Thus deq is not attributed a sequence number.
2. deq executes line 18 of Algorithm 1 in E, its sequence number is j and `deqOps[j]` has the initial value $(\perp, \perp)$ in E.
3. deq executes line 18 of Algorithm 1 in E, its sequence number is j and `deqOps[j]` $\neq (\perp, \perp)$ in E.

We remove from E, any *Dequeue*() operation of type 1 and 2. We denote DEQ the set of *Dequeue*() operations of type 3. Each operation in DEQ is associated with a unique sequence number $j \in \mathbb{N}_0$. We totally order all the operations in DEQ according to their sequence number. Also, let deq be any incomplete *Dequeue*() operation in DEQ and let j be its sequence number. We complete deq by returning the value v if `deqOps[j]` $= (i, id)$ in E and `items[id][i]` $= (v, -)$. Otherwise, we complete deq by returning the empty queue value ϵ.

We remove every *Enqueue*() operation that does not execute line 13 of Algorithm 1 in E. We denote ENQ the set of *Enqueue*() operations that appear in E and that we do not remove. Every *Enqueue*() operation enq in ENQ is uniquely identified by a pair (i, id) meaning that enq is the i-th *Enqueue*() operation performed by the process id. We associate the *Dequeue*() operation in DEQ with sequence number i with the *Enqueue*() operation (j, id) such that `deqOps[i]` $= (j, id)$.

Let ENQ_d denote the *Enqueue*() operations in ENQ that have an associated *Dequeue*() operation in DEQ. We associate each *Enqueue*() operations in ENQ_d with the sequence number of the corresponding *Dequeue*(). Thus, *Enqueue*() operations in ENQ_d are totally ordered according to the given sequence number.

We construct the linearization L of the operations in E as follows:

1. First we insert the *Enqueue*() operations in ENQ_d one by one and according to their total order, denoted $enq_{i_1}, enq_{i_2} \ldots$ in L. Notice that enq_{i_h} is the *Enqueue*() operation associated with the *Dequeue*() operation having the sequence number i_h. Assuming that $enq_{i_{h+1}}$ exists, we have $i_h < i_{h+1}$; and all the *Dequeue*() operations having a sequence number $i \in [i_h + 1, i_{h+1} - 1]$ return the value ϵ.
2. Then, we insert the *Dequeue*() operations one by one according to their the sequence number. For any sequence number k, If deq_k returns ϵ it is inserted immediately after deq_{k-1} if it exists, or at the beginning otherwise. In the case where deq_k does not return ϵ, it is linearized immediately after the furthest point in L following: (i) the previous deq_{k-1}, (ii) the matching *Enqueue* operation enq_{i_l} with $i_l = k$, and (iii) the last *Enqueue* operation that ends before the invocation of deq_k.
3. Let enq denote an *Enqueue* operation from the remaining *Enqueue*() operations with no matching *Dequeue* operations (i.e. $ENQ \setminus ENQ_d$). We insert enq after the last operation in ENQ_d and before the first *Dequeue*() operation that starts after enq ends (or at the end of L if such *Dequeue*() does not exist). If multiple operations from $ENQ \setminus ENQ_d$ are linearized at the same point, then they are ordered according to their real-time order.

For two operations op_1 and op_2, we denote $op_1 <_L op_2$ when op_1 precedes op_2 in the linearization L.

3.3.2.2 Linearization and real-time order

We show that the linearization defined in the previous section respects the real-time execution order.

▶ **Lemma 13.** *Let op_1 and op_2 be two Enqueue operations in E such that op_1 ends before op_2 is invoked. op_1 precedes op_2 in L.*

Proof. First, consider the case where both operations do not have matching *Dequeue*() operations. From linearization rule 3, an *Enqueue* operation that does not have a matching *Dequeue* operation is linearized before the first *Dequeue* operation that starts after it ends or at the end of L if such *Dequeue* operation does not exist. If op_1 is linearized at the end of L, then op_2 is also linearized at the end of L after op_1, because op_2 starts after op_1 ends and there is no *Dequeue* operation that starts after op_1 ends. We suppose that there exists a *Dequeue* operation deq_1 such that op_1 is linearized immediately before deq_1. If op_2 is linearized at the end of L, the claim is trivial. So let deq_2 be a *Dequeue* operation such that op_2 is linearized immediately before deq_2. We have $op_1 <_{ro} op_2 <_{ro} deq_2$. Meaning that $deq_2 = deq_1$ or $deq_1 <_L deq_2$, because both operations start after op_1 ends, and deq_1 is the first such operation in L. Therefore, $op_1 <_L op_2$ according to their real time execution order following linearization rule 3.

