A Wait-free Queue with Poly-logarithmic Worst-case Step Complexity

Hossein Naderibeni

A Thesis submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Graduate Program in Computer Science,
York University
Toronto, Ontario

November, 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, called the CAS Retry Problem. We show that our algorithm avoids this problem with the helping mechanism which we use 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.

Acknowledgements

First, I want to thank my supervisor Eric Ruppert. Without his guidance and support, this thesis probably would not have been completed. He helped me in every step of this thesis, taught me shared data structures, helped me find the problem, design and improve the algorithm, and write it with its complications because of lots of new notions in it. Thank you again for always pushing to make it better.

I would also like to thank my co-supervisor Franck van Breugel, not only for his careful reading of my thesis and his thoughtful comments but also for his academic advice and motivations.

Besides my supervisors, I am grateful to all members of my examining committee for their time and their helpful comments on my thesis.

Last but not least, I would like to thank my parents for providing me with unfailing support and continuous encouragement throughout my years of study. I want to thank my brother for being there for me whenever I needed help.

I would like to express my gratitude to York University for the scholarship I was awarded in 2021-2022 academic year.

I dedicate this paper to the people in Iran and all the people who are fighting for their freedom.

Table of Contents

List of Tables

List of Figures

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. 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, which makes the queue inconsistent. 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 based on how powerful objects are to solve the consensus problem among processes. Objects like LL/SC or 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.

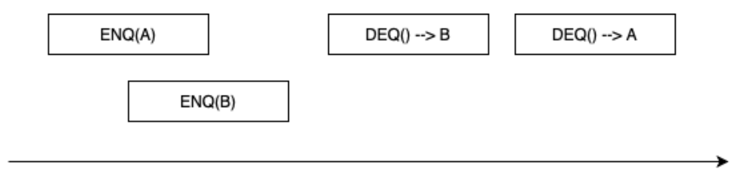

Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

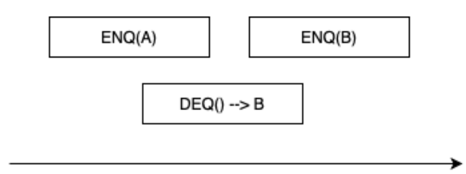

Figure 2: An example of an execution on an empty queue that is not linearizable. `Enqueue(A)` has terminated before `Enqueue(B)` is invoked and the `Dequeue` can occur before `Enqueue(A)` and return `null` or can occur after `Enqueue(A)` and return `A`.

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 one step further up the tree 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 the related work done in the area and the motivation for our implementation. Previous lock-free queues and their common problem are 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 and there is an example of a concurrent execution of the algorithm described in Section 3.3.

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

Chapter 5 is devoted to a complexity analysis for our implementation. We compute 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. The amortized step complexity of all the lock-free queues includes an $\Omega(p)$ term that comes from the worst case, where all p processes try to do an enqueue or a dequeue simultaneously. Morrison and Afek call this the *CAS retry problem* [24].

2.1 List-based Queues

Michael and Scott [21] 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 the 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.

Moir, Nussbaum, and Shalev [23] 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

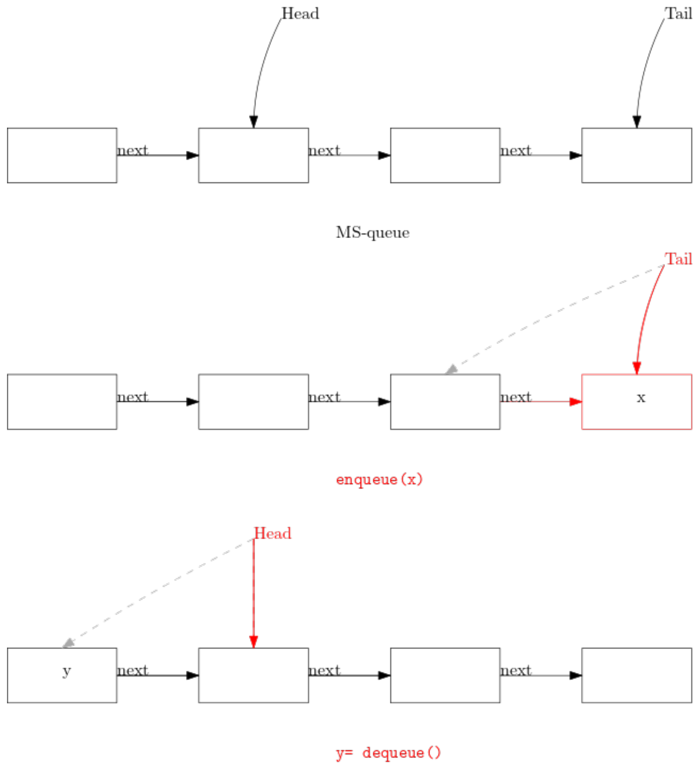

Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram, the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

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.

