

A Wait-free Queue with Poly-logarithmic Step Complexity

Hossein Naderibeni

supervised by Eric Ruppert

November 8, 2022

Abstract

In this work, we introduce a novel linearizable wait-free queue implementation. Linearizability and lock-freedom are standard requirements for designing shared data structures. To the best of our knowledge, all of the existing linearizable lock-free queues in the literature have a common problem in their worst case, the CAS Retry Problem. We show that our algorithm avoids this problem and has a worst-case running time better than prior lock-free queues. The amortized number of steps for an **Enqueue** or **Dequeue** in our algorithm is $O(\log^2 p + \log q)$, where p is the number of processes and q is the size of the queue when the operation is linearized.

Contents

1	Introduction	3
2	Related Work	6
2.1	List-based Queues	6
2.2	Restricted Queues	9
2.3	Universal Constructions and other Poly-log Time Data Structures	9
2.4	Attiya–Fouren Lower Bound	10
3	Queue Implementation	11
3.1	Details of the Implementation	17
3.2	Pseudocode	20
3.3	Example Execution	25

4	Proof of Correctness	31
4.1	Basic Properties	31
4.2	Ordering Operations	35
4.3	Propagating Operations to the Root	38
4.4	Correctness of GetEnqueue	41
4.5	Correctness of IndexDequeue	44
4.6	Linearizability	47
5	Analysis	50
6	Future Directions	52
6.1	Attiya–Fouren Lower Bound	54

1 Introduction

Shared data structures have become an essential field in distributed algorithms research. We are reaching the physical limits of how many transistors we can place on a CPU core. The industry solution to provide more computational power is to increase the number of cores of the CPU. It is not hard to see why multiple processes cannot share a sequential data structure to work with. For example, consider two processes trying to append to a sequential linked list simultaneously. Processes p, q read the same tail node, p changes the next pointer of the tail node to its new node, and then q does the same. In this run, p 's update is overwritten. One solution is to use locks; whenever a process wants to update or query a data structure, the process locks the data structure, and others have to wait until the lock is released to read or update the data structure. Using locks has some disadvantages; for example, one process might be slow, and holding a lock for a long time prevents other processes from progressing. Moreover, locks do not allow complete parallelism since only the process holding the lock can progress when the data structure is locked. For these reasons, we are interested in the design of an efficient shared queue without using locks.

A sequential queue stores a sequence of elements and supports two operations, enqueue and dequeue. **Enqueue(e)** appends the element e to the sequence stored. **Dequeue** removes and returns the first element in the queue. If the queue is empty, it returns **null**. In a concurrent version of the queue data structure, operations do not happen one at a time. The question that may arise is “What properties matter for the implementation of a shared data structure?”, since executions on a shared data structure are different from sequential ones, the correctness conditions also differ. To prove a concurrent object works perfectly, we have to show it satisfies safety and progress conditions. A *safety condition* tells us that the data structure does not return wrong responses, and a *progress property* requires that operations eventually terminate.

A system is called *asynchronous* when processes in the system run at arbitrarily varying speeds, i.e., the scheduling of each process is independent from the scheduling of other processes. Our model is an asynchronous shared-memory distributed system of p processes. Herlihy [12] showed that one cannot implement concurrent versions of all data structures with multi-reader multi-writer registers alone and introduced a hierarchy on how powerful objects are to solve the consensus problem among processes. Objects like LL/SC, CAS can be used to reach a consensus among any number of processors. A CAS object provides an atomic Compare & Swap(**new**, **old**) operation: if the value stored in the object is **old**, it updates the value to **new** and returns **true**, otherwise it returns **false**. In this work, we use Compare & Swap (CAS) objects to synchronize among processes.

The standard safety condition is called *linearizability* [13], which ensures that for any concurrent execution on a linearizable object, each operation should appear to take effect instantaneously at some moment between its invocation and response. Figure 1 depicts an example of an execution on a linearizable queue that is initially empty. The arrow shows time, and each rectangle shows the time between the invocation and the termination of an operation. Since `Enqueue(A)` and `Enqueue(B)` are concurrent, `Enqueue(B)` may or may not take effect before `Enqueue(A)`. The execution in Figure 2 is not linearizable since A has been enqueued before B, so it has to be dequeued first.

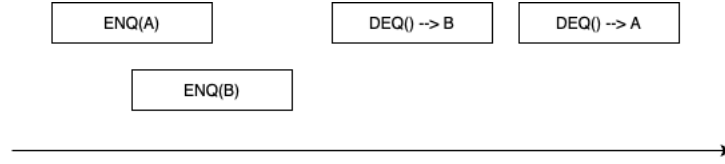


Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

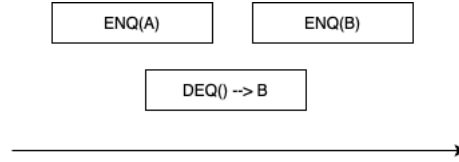


Figure 2: An example of an execution on an empty queue that is not linearizable. `Enqueue(A)` has terminated before `Enqueue(B)` is invoked and the `Dequeue` can occur before termination of `Enqueue(A)` and return `null` or can occur after termination of `Enqueue(A)` and return `A`.

There are various progress properties; the strongest is wait-freedom, and the more common is lock-freedom. An algorithm is *wait-free* if each operation terminates after a finite number of its own steps. We call an algorithm *lock-free* if, after a sufficient number of steps, one operation terminates. A wait-free algorithm is also lock-free but not vice versa; in an infinite run of a lock-free algorithm, there might be an operation that takes infinitely many steps but never terminates.

We will present a wait-free linearizable queue, which to the best of our knowledge is the first wait-free queue whose operations run in a poly-logarithmic number of steps. We design a tournament tree in which each process tries to propagate its operations along a path from the process's leaf up to the root to be linearized. Processes can do queries to get information about the linearization stored in the root. We create a helping mechanism for when a process is propagating its operation to a node, such that new operations are

ensured to be propagated after at most 2 **CAS** invocations. The amortized number of steps for an **Enqueue** or **Dequeue**, is $O(\log^2 p + \log q)$, where q is the size of the queue when the operation is linearized. **CAS** instructions cost more than other instructions for the processor. Each operation in our queue does $\Theta(\log p)$ **CAS** operations compared to $\Omega(p)$ **CAS** steps for previous queues in the worst case. We use unbounded memory space in our queue and present a way to reduce the memory size to the total number of operations. Furthermore, we do not address garbage collection but we describe a way we think it can be handled.

Thesis outline The rest of the thesis is organized as follows. Chapter 2 gives an outline of related work done in the area and the motivation for our implementation. Previous lock-free queues and their common problem is presented in Section 2.1. In Section 2.2, we mention some restricted lock-free queues. Section 2.3 talks about poly-logarithmic constructions of shared objects. Section 2.4 ends with a lower bound on the amortized time complexity of shared queues.

In Chapter 3 we introduce a poly-logarithmic step wait-free queue. In Section 3.1 we give a high-level description of our implementation. We also discuss the motivation and requirements in our design to achieve poly-log time. Section 3.2 contains the algorithm itself.

We prove the correctness of our queue in Chapter 4 by showing it is linearizable.

Chapter 5 is devoted to performance analysis for our implementation. We analyze the number of **CAS** instructions our algorithm invokes and the worst-case and amortized running time of the queue. In the end, we prove our queue is wait-free.

Finally, we give some concluding remarks in Chapter 6. It contains how we can improve our algorithm's memory usage and how to use its idea in designing other wait-free data structures.

2 Related Work

In this section, we look at previous lock-free queues. In the worst-case, step complexity of all the lock-free queues includes an $\Omega(p)$ term that comes from the case where all p processes try to do an enqueue or dequeue simultaneously. Morrison and Afek call this the *CAS retry problem* [23].

2.1 List-based Queues

Michael and Scott [20] introduced a lock-free queue which we refer to as the MS-queue. A version of it is included in the standard Java Concurrency Package. Their idea is to store the queue elements in a singly-linked list (see Figure 3). A shared variable *Head* points to the first node in the linked list that has not been dequeued, and *Tail* points to the last element in the queue. To insert a node into the linked list, they use atomic primitive operations like LL/SC or CAS. If p processes try to enqueue simultaneously, only one can succeed, and the others have to retry. This makes the amortized number of steps $\Omega(p)$ per enqueue. Similarly, dequeue can take $\Omega(p)$ steps.

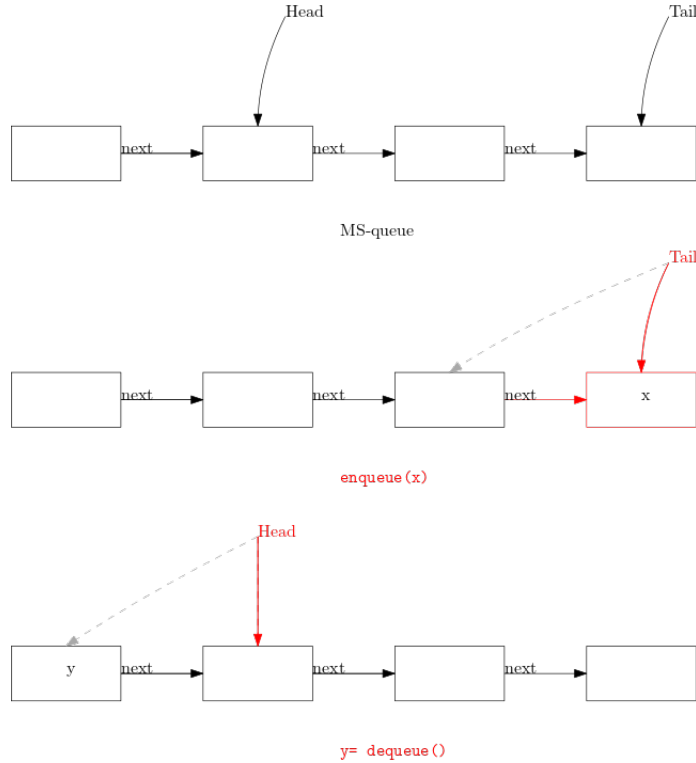


Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram, the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

Moir, Nussbaum, and Shalev [22] presented a more sophisticated queue by using the *elimination* technique. The elimination mechanism has the dual purpose of allowing operations to complete in parallel and reducing contention for the queue. An Elimination Queue consists of an MS-queue augmented with an elimination array. Elimination works by allowing opposing pairs of concurrent operations such as an enqueue and a dequeue to exchange values when the queue is empty or when concurrent operations can be linearized to empty the queue. Their algorithm makes it possible for long-running operations to eliminate an opposing operation. The empirical evaluation showed the throughput of their work is better than the MS-queue, but the worst case is still the same; in case there are p concurrent enqueues, their algorithm is not better than MS-queue.

Hoffman, Shalev, and Shavit [14] tried to make the MS-queue more parallel by introducing the Baskets Queue. Their idea is to allow more parallelism by treating the simultaneous enqueue operations as a basket. Each basket has a time interval in which all its nodes' enqueue operations overlap. Since the operations in a basket are concurrent, we can order them in any way. Enqueues in a basket try to find their order in the basket one by one using **CAS** operations. However, like the previous algorithms, if there are p concurrent enqueue operations in a basket, the amortized step complexity remains $\Omega(p)$ per operation.

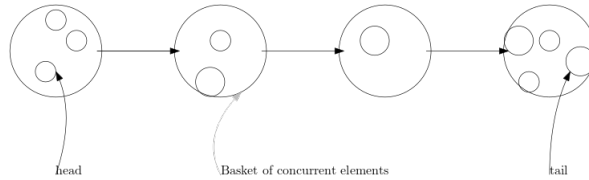


Figure 4: Baskets queue idea. At a time, all operations in a basket ran concurrently, but only one succeeded to do **CAS**. To order the operations in a basket, the mechanism in the process algorithm is to **CAS** again. The successful process will be the next one in the basket, and so on.

Ladan-Mozes and Shavit [19] presented an optimistic approach to implement a queue. MS-queue uses two **CAS**es to do an enqueue: one to change the tail to the new node and another one to change the next pointer of the previous added node to the new node. They use a doubly-linked list to do fewer **CAS** operations in an **Enqueue** than MS-queue. As in previous algorithms, the worst case happens in the case where the contention is high: when p concurrent enqueues happen, their nodes have to be appended to the linked list one by one. The amortized worst-case complexity is still $\Omega(p)$ **CAS**es.

Hendler et al. [11] proposed a new paradigm called flat combining. The key idea behind flat combining is to allow a combiner who has acquired the global lock on the data structure to learn about the requests of

threads on the queue, combine them and apply the combined results on the data structure. Their queue is linearizable but not lock-free and they present experiments that show their algorithm performs well in some situations.

Gidenstam, Sundell, and Tsigas [9] introduced a new algorithm using a linked list of arrays. The queue is stored in a shared array where head and tail pointers point to the current elements in the queue. When the array is full, an empty array is linked to the array and tail pointers are updated. A global head points to the array containing the first element in the queue, and each process has a local head index that points to the first element in that array. Global tail and local tail pointers are similar (see Figure 5). A process updates the position of the pointers after it does an operation. One process might go to sleep before setting the pointers, so the pointers might be behind their real places. They mention how to scan the arrays to update pointers while doing an operation. A process writes an element in the location head by a CAS instruction, so if p processes try to enqueue simultaneously, the step (and CAS) complexity remains $\Omega(p)$. Their design is lock-free but not wait-free.

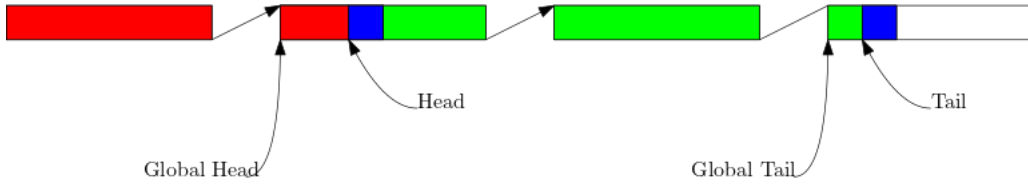


Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

Kogan and Petrank [18] introduced wait-free queues based on the MS-queue and use Herlihy’s helping technique to achieve wait-freedom. Their step complexity is $\Omega(p)$ because of the helping mechanism.

Milman et al. [21] designed a lock-free queue supporting futures. In their queue, operations return future objects instead of responses. Later when the response is needed, it can be evaluated from the future object. They also define a weaker linearizability condition such that each operation can be linearized between its invocation and when its future is evaluated. Their idea of batching allows a sequence of operations to be submitted as a batch for later execution on the MS-queue. They use some properties of the queue size before and after a batch, similar to a part of our work. Their queue is not wait-free, in fact if the batch sizes are 1, then the queue is like MS-queue.