Next, if op_1 has a matching *Dequeue*() operation but op_2 does not, we have that op_2 is linearized after the last linearized *Enqueue*() operation that has a matching *Dequeue*() operation. The case where op_1 does not have a matching *Dequeue*() operation but op_2 does, is impossible according to Lemma 10. We suppose that both op_1 and op_2 have matching *Dequeue*() operations, named respectively deq_1 and deq_2. From Lemma 11, we have that deq_1 has a smaller sequence number than deq_2. Therefore, from linearization rule 1, op_1 is linearized before op_2. ◀

▶ **Lemma 14.** *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. If enq has a matching Dequeue operation deq', then the sequence number of deq' is greater than j.*

Proof. We denote i the sequence number of deq'. By contradiction we suppose that $j > i$. We consider the configuration C where *deq* completes. According to Lemma 3, deq' also has been completed at C. Meaning that $deqOps[i] \neq (\bot, \bot)$ at C. However, from the hypothesis, *enq* has not started at C, as *enq* is not invoked until *deq* finishes. According to Lemma 9, deq' cannot match *enq*. The claim follows. ◀

▶ **Lemma 15.** *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. If enq has a matching Dequeue operation deq', then any Dequeue operation with a sequence number $l < j$ is linearized before enq.*

Proof. By contradiction, we suppose that there exists *Dequeue* operations with sequence numbers strictly smaller than j that are linearized after *enq*, and let deq_l be the first of these operations in L. Thus , if deq_{l-1} exists, we have that $deq_{l-1} <_L enq$.

If deq_l returns ϵ, from linearization rule 2, deq_l is linearized immediately after deq_{l-1} if it exits, or at the beginning of L. Therefore, $deq_l <_L enq$. There is a contradiction.

Otherwise, deq_l has a matching *Enqueue* operation denoted enq_l. We denote i the sequence number of deq'. From Lemma 14, we have that $j < i$. Therefore, $l < j < i$. Thus, $enq_l <_L enq$ from linearization rule 1. Furthermore, we have $deq_{l-1} <_L enq$ (if it exists). Therefore, since $enq_l <_L enq$ and $deq_{l-1} <_L enq$, according to linearization rule 2, $enq <_L deq_l$ because $enq <_{ro} deq_l$ (rule 2.3 of linearization) . Consequently, $deq_j <_{ro} enq <_{ro} deq_l$. Contradicting the fact that $l < j$ (Lemma 1). ◀

▶ **Theorem 16.** *Let op_1 and op_2 be two operations in E such that op_1 ends before op_2 is invoked. Then, op_1 precedes op_2 in L.*

Proof. Four cases have to be studied according to the type of operations.

1. op_1 and op_2 are two *Dequeue*() operations. Since op_1 ends before op_2 begins, the sequence number i_1 of op_1 is strictly smaller than the sequence number i_2 of op_2 (Lemma 1). From linearization rule 2, we have op_1 is before op_2 in L.
2. The case where op_1 and op_2 are *Enqueue*() operations is proved by Lemma 13.
3. op_1 is an *Enqueue* operation and op_2 is a *Dequeue*() operation. First, consider the case that op_2 does not return ϵ. If $op_1 \in ENQ_d$, then from linearization rule 2, op_2 is linearized after op_1 because op_2 is inserted after the last *Enqueue* operation that ends before op_2 starts. Otherwise, If $op_1 \notin ENQ_d$, from linearization rule 3, it is linearized before the first *Dequeue* operation that starts after op_1 ends. Thus op_1 is linearized before op_2.
   Next, consider the case where op_2 returns ϵ, and let i denote its sequence number. By Observation 2 and Lemma 12, op_1 has a matching *Dequeue* operation *deq*, and *deq* is complete before op_2 is complete.

Let j is the sequence number of *deq*. Since *deq* is complete before op_2 is complete, by Lemma 3, we have that $j < i$. Therefore, from linearization rule 2, *deq* is linearized before op_2. Thus, from linearization rule 1, $op_1 <_L deq <_L op_2$. The claim follows.