Ladan-Mozes and Shavit [20] 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 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 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

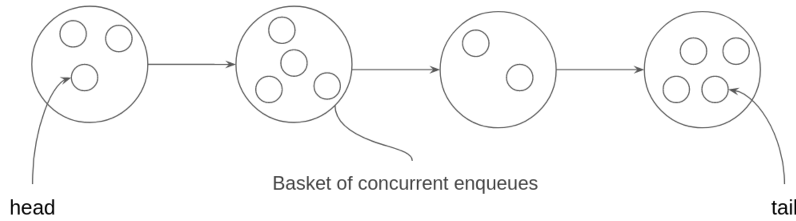

Figure 4: At some time, all operations in a basket ran concurrently, but only one succeeded to do CAS. To order the operations in a basket, processes perform another CAS. The successful process will be the next one in the basket, and so on.

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 amortized step (and CAS) complexity remains $\Omega(p)$. Their design is lock-free but not wait-free.

Kogan and Petrank [18] introduced wait-free queues based on the MS-queue and use Herlihy's helping technique [12] to achieve wait-freedom. Their amortized step complexity is $\Omega(p)$ because of the helping mechanism.

Milman et al. [22] designed a lock-free queue supporting futures. In their queue, operations

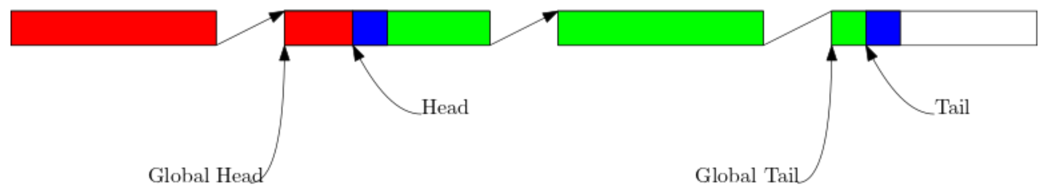

Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

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 [25] present a wait-free queue that uses the fast-path slow-path methodology introduced by Kogan and Petrank [19]. 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. The fast path ensures good performance while the slow path ensures termination. 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 amortized complexity cannot be better than $\Omega(p)$.

The CAS Retry Problem is not limited to list-based queues; array-based queues also share it [5, 27, 28]. 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. The first universal construction was 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 $\mathrm{O}(\log^2 p)$ shared accesses [4]. However, their algorithm cannot be used to create a queue. We mention a non-practical universal construction with a poly-log number of `CAS` instructions in the last paragraph of 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 a Fetch&Inc it adds its 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 of the similarities between Fetch&Inc objects and queues. Also, it is one of the few wait-free data structures achieving poly-logarithmic complexity.

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 $\Omega(c)$ amortized complexity [2]. The surprising point is that since their result is about when contention is low, their lower bound does not contradict our algorithm's complexity 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 to agree on one total ordering of the operations invoked by processes. The tree is called a *tournament tree* (see Figure 6). 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.

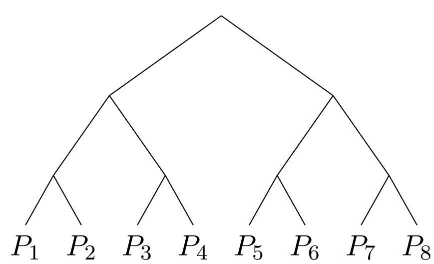

Figure 6: Each of the processes $P_1, P_2, ..., P_p$ has a leaf and in each node there is an ordering of operations stored. Each process tries to propagate its operations up to the root, which stores a total ordering of all operations.

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 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.

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

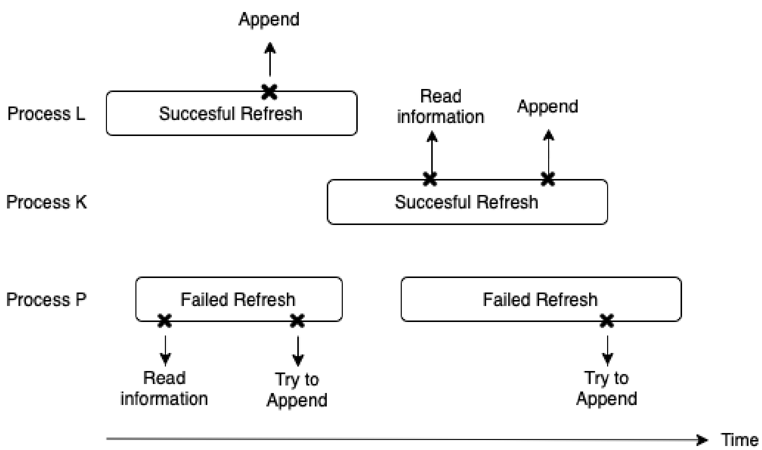

Figure 8: Time relations between the concurrent successful `Refresh`es and the two consecutive failed `Refresh`es.