Nikolaev and Ravindran [24] present a wait-free queue that uses the fast-path slow-path methodology. Their work is based on a circular queue using bounded memory. When a process wishes to do an enqueue or

a dequeue, it starts two paths. One is faster but its running time is not bounded and the other one is slower but has a time-bound. They show that these two paths do not affect each other and the queue remains consistent. If a process makes no progress, other processes help its slow path to finish. The helping phase suffers from the CAS Retry Problem, because processes compete in a CAS loop to decide which succeeds to help. Because of this, the worst-case complexity cannot be better than $\Omega(p)$.

CAS Retry Problem is not limited to list-based queues; array-based queues also share the CAS retry problem as well [5, 26, 27]. In all implementations where there is a CAS object with contention p (p processes invoking CAS instructions concurrently), CAS Retry Problem exists. Our motivation is to overcome this problem and present a wait-free sublinear queue.

2.2 Restricted Queues

David introduced the first sublinear concurrent queue [6]. Even though his algorithm does $O(1)$ steps for each operation, it is a single-enqueuer single-dequeuer queue and uses infinite memory. The author states that to reduce memory usage to be bounded, the time per operation increases linearly.

Jayanti and Petrovic introduced a wait-free poly-logarithmic multi-enqueuer single-dequeuer queue [16]. We use their idea of having a tournament tree among processes to agree on the linearization of operations to design a poly-logarithmic multi-enqueuer multi-dequeuer queue. Unlike their work, our algorithm does not put a limit on the number of concurrent dequeuers.

2.3 Universal Constructions and other Poly-log Time Data Structures

A *universal construction* is an algorithm that can implement a shared version of any given sequential object, introduced by Herlihy [12]. We can implement a concurrent queue using a universal construction. Jayanti proved an $\Omega(\log p)$ lower bound on the worst-case shared-access time complexity of p -process universal constructions [15]. He also mentions that the universal construction by Afek, Dauber, and Touitou [1] can be modified to $O(\log p)$ worst-case step complexity, using atomic access to $\Omega(p \log p)$ bit words. Chandra, Jayanti and Tan introduced a semi-universal construction that achieves $O(\log^2 p)$ shared accesses using reasonably sized words [4]. Their universal construction cannot be used to create any data structure, but a set of objects called closed objects (a queue is not a closed object). We mention a non-practical universal construction with a poly-log number of CAS instructions on page 13.

Ellen and Woelfel introduced an implementation of a Fetch&Inc object with step complexity of $O(\log p)$

using $O(\log n)$ -bit LL/SC objects, where n is the number of operations [7]. Their idea to achieve logarithmic complexity is to use a tree storing the Fetch&Inc operations invoked by processes. When a process wants to do Fetch&Inc it adds a Fetch&Inc to the tree and returns the number of elements in the tree. There are some similarities between designing a queue and a Fetch&Inc object. A Fetch&Inc object can be constructed from a queue. The algorithm by Ellen and Woelfel is interesting because it is the first wait-free data structure achieving poly-log implementation after [4].

2.4 Attiya–Fouren Lower Bound

Because of the CAS retry problem in previous list-based queues one might guess the $\Omega(p)$ term is inherent in the time complexity of concurrent queues. Attiya and Fouren gave a lower bound on amortized complexity of lock-free queues with regard to c , the number of concurrent processes. Their result says if c is $O(\log \log p)$, any implementation of queues using reads, writes and conditional operations like CAS has $O(\sqrt{\log \log p})$ amortized complexity [2]. The surprising point is that their result does not contradict ours and we manage to reach poly-log time complexity.

3 Queue Implementation

In our model there are p processes doing **Enqueue** and **Dequeue** operations on a queue concurrently. We design a linearizable wait-free queue with $O(\log^2 p + \log q)$ steps per operation, where q is the number of elements in the queue at the time of linearization. We avoid the $\Omega(p)$ worst-case step complexity of existing shared queues based on linked lists or arrays, which suffer from the CAS Retry Problem.

There is a shared binary tree among the processes (see Figure 6) to agree on one total ordering of the operations invoked by processes. Each process has a leaf in which the operations invoked by the process are stored in order. When a process wishes to do an operation it appends the operation to its leaf and tries to propagate its new operation up to the tree's root. Each node of the tree keeps an ordering of operations propagated up to it. All processes agree on the sequence of operations in the root and this ordering is used as the linearization ordering.

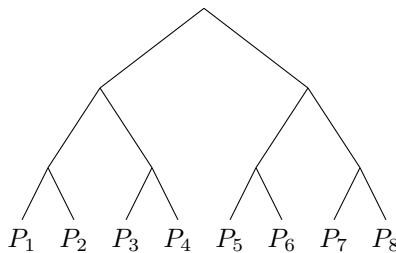


Figure 6: Each of the processes P_1, P_2, \dots, P_p has a leaf and in each node there is an ordering of operations stored. Each process tries to propagate its operations up to the root, which stores a total ordering of all operations.

To propagate operations to a node n in the tree, a process observes the operations in both of n 's children that are not already in n , merges them to create an ordering and then tries to append the ordering to the sequence stored in n . We call this procedure n .**Refresh** (see Figure 7). A **Refresh** on n with a successful append helps to propagate their operations up to the n . We shall prove that if a process invokes **Refresh** on the node n two times and fails to append the new operations to n both times, the operations that were in n 's children before the first **Refresh** are guaranteed to be in n after the second failed **Refresh**. We sketch the argument here.

We use **CAS** (Compare & Swap) instructions to implement the **Refresh**'s attempt to append described in the previous paragraph. The second failed **Refresh** of P is assuredly concurrent with a successful **Refresh**



Figure 7: Before and after a n .**Refresh** with a successful append. Operations propagating from the left child are labelled with l and from the right child with r .

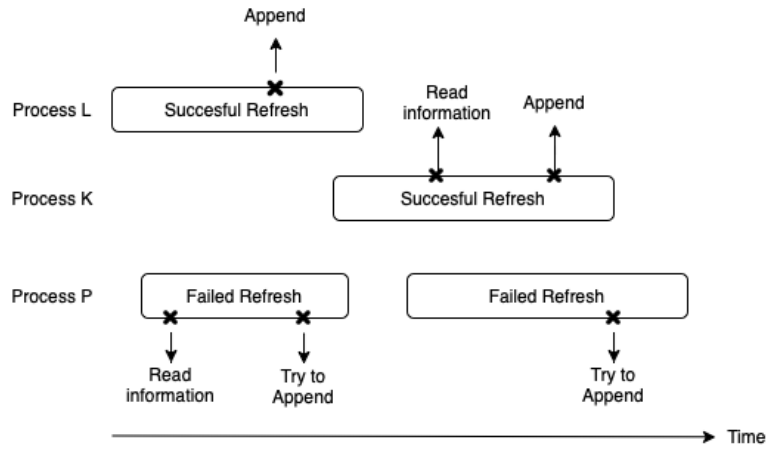


Figure 8: Time relations between the concurrent successful **Refreshes** and the two consecutive **Refreshes**.

that has read its information after the invocation of the first failed **Refresh** (see Figure 8). This is because some process L does a successful append during P 's first failed attempt, and some process K performs a **Refresh** that reads its information after L 's append and then performs a successful append during P 's second failed **Refresh**. Process K 's **Refresh** helps to append the new operations in n 's children before P 's first failed **Refresh**, in case they were not already appended. After a process appends its operation into its leaf it can call **Refresh** on the path up to the root at most two times on each node. So, with $O(\log p)$ CASes an operation can ensure it appears in the linearization. This cooperative solution allows us to overcome the CAS Retry Problem.

It is not efficient to explicitly store the sequence of operations in each node because each operation would have to be copied all the way up to the root; doing this would not be possible in poly-logarithmic time. Instead we use an implicit representation of the operations propagated together. Furthermore, we do not need to maintain an ordering on operations propagated together in a node until they have reached the root. It is sufficient to only keep track of sets of operations propagated together in each **Refresh** and then define the linearization ordering only in the root (see Figure 9). Achieving a constant-sized implicit representation of operations in a **Refresh** allows us to do CAS instruction on fixed-size objects in each **Refresh**. To do that, we introduce *blocks*. A block stores information about the operations propagated by a **Refresh**. It contains the number of operations from the left and the right child propagated to the node by the **Refresh** procedure. See Figure 10 for an example. A node stores an array of blocks of operations propagated up to it. A propagate step aggregates the new blocks in children into a new block and puts it in the parent's blocks. We call the aggregated blocks *subblocks* of the new block and the new block the *superblock* of them. In each **Refresh** there is at most one operation from each process trying to be propagated, because one process cannot invoke two operations concurrently. Thus, there are at most p operations in a block. Furthermore, since the operations in a **Refresh** step are concurrent we can linearize them among themselves in any order we wish, because if two operations are read in one successful **Refresh** step in a node they are going to be propagated up to the root together. Our choice is to put the operations propagated from the left child before the operations propagated from the right child. In this way, if we know the number of operations from the left child and the number of operations from the right child in a block, we have a complete ordering of the operations.

So far, we have a shared tree that processes use to agree on the implicit ordering stored in its root. With this agreement on linearization ordering, we can design a universal construction; for a given object, we can

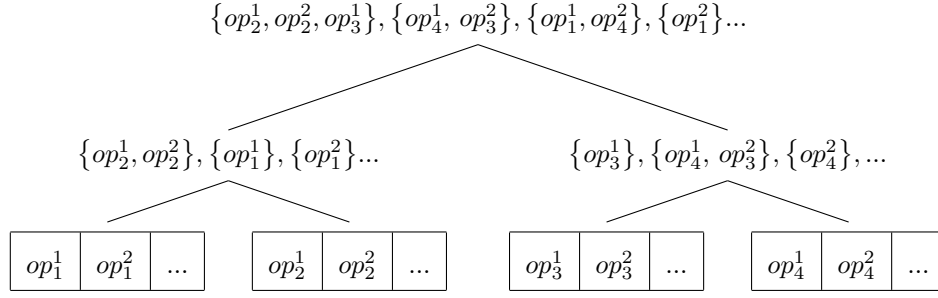


Figure 9: Leaves are for processes P_1 to P_4 from left to right. In each internal node one can arbitrarily linearize the sets of concurrent operations propagated together in a **Refresh**. For example op_4^1 and op_3^2 have propagated together in one **Propagate** step and they will be propagated up to the root together. Since their execution time intervals overlap, they can be linearized in any order.

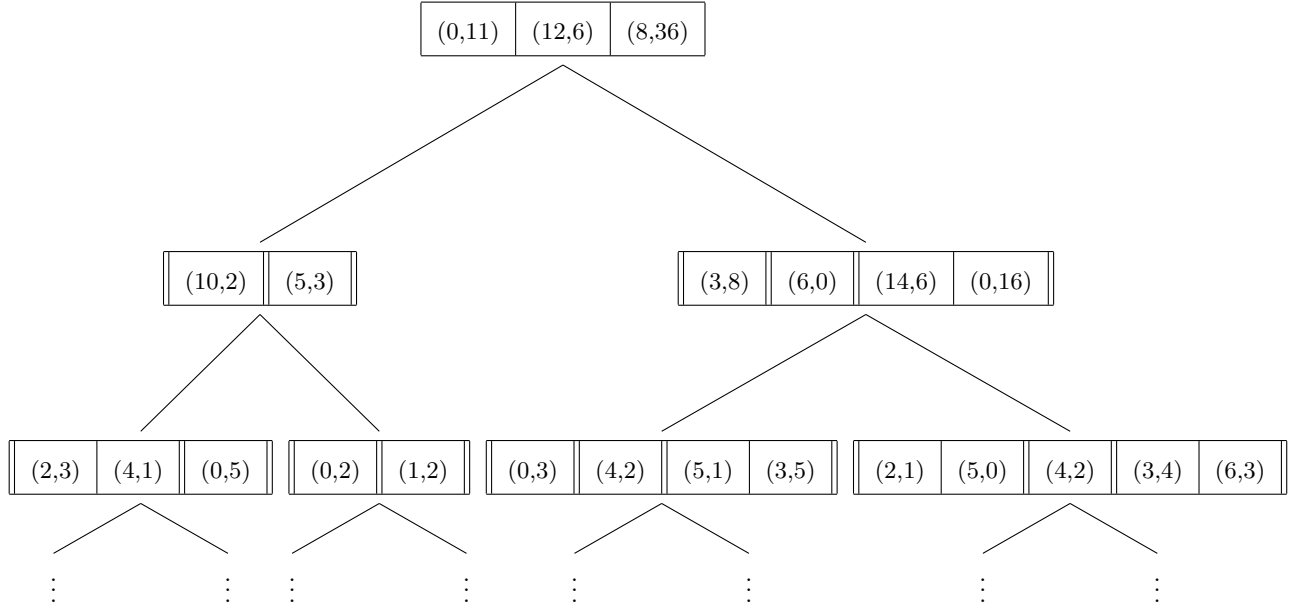


Figure 10: Using blocks to represent operations. Blocks between two lines \parallel are propagated together to the parent. Each block consists of a pair (left, right) indicating the number of operations from the left and the right child, respectively. For example, (12,6) in the root contains (10,2) from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

perform an operation op by applying all the operations up until op in the root on a local copy of the object and then return the response for op . However, this approach is not enough for an efficient queue. We show that we can build an efficient queue if we can compute two things about the ordering in the root: (1) the i th propagated operation and (2) the rank of a propagated operation in the linearization. We explain how to implement (1) and (2) in poly-logarithmic steps.

After propagating an operation op to the root, processes can find out information about the linearization ordering using (1) and (2). To get the i th operation in the root, we find the block B containing the i th operation in the root, and then recursively find the subblock of B in the descendent of the root that contains that i th operation. When we reach a block in a leaf, the operation is explicitly stored there. To make this search faster, instead of iterating over all blocks in the node, we store the prefix sum of the number of elements in the blocks sequence to permit a binary search for the required block. We also store pointers to determine the range of subblocks of a block to make the binary search faster. In each block, we store the prefix sum of operations from the left child and from the right child. Moreover, for each block, we store two pointers to its last left and right subblock (see Figure 11). We know a block size is at most p , so binary search takes at most $O(\log p)$ time, since the pointers of a block and its previous block reduce the search range size to $O(p)$.

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

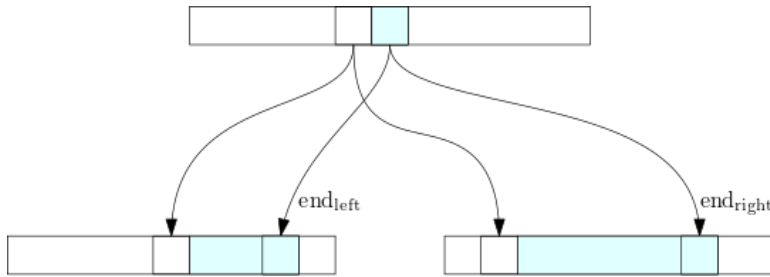


Figure 11: Each block stores the index of its last subblock in each child.