4. Finally, we suppose that op_1 is a *Dequeue* operation and that op_2 is an *Enqueue* operation. If op_2 does not have a matching *Dequeue* operation, from linearization rule 3, it is linearized before the first *Dequeue* operation that starts after op_2 ends or at the end of L if such operation does not exist. Thus, op_2 is linearized after op_1 because op_1 ends before op_2 starts.
   So consider that op_2 has a matching *Dequeue* operation *deq* and let i be its sequence number and j be the sequence number of op_1.
   If op_1 returns ϵ, from the linearization rule 2, we have $op_1 = deq_j$ is linearized immediately after deq_{j-1} (or beginning of L if it does not exist). And from Lemma 15, for each $l < j$, we have that deq_l is linearized before op_2. In particular, we have that deq_{j-1} is linearized before op_2. Therefore, op_1 is linearized before op_2.
   Otherwise, consider enq_j the matching operation of op_1. From linearization rule 2, op_1 is linearized after (i) deq_{j-1}, (ii) enq_j and after (iii) the last *Enqueue* enq' that ends before op_1 starts. We show that op_2 is linearized after all these three operations. From Lemma 15, we have that deq_{j-1} is linearized before op_2 (i). From Lemma 14, we have that $j < i$ meaning that enq_j is linearized before op_2 according to the total order of the sequence numbers of their matching *Dequeue* operations (ii). And since op_1 ends before op_2 starts, $enq' <_{ro} op_2$. Therefore, $enq' <_L op_2$ because we have shown that the linearization of the *Enqueue* operations respects the real time execution order (Lemma 13) (iii). The claim follows. ◀

3.3.2.3 Linearization and the Queue Sequential Specification

▶ **Lemma 17.** *Let deq be a Dequeue operation that returns $v \neq \epsilon$. There exists an Enqueue(v) denoted enq that such that enq is linearized before deq and there is no Dequeue operation $deq' \neq deq$ that also returns v.*

Proof. First, we prove that *enq* exists. Since *deq* returns $v \neq \epsilon$, it has read a value (j, p) in $deqOps[i]$ where i is the sequence number of *deq* (line 24 of Algorithm 1). Meaning that $items[p][j] = v$ and the *Enqueue* operation that enqueued v denoted *enq*, is the j-th instance of *Enqueue* by process p. By linearization rule 2, *deq* is linearized after *enq*. And we have shown in Lemma 7 that each *Enqueue* operation has at most a single matching *Dequeue* operation. The claim follows. ◀

▶ **Lemma 18.** *Let enq_1 and enq_2 be two Enqueue operations such that $enq_1 <_L enq_2$. If enq_2 has a matching Dequeue deq_2, then enq_1 has a matching Dequeue deq_1 and $deq_1 <_L deq_2$.*

Proof. By contradiction, we suppose that enq_1 does not have a matching *Dequeue* operation. From linearization rule 3, enq_1 is linearized after all *Enqueue* operations in ENQ_d. Especially, enq_1 is linearized after enq_2. There is a contradiction. And from linearization rule 1, enq_1 and enq_2 are linearized according to the total order of the sequence numbers of their matching *Dequeue* operations. The claim follows. ◀

From the two previous Lemmas 17–18, we have the following theorem.

▶ **Theorem 19.** *Let deq be a Dequeue operation in L. If deq does not return ϵ, then it returns the element enqueued by the first Enqueue operation in L that does not have a matching Dequeue operation linearized before deq.*

▶ **Lemma 20.** *Let deq_ϵ be a Dequeue operation that returns ϵ. And let enq be an Enqueue operation linearized before deq_ϵ. We have that enq has a matching Dequeue operation deq that is also linearized before deq_ϵ.*

Proof. First, we show that *enq* has a matching *Dequeue* operation *deq*. By contradiction, we suppose that *enq* is in $ENQ \setminus ENQ_d$. From linearization rule 3, *enq* is inserted before the first *Dequeue* operation deq' that starts after *enq* ends or at the end of L if deq' does not exist. The case where *enq* is linearized at the end of L is trivial because it contradicts the fact that *enq* is linearized before deq_ϵ. So deq' exists. By lemma 12 deq' does not return ϵ. Since $enq <_L deq_\epsilon$, we have $deq' <_L deq_\epsilon$ Hence, deq_ϵ has a greater sequence number than deq' from linearization rule 2. Thus, deq_ϵ is complete after deq' is complete (Lemma 3). We conclude by lemma 12, that deq_ϵ does not return ϵ. There is a contradiction. Thus, *enq* has a matching *Dequeue* operation denoted *deq*.