In an execution on a queue where no dequeue operation returns `null`, the k th dequeue returns the argument of the k th enqueue. In the general case a dequeue returns `null` if and only if the queue size after the previous operation is 0. We refer to such a dequeue as a *null dequeue*. If the dequeue is the k th non-null dequeue, it returns the argument of the k th enqueue. Having the size of the queue after an operation we can

compute the number of non-null dequeues from the number of enqueues before the operation. So, if we store the size of the queue after each block of operations in the root, we can compute the index of the enqueue whose argument is the response to a given dequeue in constant time.

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

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

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

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

Table 1: Augmented history of operation blocks on the queue.

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

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

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

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

Given a **Dequeue** is in block B , its response is the argument of the **Enqueue** whose rank is the number of non-null **Dequeues** in blocks $1, 2, \dots, b-1$ + index of the **Dequeue** in B 's **Dequeues**, if $(\text{size of the queue after } b-1\text{th block} + \#\text{Enqueues in } b\text{th block} - \#\text{index of Dequeue in } B\text{'s Dequeues}) \geq 0$. Otherwise, the response would be **null**.

3.1 Details of the Implementation

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

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

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

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

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

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

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

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

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

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

GetEnqueue(b, i) and DSearch(e, end) We can describe an operation in a node in two ways: the rank of the operation among all the operations in the node or the index of the block containing the operation in the node and the rank of the operation within that block. If we know the block and rank within the block of an operation, we can find the subblock containing the operation and the operation's rank within that subblock

in poly-log time. To find the response of a **Dequeue**, we know about the rank of the response **Enqueue** in the root (**e** in Line 22). We also know the **eth Enqueue** is in `root.blocks[1..end]`. **DSearch** uses doubling to find the range that contains the answer block (Lines 38–41) and then tries to find the required indices with a binary search (Line 42). A call to $n.\text{GetEnqueue}(b, i)$ returns the **element** of the i th enqueue in b th the block of n . The range of subblocks of a block is determined using the `endleft` and `endright` fields of the block and its previous block. Then, the subblock is found using binary search on the `sumenq` field (Lines 99 and 103).

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

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

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

3.2 Pseudocode

Algorithm Tree Fields Description

◊ *Shared*

- A binary tree of Nodes with one leaf for each process. root is the root node.

◊ *Local*

- *Node leaf*: process's leaf in the tree.

► *Node*

- **Node left, right, parent* : Initialized when creating the tree.
- *Block[] blocks* : Initially blocks[0] contains an empty block with all fields equal to 0.
- *int head= 1*: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.

► *Block*

- *int super* : approximate index of the superblock, read from *parent.head* when appending the block to the node

► *InternalBlock extends Block*

- *int end_{left}, end_{right}* : indices of the last subblock of the block in the left and right child
- *int sum_{enq-left}* : # enqueues in *left.blocks[1..end_{left}]*
- *int sum_{deq-left}* : # dequeues in *left.blocks[1..end_{left}]*
- *int sum_{enq-right}* : # enqueues in *right.blocks[1..end_{right}]*
- *int sum_{deq-right}* : # dequeues in *right.blocks[1..end_{right}]*

► *LeafBlock extends Block*

- *Object element* : Each block in a leaf represents a single operation. If the operation is enqueue(x) then element=x, otherwise element=null.
- *int sum_{enq}, sum_{deq}* : # enqueue, dequeue operations in the prefix for the block

► *RootBlock extends InternalBlock*

- *int size* : size of the queue after performing all operations in the prefix for this block
-

Abbreviations used in the code and the proof of correctness.

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

Algorithm Queue

```
1: void Enqueue(Object e) ▷ Creates a block with element e and adds it to the tree.
2:   block newBlock= new(LeafBlock)
3:   newBlock.element= e
4:   newBlock.sumenq= leaf.blocks[leaf.head].sumenq+1
5:   newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq
6:   leaf.Append(newBlock)
7: end Enqueue

8: Object Dequeue() ▷ Creates a block with null value element, appends it to the tree and returns its response.
9:   block newBlock= new(LeafBlock)
10:  newBlock.element= null
11:  newBlock.sumenq= leaf.blocks[leaf.head].sumenq
12:  newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq+1
13:  leaf.Append(newBlock)
14:  <b, i>= IndexDequeue(leaf.head, 1)
15:  output= FindResponse(b, i)
16:  return output
17: end Dequeue

18: element FindResponse(int b, int i) ▷ Returns the response to  $D_i(root, b)$ , the  $i$ th Dequeue in  $root.blocks[b]$ .
19:   if root.blocks[b-1].size + root.blocks[b].numenq - i < 0 then ▷ Check if the queue is empty.
20:     return null
21:   else
22:     e= i + (root.blocks[b-1].sumenq-root.blocks[b-1].size) ▷ The response is  $E_e(root)$ , the  $e$ th Enqueue in the root.
23:     return root.GetEnqueue(root.DSearch(e, b))
24:   end if
25: end FindResponse
```

Algorithm Node

\leadsto Precondition: `blocks[start..end]` contains a block with `sumenq` greater than or equal to `x`

\triangleright Does a binary search for the value `x` of `sumenq` field and returns the index of the leftmost block in `blocks[start..end]` whose `sumenq` is $\geq x$.

```
26: int BinarySearch( int x, int start, int end)
27:   while start<=end do
28:     int mid= (start+end)/2
29:     if blocks[mid].sumenq<x then
30:       start= mid+1
31:     else
32:       end= mid
33:     end if
34:   end while
35:   return start
36: end BinarySearch
```

Algorithm Root

\leadsto Precondition: `root.blocks[end].sumenq` $\geq e$

\triangleright Returns $\langle b, i \rangle$ such that $E_e(\text{root}) = E_i(\text{root}, b)$, i.e., the e th Enqueue in the `root` is the i th Enqueue within block `b` of the `root`.

```
37: <int, int> DSearch(int e, int end)
38:   start= end-1
39:   while root.blocks[start].sumenq>=e do
40:     start= max(start-(end-start), 0)
41:   end while
42:   b= root.BinarySearch( e, start, end)
43:   i= e- root.blocks[b-1].sumenq
44:   return <b,i>
45: end DSearch
```

Algorithm Leaf

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

Algorithm *Node*

```

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

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

74: void Advance(int h)
75:   hp = parent.head
76:   blocks[h].super.CAS(null, hp)
77:   head.CAS(h, h+1)
78: end Advance

    ▷ Creates and returns the block to be installed in blocks[i].
    Created block includes left.blocks[indexprev+1..indexlast] and right.blocks[indexprev+1..indexlast].
79: Block CreateBlock(int i)
80:   block new = new(InternalBlock)
81:   for each dir in {left, right} do
82:     indexprev = blocks[i-1].enddir
83:     new.enddir = dir.head-1
84:     blockprev = dir.blocks[indexprev]
85:     blocklast = dir.blocks[new.enddir]
86:     new.sumenq-dir = blocks[i-1].sumenq-dir +
                        blocklast.sumenq - blockprev.sumenq
87:     new.sumdeq-dir = blocks[i-1].sumdeq-dir +
                        blocklast.sumdeq - blockprev.sumdeq
88:   end for
89:   if this is root then
90:     new.type = InternalBlock --> RootBlock
91:     new.size = max(root.blocks[i-1].size + new.numenq
                    - new.numdeq, 0)
92:   end if
93:   return new
94: end CreateBlock
```

Algorithm Node

\rightsquigarrow Precondition: $\text{blocks}[b].\text{num}_{\text{enq}} \geq i \geq 1$

```
95: element GetEnqueue(int b, int i)  $\triangleright$  Returns the element of  $E_i(\text{this}, b)$ .
96:   if this is leaf then
97:     return blocks[b].element
98:   else if i  $\leq$  blocks[b].numenq-left then  $\triangleright E_i(\text{this}, b)$  is in the left child of this node.
99:     subBlock = left.BinarySearch( i + blocks[b-1].sumenq-left, blocks[b-1].endleft + 1, blocks[b].endleft)
100:    return left.GetEnqueue(subBlock, i)
101:   else
102:     i = i - blocks[b].numenq-left
103:     subBlock = right.BinarySearch( i + blocks[b-1].sumenq-right, blocks[b-1].endright + 1, blocks[b].endright)
104:     return right.GetEnqueue(subBlock, i)
105:   end if
106: end GetEnqueue
```

\rightsquigarrow Precondition: bth block of the node has propagated up to the root and $\text{blocks}[b].\text{num}_{\text{deq}} \geq i$.

```
107: <int, int> IndexDequeue(int b, int i)  $\triangleright$  Returns <x, y> if  $D_i(\text{this}, b) = D_y(\text{root}, x)$ .
108:   if this is root then
109:     return <b, i>
110:   else
111:     dir = (parent.left == n ? left : right)
112:     sb = (parent.blocks[blocks[b].super].sumdeq-dir > blocks[b].sumdeq ? blocks[b].super : blocks[b].super + 1)
113:     if dir is left then
114:       i += blocks[b-1].sumdeq-parent.blocks[sb-1].sumdeq-left
115:     else
116:       i += blocks[b-1].sumdeq-parent.blocks[sb-1].sumdeq-right
117:       i += parent.blocks[sb].numdeq-left
118:     end if
119:     return this.parent.IndexDequeue(sb, i)
120:   end if
121: end IndexDequeue
```

3.3 Example Execution

Here we want to demonstrate how four processes help each other to propagate their operations up to the root as time goes on and then compute their responses.

In Figure 12, processes $P1, P2, P4$ have appended the blocks containing ENQ_1, DEQ_2, DEQ_4 . Line 47 appends block B to the leaf.

Then as in Figure 13, $P2$ does a successful **Refresh** on the node $n5$ and propagates both $P1$ and $P2$ operations. $P4$ does also a successful **Refresh** on the node $n6$ and propagates DEQ_4 . $P1$'s **Refresh** on $n5$ is slower than $P2$'s and creates its new block after the $P2$'s successful **CAS** that put ENQ_1, ENQ_2 . So the block created by $P1$ is empty as there is nothing to propagate ($P1$'s 60 is after $P2$'s 70).

Since the block $P1$ creates is empty, it returns **true** in Line 68 as shown in Figure 14. $P3$ propagates ENQ_3 to $n6$ and $P4$ is very fast and it reaches DEQ_4 to the **root** even before $P2$ puts its block to $n5$.

Finally, $P4$ goes to sleep, and $P2$ helps to increment **root.head** after its failed **CAS** on the root. After incrementing **root.head**, $P3$ succeeds to do **root.Refresh** with a single **CAS** on the **root** and puts operations from $P1, P2$ and $P3$ to the **root**. $P1$ fails to do **Refresh** on the **root** two times, but for sure, its operation is propagated after the second failed **Refresh**. See Figure 15

All the fields of some blocks from Figure 15 are filled in Figure 16. $P1$ and $P3$'s operations terminated after that they appended the **Enqueues** to the root. $P2$ needs to compute the index of the block that its **Dequeue** is in. It computes the superbloc of its leaf block, which is the second block in $n5$. Then computes the superbloc of ENQ_1, DEQ_2 . As we can see **super** field of ENQ_1, DEQ_2 is 1 but the index of its superbloc is 2. Finally reaches the block ENQ_1, ENQ_3, DEQ_2 . The queue size after the previous block is 0 so DEQ_2 's response is ENQ_1 's element. Then it tries to get the ENQ_1 's element. ENQ_1 is in the left subblock of block ENQ_1, ENQ_3, DEQ_2 ; because $1 < ENQ_1, ENQ_3, DEQ_2 \cdot \text{sum}_{\text{enq-left}}$. It reaches the leaf $n1$ and returns ENQ_1 's element. DEQ_4 returns its response the same.

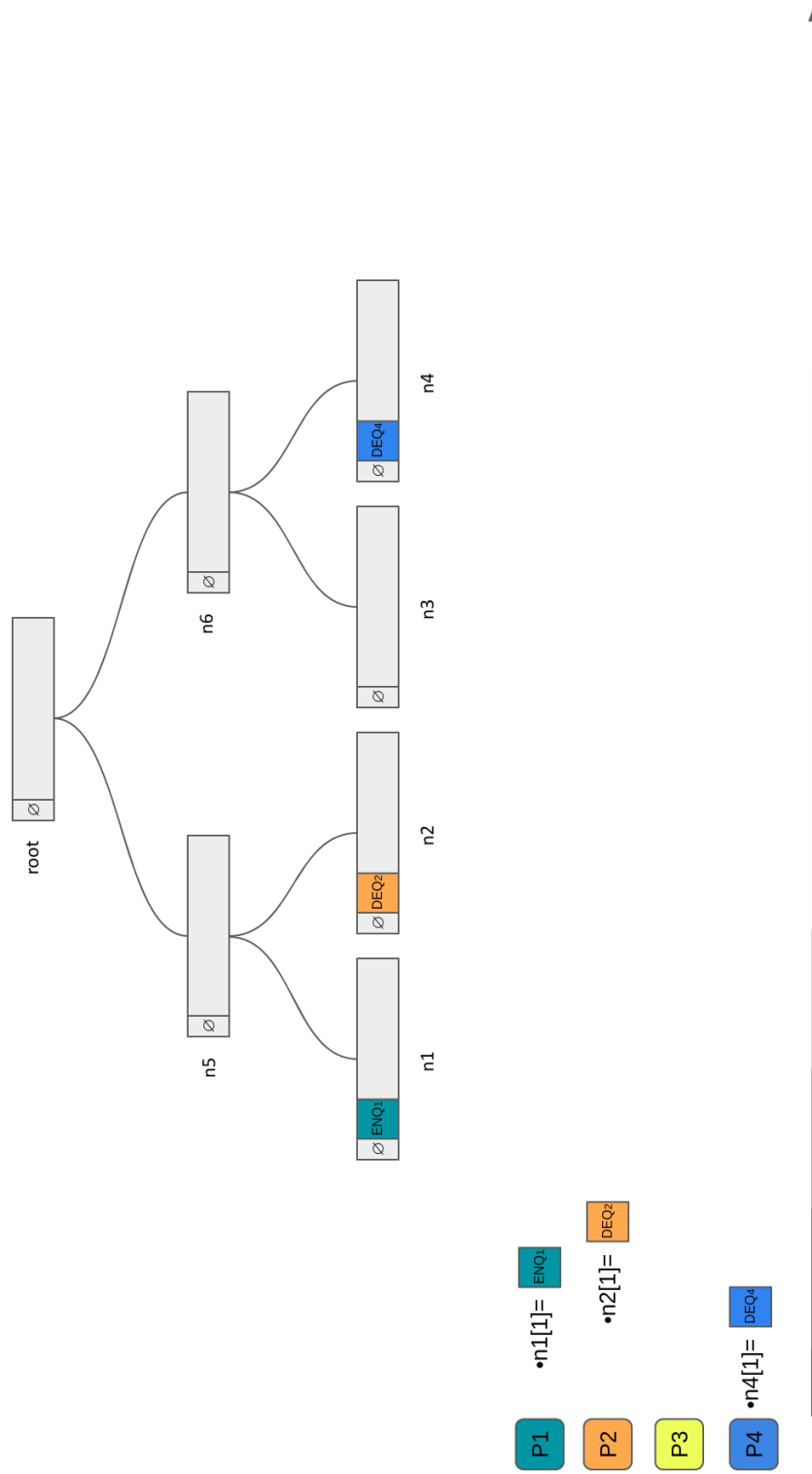


Figure 12: $P1, P2, P4$ append operations.

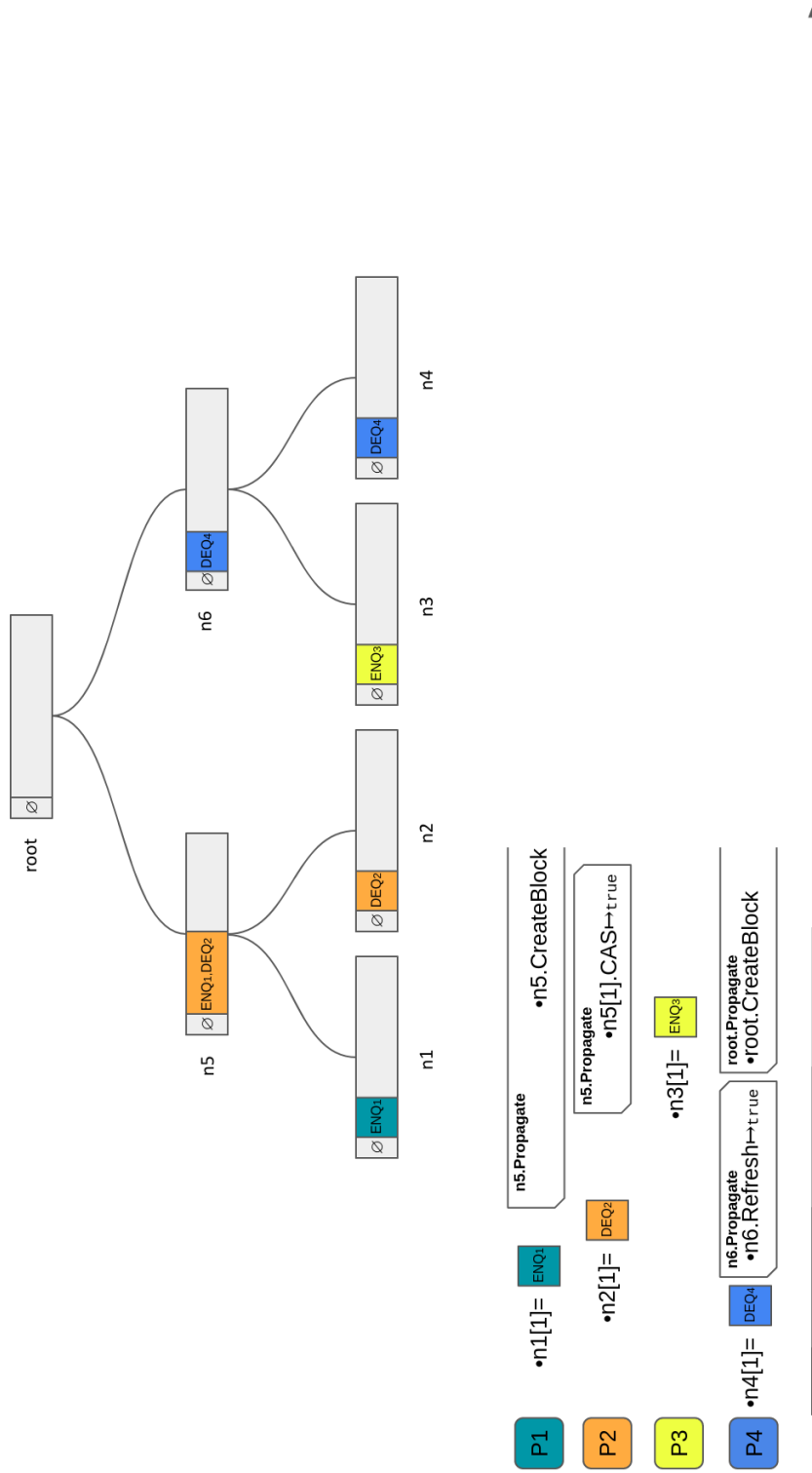


Figure 13: P_2 helps P_1 . P_4 is faster than P_3 .

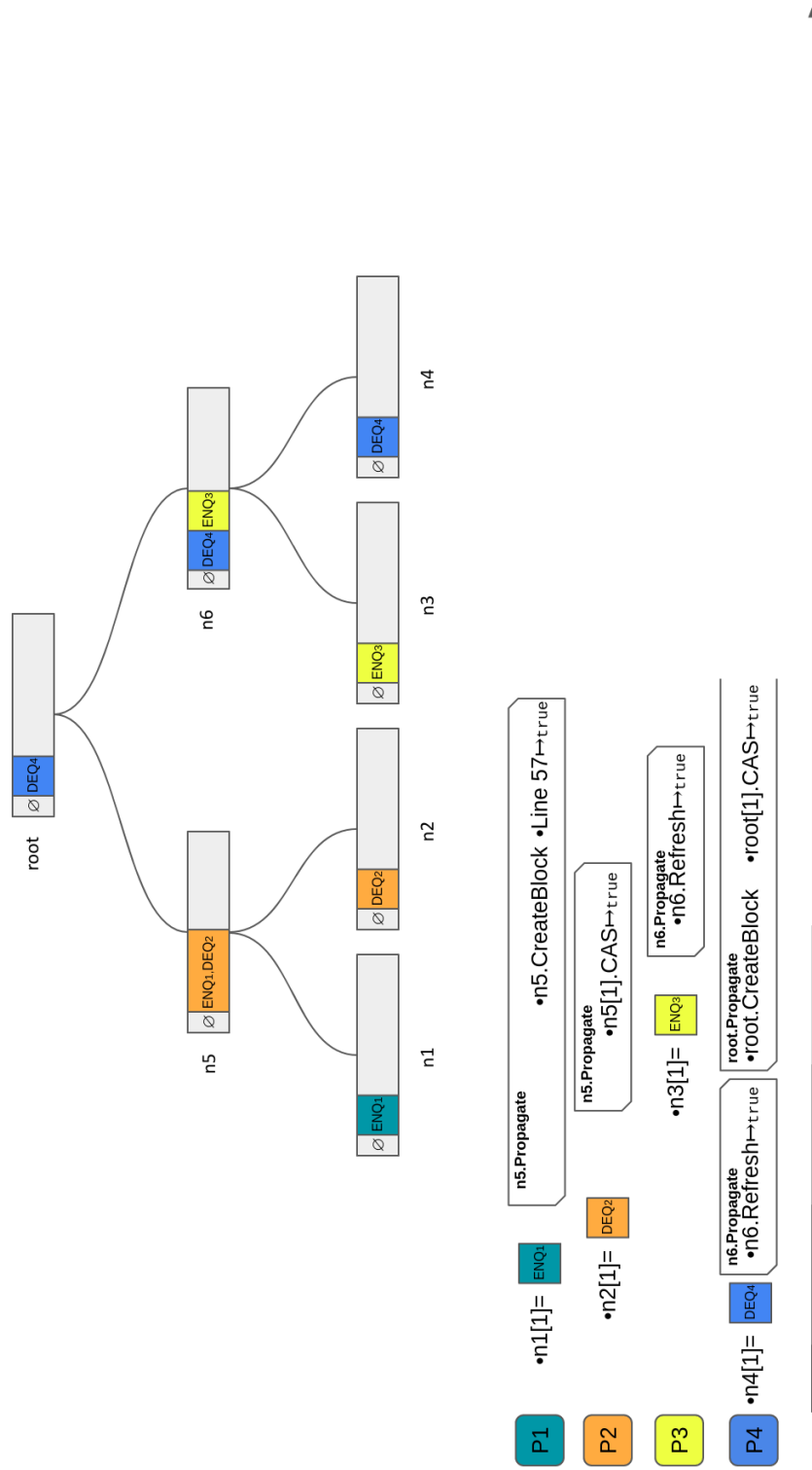


Figure 14: P_4 puts its block to the root before P_2 appending its block to n_5 .

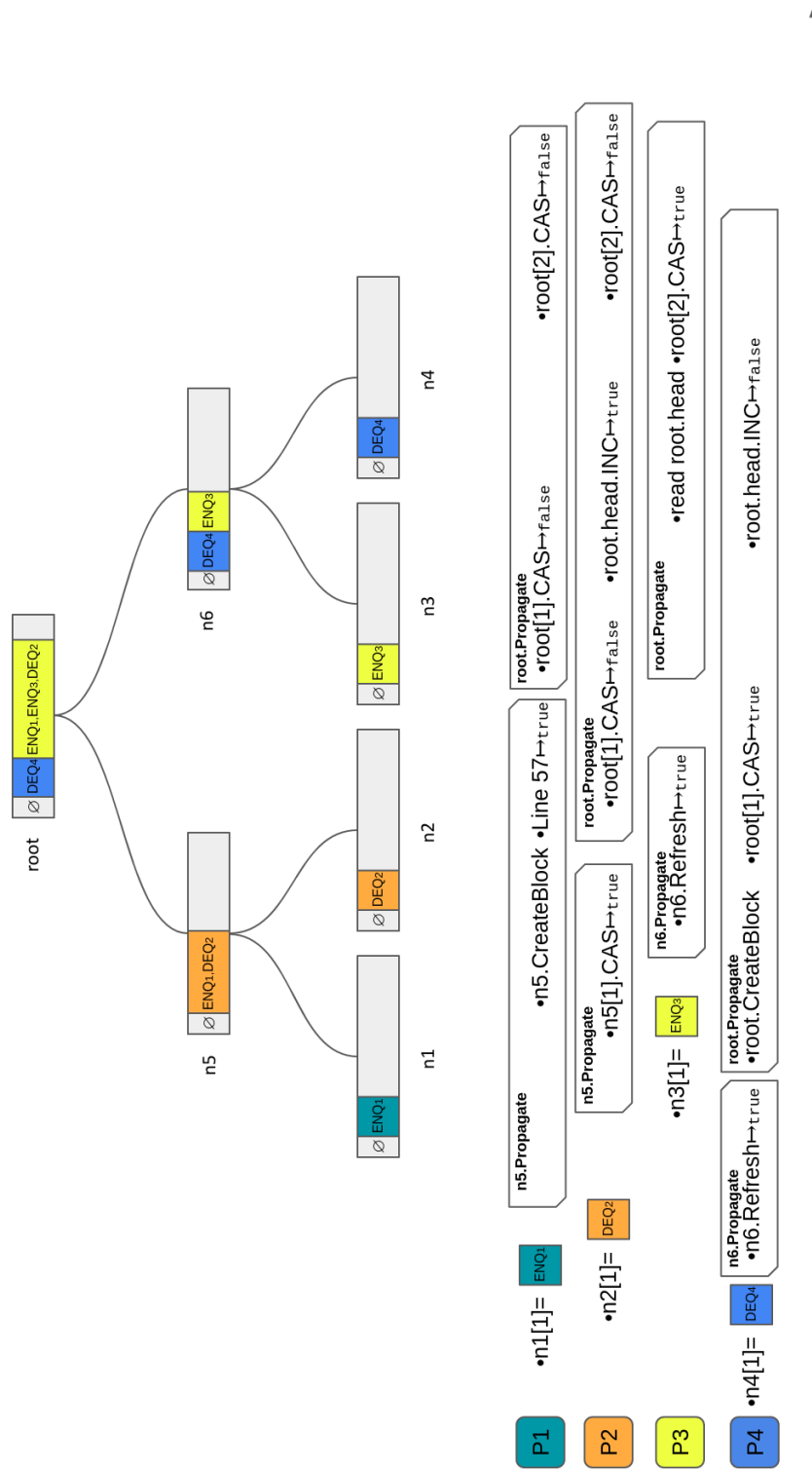


Figure 15: $P3$ succeeds to the CAS to the root. $P1$'s work is done after two failed CASes.

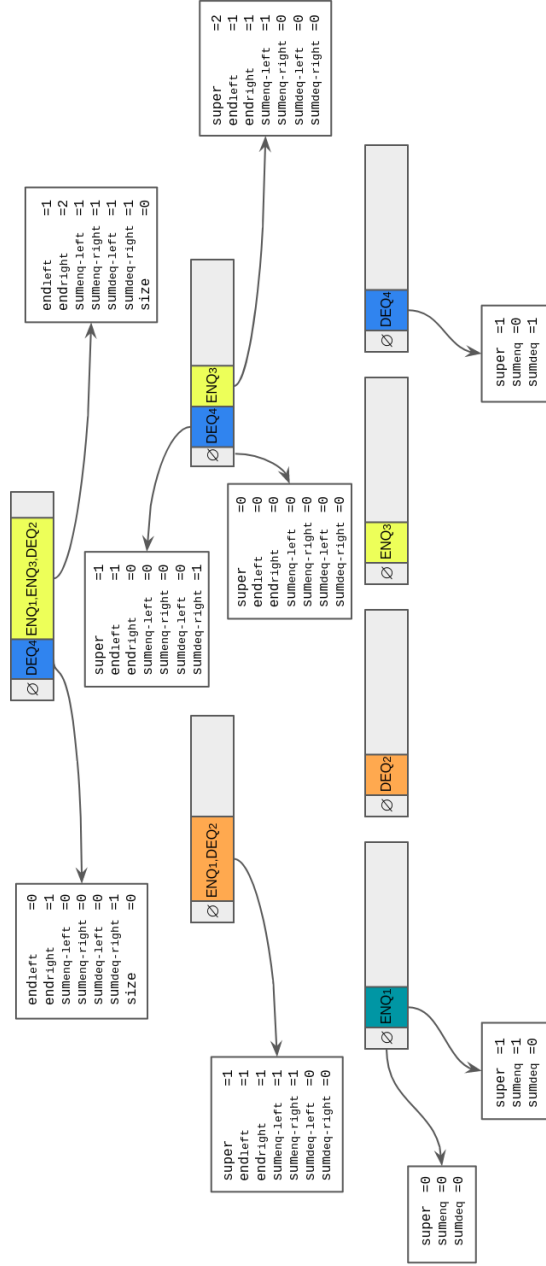


Figure 16: Figure 15 with completed fields.

4 Proof of Correctness

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

4.1 Basic Properties

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

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

Definition 1 (Ordering of a block in a node). Let b be $n.\text{blocks}[i]$ and b' be $n.\text{blocks}[j]$. We call i the *index* of block b . Block b is *before* block b' in node n if and only if $i < j$. We define *the prefix* for block b in node n to be the blocks in $n.\text{blocks}[0..i]$.

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

Observation 2. *For each node n , $n.\text{head}$ is non-decreasing over time.*

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

Lemma 3. *Let R be an instance of **Refresh** on a node n that its created block `new` is not empty (`new.sum` $\neq 0$). After R terminates, $n.\text{head}$ is greater than the value read in line 60 of R .*

Proof. If the **CAS** in Line 77 is successful, then the claim holds. Otherwise, $n.\text{head}$ has changed from the value read in Line 60. By Observation 2 this means another process has incremented $n.\text{head}$. □

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

Invariant 4 (headPosition). If the value of $n.\text{head}$ is h then $n.\text{blocks}[i] = \text{null}$ for $i > h$ and $n.\text{blocks}[i] \neq \text{null}$ for $0 \leq i < h$.