In the following, we establish that *deq* is linearized before deq_ϵ. Let i denote the sequence number of deq_ϵ and let j be the sequence number of *deq*. By contradiction, we assume that $i < j$ (i.e. *deq* is linearized after deq_ϵ). Let deq_k be the first *Dequeue* operation linearized after *enq* with k its sequence number. Such an operation exists as $enq <_L deq_\epsilon$. We have $k \leq i$, according to the linearization rule 2. Assume that deq_k returns ϵ. If $k = 0$ then no operation is linearized before deq_k; in this case, there is a contradiction. Otherwise ($k \geq 1$), there is no *Enqueue* operation linearized after deq_{k-1} and before deq_k because deq_k is linearized immediately after deq_{k-1} (linearization rule 2). This contradicts the fact that deq_k is the first *Dequeue* operation linearized after *enq*. Hence deq_k does not return ϵ. We conclude that $k < i$. Therefore, deq_k is complete before deq_ϵ is complete (Lemma 3). deq_k does not match *enq* as we assume that *deq* is linearized after deq_ϵ. From linearization rule 2, deq_k can only be linearized after *enq* because *enq* terminates before the invocation of deq_k. Thus, by Lemma 12, deq_ϵ cannot return ϵ if $j > i$. There is a contradiction. ◀

3.3.3 Step Complexity

We show that the worst-case step complexity of an *Enqueue* and *Dequeue* operation is $O(\log n)$ and $O(k \log n)$, respectively. To do so, we establish the following Lemma but omit the detailed proof because of space limitations. The main intuition is that while propagating the timestamp, the process has to read a constant number of nodes per level going from a leaf to the root. Since there are n leaves, the height of the tree is in $O(\log n)$.

▶ **Lemma 21.** *A process executes $O(\log n)$ steps during a call to the function Propagate(id).*

During the execution of an *Enqueue* operation there are no loops or function calls aside from a call to the function *Propagate*(id). And during a *Dequeue* operation, a process executes at most k instances of *Propagate*(id). The following corollary ensues.

▶ **Corollary 22.** *A process executes $O(\log n)$ steps during the execution of an Enqueue operation and $O(k \log n)$ steps during the execution of a Dequeue operation.*

4 Set Linearizable Wait-free Queue Algorithm with multiplicity

In this section, we propose an implementation of the relaxed queue with `multiplicity` where the operations have a step complexity of $O(\log n)$. For the relaxed queue with `multiplicity`, concurrent *Dequeue* operations are allowed to return the same element from the queue (Figure 2 illustrates such an execution).

Algorithm 3 Relaxed-Queue: implementation of the wait-free queue with multiplicity (Dequeue pseudo-code for process p).

```
Function Dequeue()
    num ← deqCounter.MaxRead()
    if deqOps[num].Read() ≠ (⊥, ⊥) then
        deqCounter.MaxWrite(num + 1)
        num ← num + 1
    if num ≥ 1 then
        UpdateTree(num − 1)
    FinishDeq(num)
    (h, id) ← deqOps[num].Read()
    if id = ⊥ then
        return ϵ
    else
        (ret, −) ← items[id][h]
        return ret
```

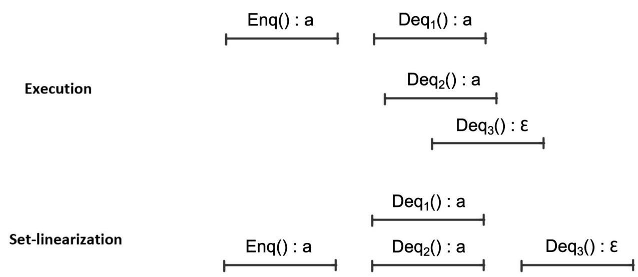

Figure 2 Example of a set-linearizable execution of the relaxed queue with multiplicity.