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

In the algorithm, $n.\text{blocks}$ is modified only on Line 70, which updates $n.\text{blocks}[h]$ where h is the value read from $n.\text{head}$ in Line 60. Since the CAS in Line 70 is successful, it means $n.\text{head}$ has not changed from h before doing the CAS: if $n.\text{head}$ had changed before the CAS then it would be greater than h by Observation 2 and hence $n.\text{blocks}[h] \neq \text{null}$ and by the induction hypothesis, so the CAS would fail. Writing into $n.\text{blocks}[h]$ when $h = n.\text{head}$ preserves the invariant, since the claim does not talk about the content of $n.\text{blocks}[n.\text{head}]$.

The value of $n.\text{head}$ is modified only in Line 77 of **Advance**. If $n.\text{head}$ is incremented to $h+1$ it is sufficient to show $n.\text{blocks}[h] \neq \text{null}$. **Advance** is called in Lines 64 and 71. For Line 64, $n.\text{blocks}[h] \neq \text{null}$ because of the **if** condition in Line 63. For Line 71, Line 70 was finished before doing 71. Whether Line 70 is successful or not, $n.\text{blocks}[h] \neq \text{null}$ after the $n.\text{blocks}[h].\text{CAS}$. \square

We define the subblocks of a block recursively.

Definition 5 (Subblock). A block is a *direct subblock* of the i th block in node n if it is in

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

or in

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

Block b is a *subblock* of block c if b is a direct subblock of c or a subblock of a direct subblock of c . We say block b is *propagated* to node n if b is in $n.\text{blocks}$ or is a subblock of a block in $n.\text{blocks}$.

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

Lemma 6. If $n.\text{blocks}[i] \neq \text{null}$ and $i > 0$ then $n.\text{blocks}[i].\text{end}_{\text{left}} \geq n.\text{blocks}[i-1].\text{end}_{\text{left}}$ and $n.\text{blocks}[i].\text{end}_{\text{right}} \geq n.\text{blocks}[i-1].\text{end}_{\text{right}}$.

Proof. Consider the block b written into $n.\text{blocks}[i]$ by CAS at Line 70. Block b is created by the `CreateBlock(i)` called at Line 67. Prior to this call to `CreateBlock(i)`, $n.\text{head} = i$ at Line 60, so $n.\text{blocks}[i - 1]$ is already a non-null value b' by Invariant 4. Thus, the `CreateBlock($i - 1$)` that created b' terminated before the `CreateBlock(i)` that creates b is invoked. The value written into $b.\text{end}_{\text{left}}$ at Line 83 of `CreateBlock(i)` was one less than the value of $n.\text{left.head}$ read at Line 83 of `CreateBlock(i)`. Similarly, the value in $n.\text{blocks}[i - 1].\text{end}_{\text{left}}$ was one less than the value read from $n.\text{left.head}$ during the call to `CreateBlock($i - 1$)`. By Observation 2, $n.\text{left.head}$ is non-decreasing, so $b'.\text{end}_{\text{left}} \leq b.\text{end}_{\text{left}}$. The proof for $\text{end}_{\text{right}}$ is similar. \square

Lemma 7. *Subblocks of any two blocks in node n do not overlap.*

Proof. We are going to prove the lemma by contradiction. Consider the lowest node n in the tree that violates the claim. Then subblocks of $n.\text{blocks}[i]$ and $n.\text{blocks}[j]$ overlap for some $i < j$. Since n is the lowest node in the tree violating the claim, direct subblocks of blocks of $n.\text{blocks}[i]$ and $n.\text{blocks}[j]$ have to overlap. Without loss of generality, assume left child subblocks of $n.\text{blocks}[i]$ overlap with the left child subblocks of $n.\text{blocks}[j]$. By Lemma 6 we have $n.\text{blocks}[i].\text{end}_{\text{left}} \leq n.\text{blocks}[j - 1].\text{end}_{\text{left}}$, so the ranges $[n.\text{blocks}[i - 1].\text{end}_{\text{left}} + 1 \dots n.\text{blocks}[i].\text{end}_{\text{left}}]$ and $[n.\text{blocks}[j - 1].\text{end}_{\text{left}} + 1 \dots n.\text{blocks}[j].\text{end}_{\text{left}}]$ cannot overlap. Therefore, direct subblocks of $n.\text{blocks}[i]$ and $n.\text{blocks}[j]$ cannot overlap. \square

Definition 8 (Superblock). Block b is *superblock* of block c if c is a direct subblock of b .

Corollary 9. *Every block has at most one superblock.*

Proof. A block having more than one superblock contradicts Lemma 7. \square

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

Definition 10 (Operations of a block). A block b in a leaf represents an `Enqueue` if $b.\text{element} \neq \text{null}$. Otherwise, if $b.\text{element} = \text{null}$, b represents a `Dequeue`. The set of operations of block b is the union of the operations in leaf subblocks of b . We denote the set of operations of block b by $\text{ops}(b)$ and the union of operations of a set of blocks B by $\text{ops}(B)$. We also say b contains op if $op \in \text{ops}(b)$.

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

Lemma 11. *If op is in $n.\text{blocks}[i]$ then there is no $j \neq i$ such that op is in $n.\text{blocks}[j]$.*

Proof. We prove this claim by contradiction using Lemma 7. Assume op is in the subblocks of both $n.blocks[i]$ and $n.blocks[j]$. From Lemma 7 we know that the subblocks of these blocks are different, so there are two leaf blocks containing op . Since each process puts each operation in only one block of its leaf, then op cannot be in two leaf blocks. This is a contradiction. \square

Definition 12. $n.blocks[i]$ is *established* if $n.head > i$. An operation is *established* in node n if it is in an established block of n . EST_t^n is the set of established operations in node n at time t .

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

Observation 13. If time $t < \text{time } t'$ (t is before t'), then $ops(n.blocks)$ at time t is a subset of $ops(n.blocks)$ at time t' .

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

Corollary 14. If time $t < \text{time } t'$, then $EST_n^t \subseteq EST_n^{t'}$.

Proof. From Observations 2, 13. \square

4.2 Ordering Operations

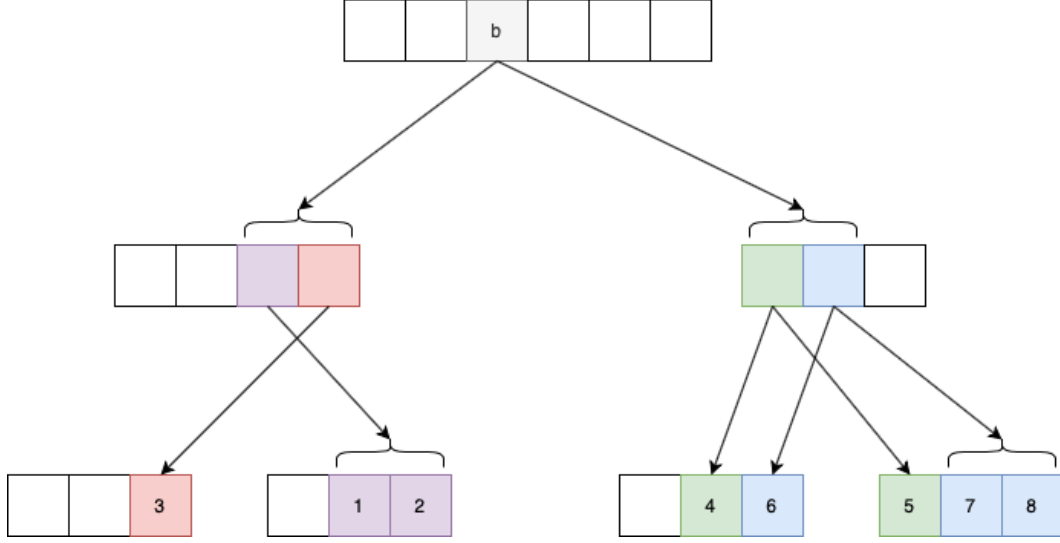


Figure 17: Order of operations in b . Operations in the leaves are ordered in the numerical order shown in the drawing.

Now we define the ordering of operations stored in each node. In the non-root nodes, we only need to order operations of a type among themselves. Processes are numbered from 1 to p , and leaves of the tree are assigned from left to right. We will show in Lemma 28 that there is at most one operation from each process in a given block.

Definition 15 (Ordering of operations inside the nodes).

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

$$E(n, b) = E(n.left, n.blocks[b-1].end_{left} + 1) \cdots E(n.left, n.blocks[b].end_{left}) \cdot \\ E(n.right, n.blocks[b-1].end_{right} + 1) \cdots E(n.right, n.blocks[b].end_{right}).$$

- $E_i(n, b)$ is the i th enqueue in $E(n, b)$.
- The order of the enqueue operations in the node n is $E(n) = E(n, 1) \cdot E(n, 2) \cdot E(n, 3) \cdots$
- $E_i(n)$ is the i th enqueue in $E(n)$.

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

$$D(n, b) = D(n.left, n.blocks[b-1].end_{left} + 1) \cdots D(n.left, n.blocks[b].end_{left}) \cdot \\ D(n.right, n.blocks[b-1].end_{right} + 1) \cdots D(n.right, n.blocks[b].end_{right}).$$

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

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

Definition 16 (Linearization).

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

Observation 17. For any node n and indices $i < j$ of blocks in n , we have

$$n.blocks[j].sum_x - n.blocks[i].sum_x = \sum_{k=i+1}^j n.blocks[k].num_x$$

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

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

Lemma 18. Let B, B' be $n.blocks[b], n.blocks[b-1]$ respectively.

- (1) If n is an internal node then $B.num_{enq-left} = \left| E(n.left, B'.end_{left} + 1) \cdots E(n.left, B.end_{left}) \right|$.
- (2) If n is an internal node then $B.num_{enq-right} = \left| E(n.right, B'.end_{right} + 1) \cdots E(n.right, B.end_{right}) \right|$.
- (3) $B.num_{enq} = \left| E(n, b) \right|$.

Proof. We prove the claim by induction on the height of node n . Base case (3) for leaves is trivial, and (1) and (2) are vacuously true for leaves. Supposing the claim is true for n 's children, we prove the correctness

of the claim for n .

$$\begin{aligned}
B.\text{num}_{\text{enq-left}} &= B.\text{sum}_{\text{enq-left}} - B'.\text{sum}_{\text{enq-left}} && \text{Definition of num}_{\text{enq}} \\
&= B'.\text{sum}_{\text{enq-left}} + n.\text{left.blocks}[B.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} \\
&\quad - n.\text{left.blocks}[B'.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} - B'.\text{sum}_{\text{enq-left}} && \text{CreateBlock} \\
&= n.\text{left.blocks}[B.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} - n.\text{left.blocks}[B'.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} \\
&= \sum_{i=B'.\text{end}_{\text{left}}+1}^{B.\text{end}_{\text{left}}} n.\text{left.blocks}[i].\text{num}_{\text{enq}} && \text{Observation 17} \\
&= \left| E(n.\text{left}, B'.\text{end}_{\text{left}} + 1) \cdots E(n.\text{left}, B.\text{end}_{\text{left}}) \right| && \text{Induction hypothesis (3)}
\end{aligned}$$

The last line holds because of the induction hypothesis (3). (2) is similar to (1). Now we prove (3) starting from the Definition of $E(n, b)$.

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

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

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

Corollary 19. *Let B be $n.\text{blocks}[b]$.*

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

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

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

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

4.3 Propagating Operations to the Root

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

Definition 20. Let t^{op} be the time op is invoked, ^{op}t be the time op terminates, t_l^{op} be the time immediately before running Line l of operation op and $^{op}_l t$ be the time immediately after running Line l of operation op . We sometimes suppress op and write t_l or ${}_l t$ if op is clear from the context. In the text, v_l is the value of variable v immediately after line l for the process we are talking about and v_t is the value of variable v at time t .

Definition 21 (Successful Refresh). An instance of **Refresh** is *successful* if its **CAS** in Line 70 returns **true**. If a successful instance of **Refresh** terminates, we say it is *complete*.

In the next two results, we show that for every successful **Refresh**, all the operations established in the children before the **Refresh** are in the parent after the **Refresh**'s successful **CAS** at Line 70.

Lemma 22. *If R is a successful instance of n .**Refresh**, then we have $EST_{n.\text{left}}^{t^R} \cup EST_{n.\text{right}}^{t^R} \subseteq ops(n.\text{blocks}_{70})$.*

Proof. We show

$$\begin{aligned} EST_{n.\text{left}}^{t^R} &= ops(n.\text{left.blocks}[0..n.\text{left.head}_{t^R} - 1]) \\ &\subseteq ops(n.\text{blocks}_{70}) = ops(n.\text{blocks}[0..n.\text{head}_{70}]). \end{aligned}$$

In every node, $\text{blocks}[0]$ is an empty block without any operations. Line 70 stores a block **new** in n that has $\text{end}_{\text{left}} = n.\text{left.head}_{83} - 1$. Therefore, by Definition 5, after the successful **CAS** in Line 70 we know all blocks in $n.\text{left.blocks}[1 \dots n.\text{left.head}_{83} - 1]$ are subblocks of $n.\text{blocks}[1 \dots n.\text{head}_{60}]$. Because of Observation 2 we have $n.\text{left.head}_{t^R} - 1 \leq n.\text{left.head}_{83} - 1$ and $n.\text{head}_{60} \leq n.\text{head}_{70}$. From Observation 13 the claim follows. The proof for the right child is the same. \square

Corollary 23. *If R is a complete instance of n .**Refresh**, then we have $EST_{n.\text{left}}^{t^R} \cup EST_{n.\text{right}}^{t^R} \subseteq EST_n^{Rt}$.*

Proof. The left-hand side is the same as Lemma 22, so it is sufficient to show when R terminates the established blocks in n are a superset of $n.\text{blocks}_{70}$. Line 70 writes the block **new** in $n.\text{blocks}[h]$ where h is value of $n.\text{head}$ read at Line 60. Because of Lemma 3 we are sure that $n.\text{head} > h$ when R terminates. So the block **new** appended to n at Line 70 is established at ${}^R t$. \square

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

Lemma 24. *Consider two consecutive terminating instances R_1, R_2 of **Refresh** on internal node n by process p . If neither R_1 nor R_2 is a successful **Refresh**, then we have $EST_{n.\text{left}}^{t^{R_1}} \cup EST_{n.\text{right}}^{t^{R_1}} \subseteq EST_n^{R_2 t}$.*

Proof. Let R_1 read i from $n.\text{head}$ at Line 60. By Lemma 3, R_1 and R_2 both cannot read the same value i . By Observation 2, R_2 reads a larger value of $n.\text{head}$ than R_1 .