Only the algorithm of the *Dequeue* operation is different from the Algorithm in Section 3. In the implementation of the relaxed queue, we do not require the unicity of the sequence numbers of the *Dequeue* operations. Therefore, we use a max register object for *deqCounter* instead of the previously used $Fetch\&Inc$. Multiple concurrent *Dequeue* operations retrieve the same sequence number num from $deqCounter$ as long as $deqOps[num]$ remains unchanged. A *Dequeue* operation takes the sequence number $num + 1$ only after the *Dequeue* operations with the sequence number num are completed (i.e. $deqOps[num] \neq (\perp, \perp)$). Thus, we relinquish the need for a helping mechanism for slow *Dequeue* operations since an operation with the same sequence number will need to finish and write to *deqOps* before the next sequence number is assigned.

If the value of *deqCounter* changes between the step a *Dequeue* operation retrieves the value num and the step it reads $deqops[num]$, the operation writes $num + 1$ to $deqCounter$ and assigns it as its sequence number. Similarly to Algorithm 1, the operation then executes the necessary steps to write $deqOps[seq]$ where $seq \in \{num, num+1\}$ is the sequence number of the operation. Meaning that the process executes $UpdateTree(seq - 1)$ if the *Dequeue* operation with the sequence number $seq - 1$ exists, to ensure that the root of the tree has an accurate value. Then, the process executes $FinishDeq(seq)$, after which $deqOps[seq]$ is set to a value different than its initial value. If $DeqOps[seq] = (i, p)$ the *Dequeue* operation returns $items[p][i].val$, otherwise it returns ϵ. Several *Dequeue* operations may have the same sequence number, and thus return the same value. The design of the algorithm ensures that two *Dequeue* operations can have the same sequence number only if they are concurrent. The full proof of correctness of the relaxed queue implementation is omitted because of space limitations but uses similar techniques as the previous sections.

5 Discussion

We have presented a wait-free implementation of a k-multiple dequeuer n-multiple enqueuer FIFO queue. The worst case step complexity of the *Enqueue* operation is in $O(\log n)$ and the *Dequeue* operation is in $O(k \log n)$. Meaning, that as long as the number k of dequeuer processes is constant, our implementation has logarithmic step complexity, which improves on the previous upper bound of $O(\sqrt{n})$. While we focused on theoretical evaluations of step complexity, it could also be of interest to compare the algorithm empirically to other FIFO implementations to gauge its applicative relevance.

Then, to the best of our knowledge, we presented the first relaxed FIFO queue with logarithmic step complexity where every process can perform both *Enqueue*(v) and *Dequeue*$()$ operations. It remains an open question whether it is possible to implement an exact wait-free linearizable FIFO queue with worst-case logarithmic step complexity without restriction on the number of enqueuers and dequeuers or to implement a relaxed FIFO queue in constant or near-constant step complexity.

References

1. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Relaxed queues and stacks from read/write operations. In *24th International Conference on Principles of Distributed Systems*, OPODIS 2020,, pages 13:1–13:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.13.
2. Armando Castañeda and Miguel Piña. Fully read/write fence-free work-stealing with multiplicity, 2020. doi:10.48550/arXiv.2008.04424.
3. Matei David. A single-enqueuer wait-free queue implementation. In *Proceedings of 18th International Conference Distributed Computing*, DISC 2004, pages 132–143, Berlin, Heidelberg, 2004. Springer-Verlag. doi:10.1007/978-3-540-30186-8_10.
4. David Eisenstat. Two-enqueuer queue in common2, 2008.
5. Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theor. Comp. Sys.*, 55(3):475–520, October 2014. doi:10.1007/s00224-013-9491-y.
6. Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. *SIGPLAN Not.*, 48(1):317–328, January 2013. doi:10.1145/2480359.2429109.
7. Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.
8. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.
9. Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS '05, pages 408–419, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11590156_33.
10. Pankaj Khanchandani and Roger Wattenhofer. On the importance of synchronization primitives with low consensus numbers. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN '18, pages 18:1–18:10, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3154273.3154306.

11 Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and scalable, lock-free k-fifo queues. In *Proceedings of 12th International Conferenc Parallel Computing Technologies*, PaCT'13, pages 208–223, Berlin, Heidelberg, 2013. Springer-Verlag. `doi:10.1007/978-3-642-39958-9_18`.

12 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. *SIGPLAN Not.*, 46(8):223–234, February 2011. `doi:10.1145/2038037.1941585`.

13 Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, Univ. of Toronto, January 2001.

14 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2442516.2442527`.

15 Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 396, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/197917.198176`.

16 Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. *SIGPLAN Notices*, 51(8):1–13, February 2016. `doi:10.1145/3016078.2851168`.