Consider the case where R_1 reads i and R_2 reads $i+1$ from Line 60. As R_2 's CAS in Line 70 returns **false**, there is another successful instance R'_2 of $n.\text{Refresh}$ that has done a CAS successfully into $n.\text{blocks}[i+1]$ before R_2 tries to CAS. R'_2 creates its block **new** after reading the value $i+1$ from $n.\text{head}$ (Line 60) and R_1 reads the value i from $n.\text{head}$. By Observation 2 we have ${}^{R_1}t < t_{60}^{R_1} < t_{60}^{R'_2}$ (see Figure 18). By Lemma 22 we have $EST_{R'_2 t}^{n.\text{left}} \cup EST_{R'_2 t}^{n.\text{right}} \subseteq ops(n.\text{blocks}_{t_{70}^{R'_2}})$. Also by Lemma 3 on R_2 , the value of $n.\text{head}$ is more than $i+1$ after R_2 terminates, so the block appended by R'_2 into $n.\text{blocks}[i]$ is established by the time R_2 terminates. To summarize, ${}^{R_1}t$ is before R'_2 's read of $n.\text{head}$ ($t_{60}^{R'_2}$), so we have $EST_{n.\text{left}}^{t^{R_1}} \cup EST_{n.\text{right}}^{t^{R_1}} \subseteq ops(n.\text{blocks}_{t_{70}^{R'_2}})$. R'_2 's successful CAS ($t_{70}^{R'_2}$) is before R_2 's termination (t^{R_2}), so by Lemma 3 $n.\text{head}$ has been incremented when R_2 terminates and the block R'_2 put into n is established by then. So we have $ops(n.\text{blocks}_{t_{70}^{R'_2}}) \subseteq EST_n^{R_2 t}$.

If R_2 reads some value greater than $i+1$ in Line 60 it means $n.\text{head}$ has been incremented at least two times since ${}^{R_1}t$. By Invariant 4, when $n.\text{head}$ is incremented from $i+1$ to $i+2$, $n.\text{blocks}[i+1]$ is non-null. Let R_3 be the **Refresh** on n that has put the block in $n.\text{blocks}[i+1]$. R_3 read $n.\text{head} = i+1$ at Line 60 and has put its block in $n.\text{blocks}[i+1]$ before R_2 's read of $n.\text{head}$ at Line 60. So we have $t^{R_1} < {}^{R_3}t < {}^{R_3}t < t_{60}^{R_2} < {}^{R_2}t$. From Observation 13 on the operations before and after R_3 's CAS and Lemmas 22, 3 on R_3 the claim holds. \square

Corollary 25. $EST_{n.\text{left}}^{t_{52}^{52}} \cup EST_{n.\text{right}}^{t_{52}^{52}} \subseteq EST_n^{t_{53}^{53}}$

Proof. If the first **Refresh** in line 52 returns **true**, then by Corollary 23 the claim holds. If the first **Refresh** failed and the second **Refresh** succeeded, the claim still holds by Corollary 23. Otherwise, both failed, and the claim is satisfied by Lemma 24. \square

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

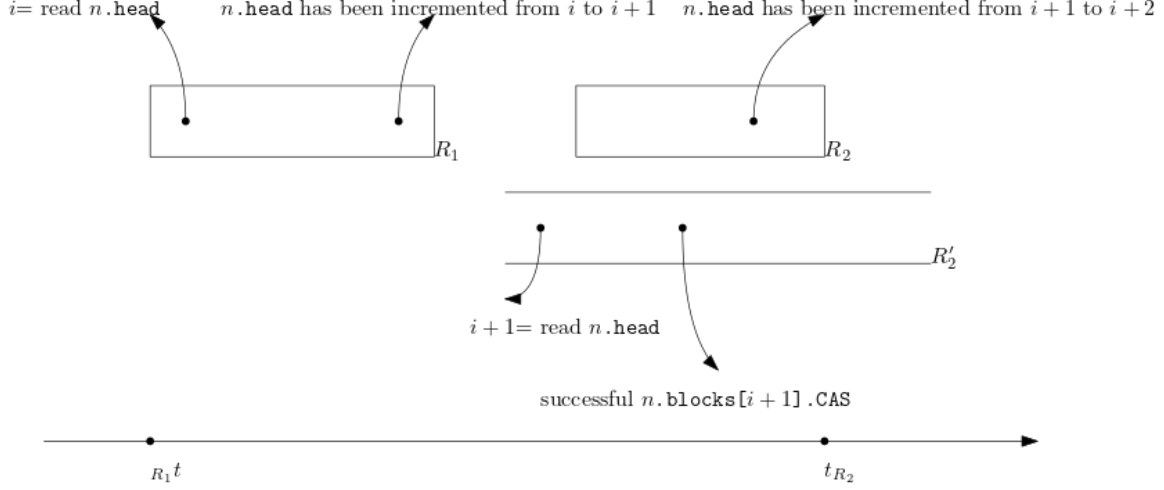


Figure 18: $R_1t < t_{60}^{R_1} < \text{incrementing } n.\text{head} \text{ from } i \text{ to } i+1 < t_{60}^{R'_2} < t_{70}^{R'_2} < \text{incrementing } n.\text{head} \text{ from } i+1 \text{ to } i+2 < t_{R_2}$

Corollary 26. For $A = l.\text{Append}(b)$ we have $\text{ops}(b) \subseteq \text{EST}_n^{t^A}$ for each node n in the path from l to root.

Proof. A adds b to the assigned leaf of the process, establishes it at Line 48 and then calls **Propagate** on the leaf's parent where it appended b . For every node n , $n.\text{Propagate}$ appends b to n , establishes it in n by Corollary 25 and then calls $n.\text{parent}.\text{Propagate}$ until n is root. \square

Corollary 27. After $l.\text{Append}(b)$ finishes, b is a subblock of exactly one block in each node along the path from l to the root.

Proof. By the previous corollary and Lemma 11 there is exactly one block in each node containing b . \square

4.4 Correctness of GetEnqueue

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

Lemma 28. *Each block contains at most one operation of each process and, therefore, at most p operations in total.*

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process p in block B in node n . Without loss of generality op_1 is invoked earlier than op_2 . Process p cannot invoke more than one operation concurrently, so op_1 has to be finished before op_2 begins. By Corollary 27, before op_2 calls `Append`, op_1 exists in every node of the tree on the path from p 's leaf to the root. Since b contains op_2 , it must be created after op_2 is invoked. The fact that op_2 .`Append` is invoked after op_1 .`Append` terminated means that there is some block B' in n before B that contains op_1 . The existence of op_1 in B and B' contradicts Lemma 11. \square

Lemma 29. *Each block has at most p direct subblocks.*

Proof. From the definition of operations in a block in Definition 10 we know the operations in a block are the union of the operations in the direct subblocks of the block. The claim follows from the equivalence of the number of operations in a block and the number of operations in the subblocks of the block and the observation that each block appended to an internal node contains at least one operation due to the test on Line 68. Also, blocks in the leaves contain only one `Enqueue` or `Dequeue` operation. \square

`DSearch(e , end)` returns a pair $\langle b, i \rangle$ such that the i th `Enqueue` in the b th block of the root is the e th `Enqueue` in the sequence stored in the root.

Lemma 30 (`DSearch` Correctness). *If $\text{root.blocks}[end] \neq \text{null}$ and $1 \leq e \leq \text{root.blocks}[end].\text{sum}_{\text{enq}}$, `DSearch(e , end)` returns $\langle b, i \rangle$ such that $E_i(\text{root}, b) = E_e(\text{root})$.*

Proof. From Lines 86 and 87 we know the $\text{sum}_{\text{enq-left}}$, and $\text{sum}_{\text{enq-right}}$ fields of `blocks` in each node are sorted in non-decreasing order. Since $\text{sum}_{\text{enq}} = \text{sum}_{\text{enq-left}} + \text{sum}_{\text{enq-right}}$, the sum_{enq} values of $\text{root.blocks}[0..end]$ are also non-decreasing. By Corollary 19 we know that sum_{enq} field in a block shows the prefix sum of the number of `Enqueue` operations in that block. Furthermore, since $\text{root.blocks}[0].\text{sum}_{\text{enq}} = 0$ and $\text{root.blocks}[end].\text{sum}_{\text{enq}} \geq e$, there is a b such that $\text{root.blocks}[b].\text{sum}_{\text{enq}} \geq e$ and $\text{root.blocks}[b-1].\text{sum}_{\text{enq}} < e$. Block $\text{root.blocks}[b]$ contains $E_i(\text{root}, b)$. Lines 38–41 doubles the search range in Line 40 and will

eventually reach `start` such that `root.blocks[start].sumenq ≤ e ≤ root.blocks[end].sumenq`. Then, in Line 42, the binary search finds the b such that `root.blocks[b-1].sumenq < e ≤ root.blocks[b].sumenq`. By Corollary 19, `root.blocks[b]` is the block that contains $E_e(\text{root})$. Finally, i is computed using the definition of `sumenq` and Corollary 19. \square

Lemma 31 (GetEnqueue correctness). *If $1 \leq i \leq n.\text{blocks}[b].\text{num}_{\text{enq}}$ then $n.\text{GetEnqueue}(b, i)$ returns $E_i(n, b).\text{element}$.*

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

For the induction step, we prove if $n.\text{dir}.\text{GetEnqueue}(b', i)$ returns $E_i(n.\text{dir}, b')$ then $n.\text{GetEnqueue}(b, i)$ returns $E_i(n, b)$. From Definition 15 of $E(n, b)$, we know that operations from the left subblocks come before the operations from the right subblocks in a block (see Figure 19). By Lemma 18, the `numenq-left` field in $n.\text{blocks}[b]$ is the number of `Enqueue` operations from the blocks' subblocks in the left child of n . So the i th `Enqueue` operation in $n.\text{blocks}[b]$ is propagated from the right child if and only if i is greater than $n.\text{blocks}[b].\text{num}_{\text{enq-left}}$. Line 98 decides whether the i th enqueue in the b th block of internal node n is in the left child or right child subblocks of $n.\text{blocks}[b]$. By Definitions 5 and 10 to find an operation in the subblocks of $n.\text{blocks}[i]$ we need to search in the range

$$\begin{aligned} & n.\text{left}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1..n.\text{blocks}[i].\text{end}_{\text{left}}] \text{ or} \\ & n.\text{right}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}+1..n.\text{blocks}[i].\text{end}_{\text{right}}]. \end{aligned}$$

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

Otherwise, the enqueue we are looking for is in the right child. Because `Enqueues` from the left subblocks are ordered before the `Enqueues` from the right subblocks, there are $n.\text{blocks}[b].\text{num}_{\text{enq-left}}$ enqueues

ahead of $E_i(n, b)$ from the left child. So we need to search for $i - n.\text{blocks}[b].\text{num}_{\text{enq-left}} + n.\text{blocks}[b - 1].\text{sum}_{\text{enq-right}}$ in the right child (Line 103). Other parameters for the right child are chosen similarly to the left child.

So, in both cases, the direct subblock containing $E_i(n, b)$ is computed in Line 99 or 103. Let *subblock* be the block in $n.\text{dir}$ containing $E_i(n, b)$. Finally, $n.\text{child}.\text{GetEnqueue}(\text{subblock}, i)$ is invoked and it returns $E_i(n, b).\text{element}$ by the hypothesis of the induction. \square

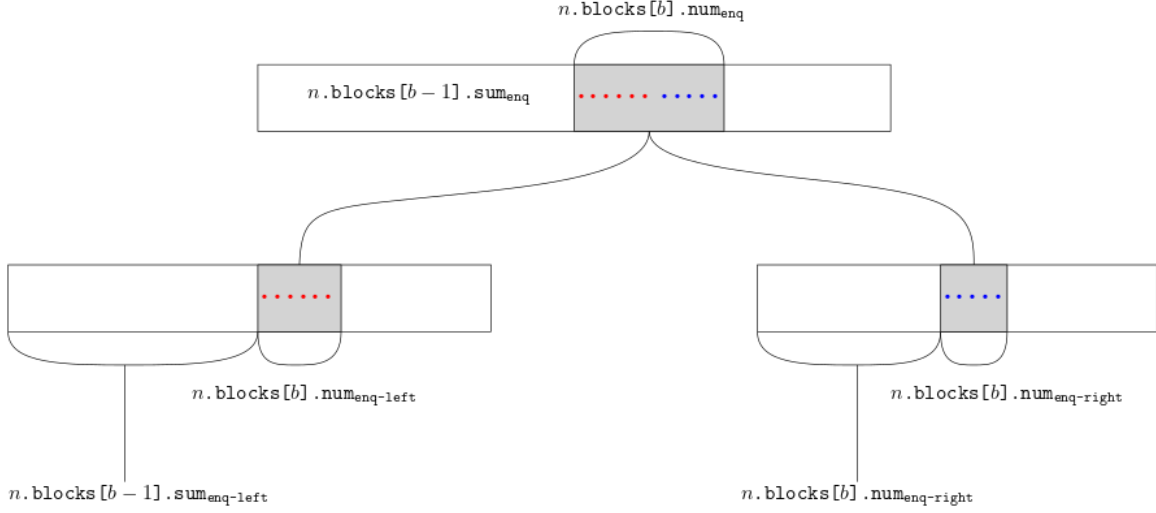


Figure 19: The number and order of the enqueue operations propagated from the left and the right child to $n.\text{blocks}[b]$. Both $n.\text{blocks}[b]$ and its subblocks are shown in grey. **Enqueue** operations from the left subblocks (colored red), are ordered before the **Enqueue** operations from the right child (colored blue).

4.5 Correctness of IndexDequeue

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

Definition 32. If a `Refresh` instance R_1 does its CAS at Line 70 earlier than `Refresh` instance R_2 we say R_1 has *happened before* R_2 .

Observation 33. After $n.\text{blocks}[i].\text{CAS}(\text{null}, B)$ succeeds, $n.\text{head}$ cannot increase from i to $i + 1$ until $B.\text{super}$ is set.

Proof. From Observation 2 we know the $n.\text{head}$ changes only by the increment on Line 77. Before an instance of `Advance` increments $n.\text{head}$ on Line 77, Line 76 ensures that $n.\text{blocks}[\text{head}].\text{super}$ was set at Line 76. □

Corollary 34. If $n.\text{blocks}[i].\text{super}$ is null, then $n.\text{head} \leq i$ and $n.\text{blocks}[i + 1]$ is null.

Proof. By Invariant 4 and Observation 33. □

Now let us consider how the `Refreshes` that took place on the parent of node n after block B was stored in n will help to set $B.\text{super}$ and propagate B to the parent.

Observation 35. If the block created by an instance R_p of $n.\text{parent.Refresh}$ contains block $B = n.\text{blocks}[b]$ then R_p reads a value greater than b from $n.\text{head}$ in Line 83.

Lemma 36. If $B = n.\text{blocks}[b]$ is a direct subblock of $n.\text{parent.blocks}[sb]$ then $B.\text{super} \leq sb$.

Proof. Let R_p be the instance of $n.\text{parent.Refresh}$ that does a successful CAS on $n.\text{parent.blocks}[sb]$. By Observation 35 if R_p propagates B it has to read a greater value than b from $n.\text{head}$, which means $n.\text{head}$ was incremented from b to $b + 1$ in Line 77. By Observation 33 $B.\text{super}$ was already set in Line 76. The value written in $B.\text{super}$, was read in Line 75 before the CAS that sets $B.\text{super}$ in Line 76. From Observation 2 we know $n.\text{parent.head}$ is non-decreasing so $B.\text{super} \leq sb$, since $n.\text{parent.head}$ is still equal to sb when R_p executes its CAS at Line 70 by Lemma 6. The reader may wonder when the case $b.\text{super} = sb$ happens. This can happen when $n.\text{parent.blocks}[B.\text{super}] = \text{null}$ when $B.\text{super}$ is written and R_p puts its created block into $n.\text{parent.blocks}[B.\text{super}]$ afterwards. □

Lemma 37. *Let R_n be a Refresh that puts B in $n.\text{blocks}[b]$ at Line 70. Then, the block created by one of the next two successful $n.\text{parent.Refreshes}$ according to Definition 32 contains B and $B.\text{super}$ is set when the second successful $n.\text{parent.Refresh}$ reaches Line 67.*

Proof. Let R_{p1} be the first successful $n.\text{parent.Refresh}$ after R_n and R_{p2} be the second next successful $n.\text{parent.Refresh}$. To derive a contradiction assume B was neither propagated to $n.\text{parent}$ by R_{p1} nor by R_{p2} .

Since R_{p2} 's created block does not contain B , by Observation 35 the value R_{p2} reads from $n.\text{head}$ in Line 83 is at most b . From Observation 2 the value R_{p2} reads in Line 62 is also at most b .

R_n puts B into $n.\text{blocks}[b]$ so R_n reads the value b from $n.\text{head}$. Since R_{p2} 's CAS into $n.\text{parent.blocks}$ is successful there should be a Refresh instance R'_p on $n.\text{parent}$ that increments $n.\text{parent.head}$ (Line 77) after R_{p1} 's Line 70 and before R_{p2} 's Line 60. We assumed $t_{70}^{R_n} < t_{70}^{R_{p1}} < t_{70}^{R_{p2}}$ by Definition 32. Finally, Line 62 is after Line 60 and R_{p2} 's Line 60 is after R'_p 's Line 77, which is after R_n 's $n.\text{blocks.CAS}$.

$$\left. \begin{array}{l} R_n t_{70} <_{R_{p1}} t \\ R_{p1} t_{70} <_{R'_p} t <_{R_{p2}} t \\ R_{p2} t_{60} <_{R_{p2}} t \end{array} \right\} \Rightarrow R_n t_{70} <_{R_{p2}} t$$

So R_{p2} reads a value greater than or equal to b for $n.\text{head}$ by Observation 2.

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

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

Corollary 38. *If $B = n.\text{blocks}[b]$ is propagated to $n.\text{parent}$, then $B.\text{super}$ is equal to or one less than the index of the superblock of B .*

Proof. Let R_n be the $n.\text{Refresh}$ that put B in $n.\text{blocks}$ and let R_{p1} be the first successful $n.\text{parent.Refresh}$ after R_n and R_{p2} be the second next successful $n.\text{parent.Refresh}$. Before B can be propagated to n 's parent, $n.\text{head}$ must be greater than b , so by Observation 33 $B.\text{super}$ is set. From the previous lemma we know that B is propagated by the second next successful Refresh's CAS on $n.\text{parent.blocks}$. To summarize, we have $n.\text{parent.head}_{R_{p2} t_{70}} = n.\text{parent.head}_{R_{p1} t_{70}} + 1$ and by Definition 32 and Observation 2

$n.\text{parent}.\text{head}_{\frac{R_{p^1}t}{70}} \leq n.\text{parent}.\text{head}_{\frac{R_nt}{70}}$. The value that is set in $B.\text{super}$ is read from $n.\text{parent}.\text{head}$ after $\frac{R_nt}{70}$. So $B.\text{super}$ is equal to or one less than the index of the superblock of B . \square

We prove `IndexDequeue`'s correctness using Corollary 38 on each step of the `IndexDequeue`.

Lemma 39 (`IndexDequeue` correctness). *If $1 \leq i \leq n.\text{blocks}[b].\text{num_deq}$ then $n.\text{IndexDequeue}(b, i)$ returns $\langle x, y \rangle$ such that $D_i(n, b) = D_y(\text{root}, x)$.*

Proof. We will prove this by induction on the distance of n from the `root`. The base case where n is `root` is trivial (see Line 109). For the non-root nodes $n.\text{IndexDequeue}(b, i)$ computes sb , the index of the superblock of the b th block in n , in Line 112 by Corollary 38. After that, the position of $D_i(n, b)$ in $D(n.\text{parent}, sb)$ is computed in Lines 113–118. By Definition 15, `Dequeues` in a block are ordered based on the order of its subblocks from left to right. If $D_i(n, b)$ was propagated from the left child, the number of dequeues in the left subblocks of $n.\text{parent}.\text{blocks}[sb]$ before $n.\text{blocks}[b]$ is considered in Line 114 (see Figure 20). Otherwise, if $D_i(n, b)$ was propagated from the right child, the number of dequeues in the subblocks from the left child is considered to be ahead of the computed index (Line 115) (see Figure 21). Finally, `IndexDequeue` is called on $n.\text{parent}$ recursively, and it returns the correct response by the induction hypothesis. \square

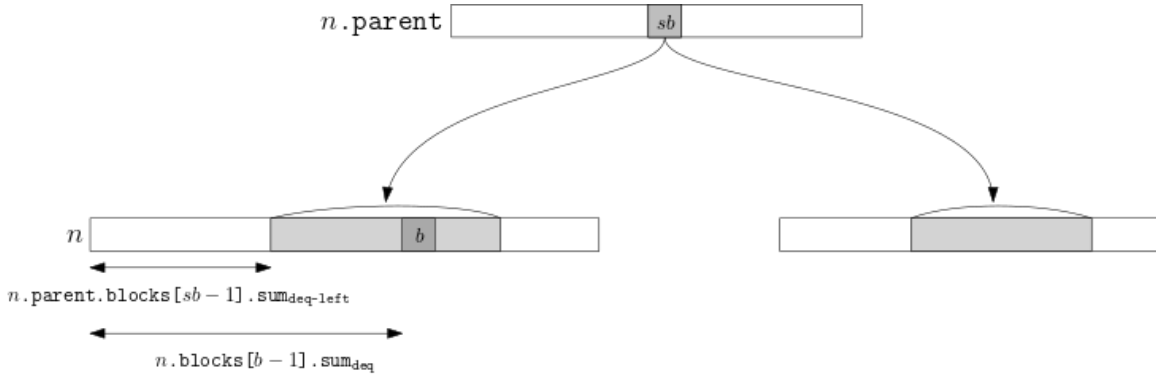


Figure 20: The number of `Dequeue` operations before $E_i(n, b)$ shown in the case where n is a left child.

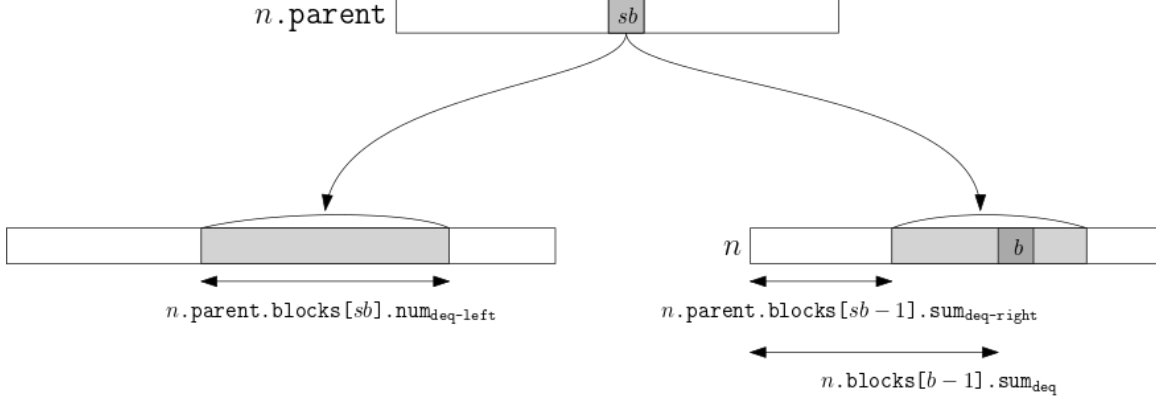


Figure 21: The number of **Dequeue** operations before $E_i(n, b)$ shown in the case where n is a right child.

4.6 Linearizability

We now prove the two properties needed for linearizability.

Lemma 40. *L is a legal linearization ordering.*

Proof. We must show for any execution that every operation that terminates is in L exactly once. Also, if op_1 terminates before op_2 starts in the execution, then op_1 is before op_2 in the linearization. The first claim is directly reasoned from Corollary 27. For the latter, if op_1 terminates before op_2 starts, op_1 .**Append** has terminated before op_2 .**Append** started. From Corollary 26, op_1 is in **root.blocks** before op_2 starts to propagate. By definition of L , op_1 is linearized before op_2 . \square

Once some operations are aggregated in one block, they will get propagated up to the root together, and they can be linearized in any order among themselves. We have chosen to put **Enqueues** in a block before **Dequeues** (see Definition 15).

Definition 41. If a **Dequeue** operation returns **null** it is called a *null Dequeue*, otherwise it is called *non-null Dequeue*.

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

Definition 42. Assume the operations in **root.blocks** are applied sequentially on an empty queue in the order of L . $Resp(d) = e.element$ if the element of **Enqueue** e is the response to **Dequeue** d . Otherwise if d is a **null dequeue** then $Resp(d) = null$.

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

Lemma 43. `root.blocks[b].size` is the size of the queue if the operations in `root.blocks[0...b]` are applied in the order of L .

Proof. We prove the claim by induction on b . The base case when $b = 0$ is trivial since the queue is initially empty and `root.blocks[0]` contains an empty block with `size` field equal to 0. We are going to show the correctness when $b = i$ assuming correctness when $b = i - 1$. By Definition 15 **Enqueue** operations come before **Dequeue** operations in a block in L . By Lemma 18 `numenq` and `numdeq` fields in a block show the number of **Enqueue** and **Dequeue** operations in it. If there are more than `root.blocks[i - 1].size + root.blocks[i].numenq` dequeue operations in `root.blocks[i]` then the queue would become empty after `root.blocks[i]`. Otherwise, the size of the queue after the b th block in the root is `root.blocks[b - 1].size + root.blocks[b].numenq - root.blocks[b].numdeq`. In both cases, this is the same as the assignment on Line 91. \square

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

Lemma 44. If operations in the root are applied in the order of L , the number of non-null **Dequeues** in `root.blocks[0...b]` is `root.blocks[b].sumenq - root.blocks[b].size`.

Proof. There are `root.blocks[b].sumenq` **Enqueue** operations in `root.blocks[0...b]` by Corollary 19. The size of the queue after doing `root.blocks[0...b]` in the order of L is the number of *enqueues* in `root.blocks[0...b]` minus the number of *non-null Dequeues* in `root.blocks[0...b]`. By the correctness of the `size` field from Lemma 43 and `sumenq` field from Lemma 18, the number of *non-null Dequeues* is `root.blocks[b].sumenq - root.blocks[b].size`. \square

Corollary 45. If operations in the root are applied in the order of L , the number of non-null dequeues in `root.blocks[b]` is `root.blocks[b].numenq - root.blocks[b].size + root.blocks[b - 1].size`.

Lemma 46. $\text{Resp}(D_i(\text{root}, b))$ is null iff `root.blocks[b - 1].size + root.blocks[b].numenq - i < 0`.

Proof. From Corollary 45 and Lemma 18. \square

Lemma 47. `FindResponse(b, i)` returns $\text{Resp}(D_i(\text{root}, b))$.

Proof. $D_i(\text{root}, b)$ is $D_{\text{root.blocks}[b-1].\text{sum}_{\text{deq}}+i}(\text{root})$ by Definition 15 and Lemma 19. $D_i(\text{root}, b)$ returns null at Line 20 if `root.blocks[b - 1].size + root.blocks[b].numenq - i < 0` and $\text{Resp}(D_i(\text{root}, b)) = \text{null}$ in this case by Lemma 46. Otherwise, if $D_i(\text{root}, b)$ is the i th non-null dequeue in L it should return the i th

enqueued value. By Lemma 44 there are $\text{root.blocks}[b-1].\text{sum}_{\text{enq}} - \text{root.blocks}[b-1].\text{size}$ non-null Dequeue operations in $\text{root.blocks}[0 \dots b-1]$. The Dequeues in $\text{root.blocks}[b]$ before $D_i(\text{root}, b)$ are non-null dequeues. So $D_i(\text{root}, b)$ is the e th non-null Dequeue where $e = i + \text{root.blocks}[b-1].\text{sum}_{\text{deq}} - \text{root.blocks}[b-1].\text{size}$ (Line 22). See Figure 22.

After computing e at Line 22, the code finds b, i such that $E_i(\text{root}, b) = E_e(\text{root})$ using DSearch and then finds its element using GetEnqueue (Line 23). Correctness of DSearch and GetEnqueue routines are shown in Lemmas 30 and 31. \square

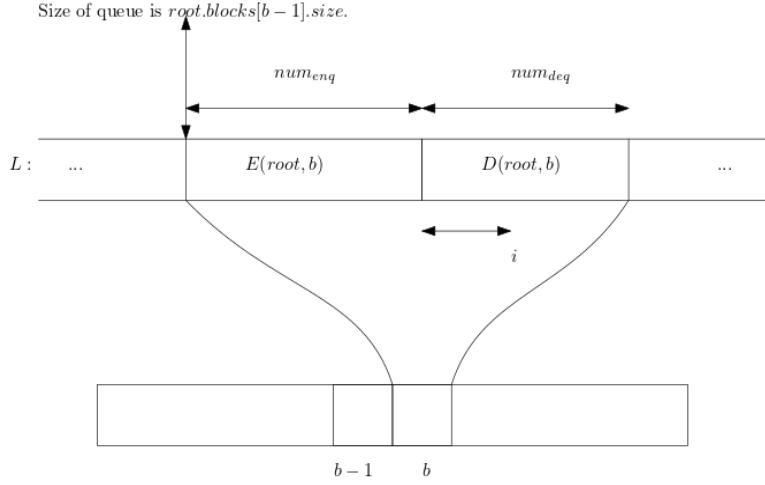


Figure 22: The position of $D_i(\text{root}, b)$.

Lemma 48. *The responses to operations in our algorithm are the same as in the sequential execution in the order given by L .*

Proof. Enqueue operations do not return any value. By Lemma 47, the response of a Dequeue in our algorithm is same as the response from the sequential execution of L . \square

Theorem 49 (Main). *The queue implementation is linearizable.*

Proof. The theorem follows from Lemmas 40 and 48. \square

Remark In fact our algorithm is strongly linearizable as defined in [10]. By Definition 15 the linearization ordering of operations will not change as blocks containing new operations are appended to the root.

5 Analysis

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

Proposition 50. An Enqueue or Dequeue operation does at most $14 \log p$ CAS operations.

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

Lemma 51 (DSearch Analysis). *If the element enqueued by $E_i(\text{root}, b) = E_e(\text{root})$ is the response to some Dequeue operation in $\text{root.blocks}[end]$, then $\text{DSearch}(e, end)$ takes $O(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[end].\text{size}))$ steps.*

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

So, the doubling search reaches **start** such that the $\text{root.blocks}[\text{start}].\text{sum}_{\text{enq}}$ is less than e in $O(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[end].\text{size}))$ steps. See Figure 23. After Line 41, the binary search that finds b also takes $O(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[end].\text{size}))$. Next, i is computed via the definition of sum_{enq} in constant time (Line 43). \square

Lemma 52 (Worst Case Time Analysis). *The worst case number of steps for an Enqueue is $O(\log^2 p)$ and for a Dequeue, is $O(\log^2 p + \log q_e + \log q_d)$, where q_d is the size of the queue when the Dequeue is linearized*

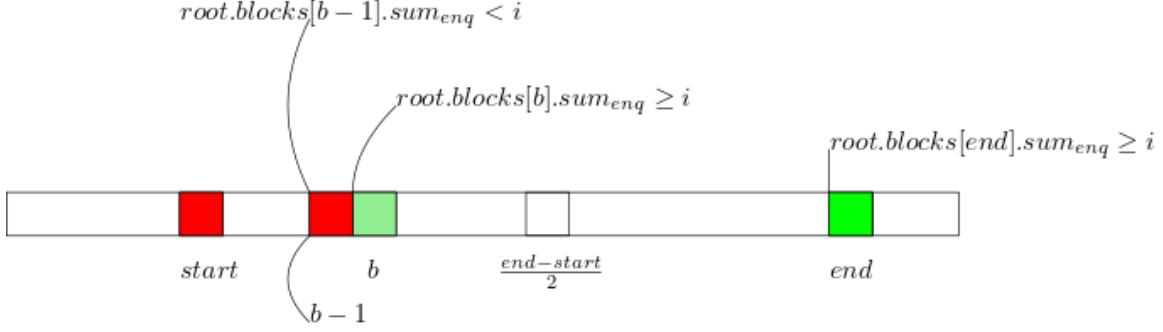


Figure 23: Distance relations between `start`, `b`, `end`.

and q_e is the size of the queue at the time the response of the `Deque` is linearized.

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

A `Deque` creates a block whose `element` is `null`, appends it to the tree, computes its rank among non-null dequeues, finds the corresponding enqueue and returns the response. The first two parts are similar to an `Enqueue` operation and take $O(\log p)$ steps. To compute the rank of a `Deque` in $D(n)$, the `Deque` calls `IndexDeque()`. `IndexDeque` does $O(1)$ steps in each level which takes $O(\log p)$ steps. If the response to the `Deque` is `null`, `FindResponse` returns `null` in $O(1)$ steps. Otherwise, if the response to a dequeue in `root.blocks[end]` is in `root.blocks[b]` the `DSearch` takes $\Theta(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[end].\text{size}))$ by Lemma 51, which is $O(\log \text{size of the queue when Enqueue is linearized}) + \log \text{size of the queue when Dequeue is linearized})$. Each search in `GetEnqueue()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma 29), so `GetEnqueue()` takes $O(\log^2 p)$ steps. \square

Lemma 53 (Amortized Time Analysis). *The amortized number of steps for an `Enqueue` or `Deque`, is $O(\log^2 p + \log q)$, where q is the size of the queue when the operation is linearized.*

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

Theorem 54. *The queue implementation is wait-free.*

Proof. To prove the claim, it is sufficient to show that every `Enqueue` and `Deque` operation terminates after a finite number of its own steps. This is directly concluded from Lemma 52. \square

6 Future Directions

We designed a tree to achieve agreement on a linearization of operations invoked by p processes in an asynchronous model, which we will call a *block tree*. We implemented two queries to know information about the ordering agreed in the block tree. Then we used the tree to implement a queue where the number of steps per operation is poly-logarithmic with respect to the size of the queue and the number of processes. Block trees can be used as a mechanism to achieve agreement among processes to construct more poly-logarithmic wait-free linearizable objects. In the next paragraphs, we talk about possible improvements on block trees and the data structures that we can implement with block trees.

Reducing Space Usage The `blocks` arrays defined in our algorithm are unbounded. To use $O(n)$ space in each node where n is the total number of operations, instead of unbounded arrays, we could use the memory model of the wait-free vector introduced by Feldman, Valera-Leon and Damian [8]. We can create an array called `arr` of pointers to array segments (see Figure 24). When a process wishes to write into location `head` it checks whether `arr[[log head]]` points to an array or not. If not, it creates a shared array of size $2^{\lceil \log \text{head} \rceil}$ and tries to `CAS` a pointer to the created array into `arr[[log head]]`. Whether the `CAS` is successful or not, `arr[[log head]]` points to an array. When a process wishes to access the i th element it looks up `arr[[log i]]` `[i - 2⌊log i⌋]`, which takes $O(1)$ steps. The CAS Retry Problem does not happen here because if n elements are appended to the array, then only $O(p \times \log n)$ `CAS` steps have happened on the array `arr`. Furthermore, at most p arrays with size $2^{\lceil \log i \rceil}$ are allocated by processes while processes try to do the `CAS` on `arr[i]`. Jayanti and Shun [17] present a way to initialize wait-free arrays in constant steps. The time taken to allocate arrays in an execution containing n operations is $O(\frac{p \log n}{n})$ per operation, which is negligible if $n \gg p$. The vector implementation also has a mechanism for doubling `arr` when necessary, but this happens very rarely since increasing `arr` from s to $2s$ increases the capacity of the vector from 2^s to 2^{2s} .

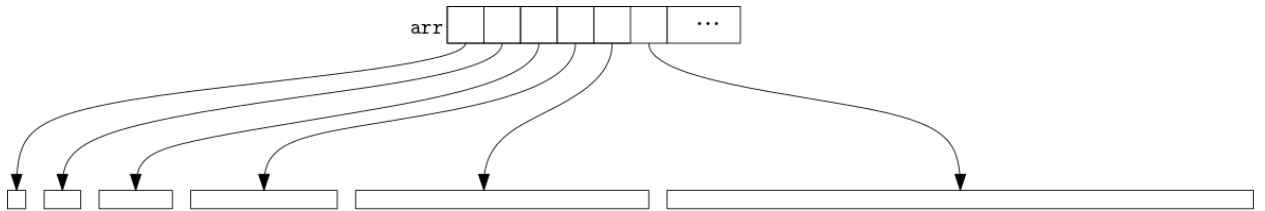


Figure 24: Array Segments

Garbage Collection We did not handle garbage collection: **Enqueue** operations remain in the nodes even after their elements have been dequeued. We can keep track of the **blocks** in the **root** whose operations are all terminated, i.e., all of its enqueues have been dequeued, and the responses of all of its dequeues have been computed. We call these blocks *finished blocks*. If we help the operations of all processes to compute their responses, then we can say if block B is finished, then all blocks before B are also finished. Knowing the most recent finished block in a node, we can reclaim the memory taken by finished blocks. We cannot use arrays (or vectors) to throw the garbage blocks away. We need a data structure that supports **tryAppend()**, **read(i)**, **write(i)** and **split(i)** operations in $O(\log n)$, where **split(i)** removes all the indices less than i . If each process tries to do the garbage collection every p^2 operation on the queue, then the amortized complexity remains the same. We can use a concurrent implementation of persistent red-black trees for this [25]. Bashari and Woelfel [3] used persistent red-black trees in a similar way.

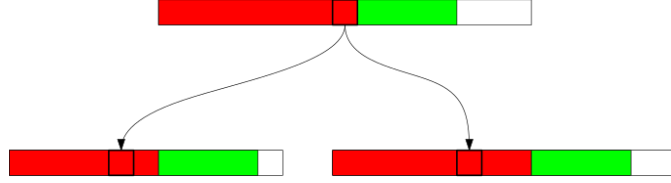


Figure 25: Finished blocks are shown with red color and unfinished blocks are shown with green color. All the subblocks of a finished block are also finished.

Poly-logarithmic Wait-free Sequences Consider a data structure storing a sequence that supports three operations **append(e)**, **get(i)** and **index(e)**. An **append(e)** adds e to the end of the sequence, a **get(i)** gets the i th element in the sequence and an **index(e)** computes the position of element e in the sequence. We can modify our queue to design such a data structure. An **append(e)** is implemented like **Enqueue(e)**, **get(i)** is done by calling **DSearch** but with a **BinarySearch** on the entire **root.blocks** array and **index(e)** is done similarly to **IndexDequeue** (except operating on enqueues instead of dequeues). We achieve this with poly-logarithmic steps for each operation with respect to the number of **appends** done.

Other Poly-log Wait-free Data structures There are two reasons the block tree worked well to implement a queue. Firstly, to respond to a **Dequeue** we do not need to look at the entire history of operations: if a **Dequeue** does not return **null**, we can compute the index of the **Enqueue** that is its response in $O(\log n)$ time if we keep the number of enqueues and the size. Secondly, the operations we need to search to respond

to the `Dequeue` are not very far from it in the sequence of operations: the distance is at most linear in the size of the queue. Creating a wait-free poly-logarithmic implementation of other objects whose operations satisfy these two conditions may be possible.

6.1 Attiya–Fouren Lower Bound

As we talked about earlier in Section 2.4 the Attiya–Fouren lower bound says that a concurrent implementation of queues using reads, writes and conditional operations like `CAS` has $O(\sqrt{\log \log p})$ amortized complexity [2] when the number of concurrent processes is $O(\log \log p)$. Our amortized complexity is $O(\log^2 p + \log q)$. It is an open problem to reduce the gap between our algorithm and the Attiya–Fouren lower bound.

References

- [1] Afek, Y., Dauber, D., and Touitou, D. Wait-free made fast (extended abstract). In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA* (1995), F. T. Leighton and A. Borodin, Eds., ACM, pp. 538–547.
- [2] Attiya, H., and Fouren, A. Lower bounds on the amortized time complexity of shared objects. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017* (2017), J. Aspnes, A. Bessani, P. Felber, and J. Leitão, Eds., vol. 95 of *LIPICs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 16:1–16:18.
- [3] Bashari, B., and Woelfel, P. An efficient adaptive partial snapshot implementation. In *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021* (2021), A. Miller, K. Censor-Hillel, and J. H. Korhonen, Eds., ACM, pp. 545–555.
- [4] Chandra, T. D., Jayanti, P., and Tan, K. A polylog time wait-free construction for closed objects. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998* (1998), B. A. Coan and Y. Afek, Eds., ACM, pp. 287–296.
- [5] Colvin, R., and Groves, L. Formal verification of an array-based nonblocking queue. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China* (2005), IEEE Computer Society, pp. 507–516.
- [6] David, M. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings* (2004), R. Guerraoui, Ed., vol. 3274 of *Lecture Notes in Computer Science*, Springer, pp. 132–143.
- [7] Ellen, F., and Woelfel, P. An optimal implementation of fetch-and-increment. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205* (Berlin, Heidelberg, 2013), DISC 2013, Springer-Verlag, p. 284–298.
- [8] Feldman, S., Valera-Leon, C., and Dechev, D. An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2016), 654–667.

- [9] Gidenstam, A., Sundell, H., and Tsigas, P. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings* (2010), C. Lu, T. Masuzawa, and M. Mosbah, Eds., vol. 6490 of *Lecture Notes in Computer Science*, Springer, pp. 302–317.
- [10] Golab, W. M., Higham, L., and Woelfel, P. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011* (2011), L. Fortnow and S. P. Vadhan, Eds., ACM, pp. 373–382.
- [11] Hendler, D., Incze, I., Shavit, N., and Tzafrir, M. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010* (2010), F. M. auf der Heide and C. A. Phillips, Eds., ACM, pp. 355–364.
- [12] Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (jan 1991), 124–149.
- [13] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492.
- [14] Hoffman, M., Shalev, O., and Shavit, N. The baskets queue. In *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings* (2007), E. Tovar, P. Tsigas, and H. Fouchal, Eds., vol. 4878 of *Lecture Notes in Computer Science*, Springer, pp. 401–414.
- [15] Jayanti, P. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998* (1998), B. A. Coan and Y. Afek, Eds., ACM, pp. 201–210.
- [16] Jayanti, P., and Petrovic, S. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings* (2005), R. Ramanujam and S. Sen, Eds., vol. 3821 of *Lecture Notes in Computer Science*, Springer, pp. 408–419.

- [17] Jayanti, S., and Shun, J. Fast arrays: Atomic arrays with constant time initialization. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)* (2021), S. Gilbert, Ed., vol. 209 of *LIPICs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 25:1–25:19.
- [18] Kogan, A., and Petrank, E. Wait-free queues with multiple enqueueers and dequeuers. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011* (2011), C. Cascaval and P. Yew, Eds., ACM, pp. 223–234.
- [19] Ladan-Mozes, E., and Shavit, N. An optimistic approach to lock-free FIFO queues. *Distributed Comput.* 20, 5 (2008), 323–341.
- [20] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996* (1996), J. E. Burns and Y. Moses, Eds., ACM, pp. 267–275.
- [21] Milman, G., Kogan, A., Lev, Y., Luchangco, V., and Petrank, E. BQ: A lock-free queue with batching. In *Proceedings of the 30th Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018* (2018), C. Scheideler and J. T. Fineman, Eds., ACM, pp. 99–109.
- [22] Moir, M., Nussbaum, D., Shalev, O., and Shavit, N. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA* (2005), P. B. Gibbons and P. G. Spirakis, Eds., ACM, pp. 253–262.
- [23] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23-27, 2013* (2013), A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, Eds., ACM, pp. 103–112.
- [24] Nikolaev, R., and Ravindran, B. wCQ: A fast wait-free queue with bounded memory usage. *CoRR abs/2201.02179* (2022).

- [25] Okasaki, C. Functional data structures. In *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text* (1996), J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129 of *Lecture Notes in Computer Science*, Springer, pp. 131–158.
- [26] Shafiei, N. Non-blocking array-based algorithms for stacks and queues. In *Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3-6, 2009. Proceedings* (2009), V. K. Garg, R. Wattenhofer, and K. Kothapalli, Eds., vol. 5408 of *Lecture Notes in Computer Science*, Springer, pp. 55–66.
- [27] Tsigas, P., and Zhang, Y. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2001, Heraklion, Crete Island, Greece, July 4-6, 2001* (2001), A. L. Rosenberg, Ed., ACM, pp. 134–143.