We use `CAS` (Compare&Swap) instructions to implement the `Refresh`'s attempt to append described in the previous paragraph. The second failed `Refresh` of P is assuredly concurrent with a successful `Refresh` that has read its information after the invocation of the first failed `Refresh` (see Figure 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 that were in n's children before P's first failed `Refresh`, in case they were not already appended. After a process appends its operation into its leaf it can call `Refresh` on the path up to the root at most two times on each node. So, with $O(\log p)$ `CAS`es 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` instructions on fixed-size objects in each `Refresh`. To do that, we introduce *block*s. A block stores information about the operations propagated by a `Refresh`. It contains the number of operations from the left and the right child propagated to the node by the `Refresh` procedure. See Figure 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 the children into a new block and stores the new block in the parent. We call the aggregated blocks *subblocks* of the new block and the new block the *superblock* of them. In each `Refresh` there is at most one operation from each process trying to be propagated, because one process cannot invoke two operations concurrently. Thus, there are at most p operations in a block. Furthermore, since the operations in a `Refresh` step are concurrent we can linearize them among themselves in any order we wish, because if two operations are read in one successful `Refresh` step in a node they are going to be propagated up to the root together. Our choice is to put the operations propagated from the left child before the operations propagated from the right child. In this way, if we know the number of operations from the left child and the number of operations from the right child in a block, we have a complete ordering of the operations.

So far, we have a shared tree that processes use to agree on the implicit ordering stored in its root. With this agreement on the linearization ordering, we can design a universal construction; for a given object, we can perform an operation *op* by applying all the operations up until *op*

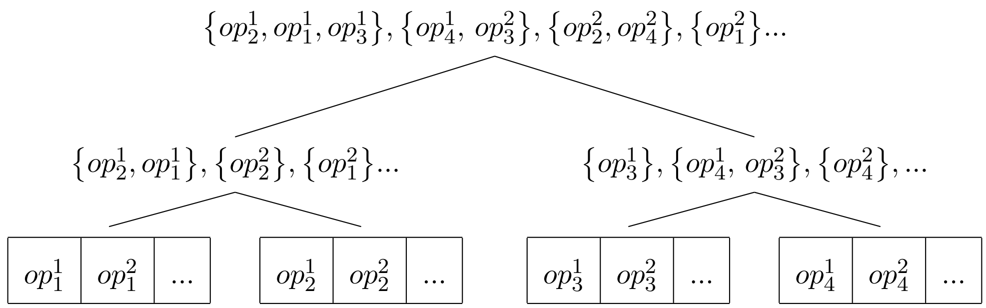

Figure 9: Leaves are for processes P_1 to P_4 from left to right. In each internal node one can arbitrarily linearize the sets of concurrent operations propagated together in a `Refresh`. For example op_4^1 and op_3^2 have propagated together in one `Propagate` step and they will be propagated up to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

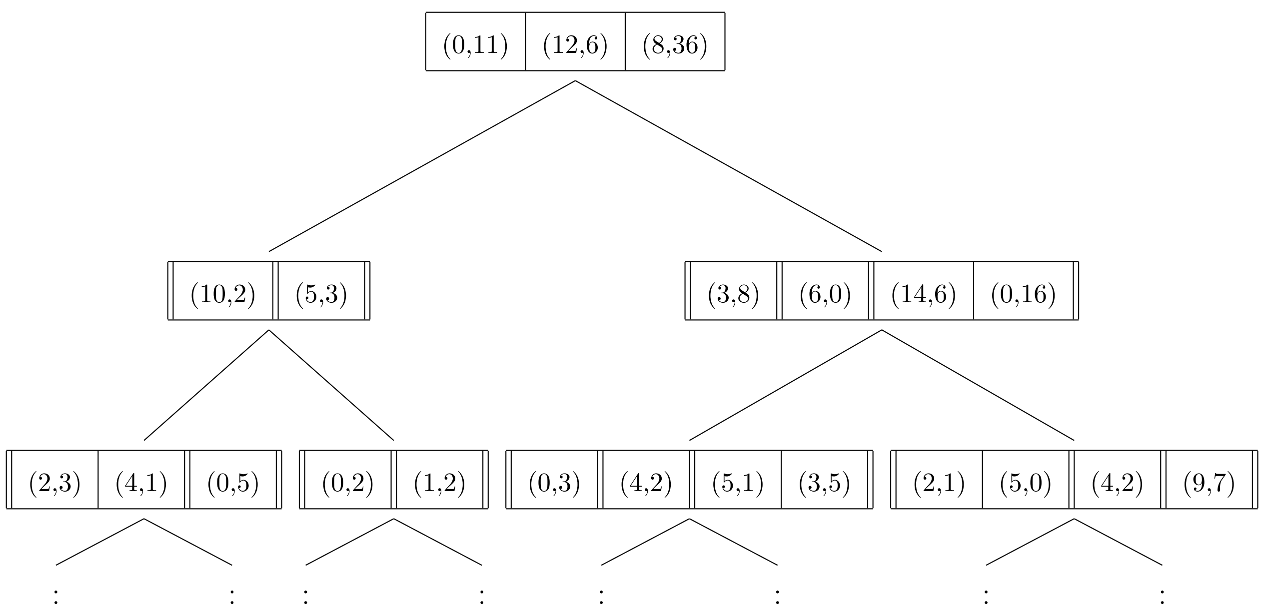

Figure 10: Using blocks to represent operations. Blocks between two lines || are propagated together to the parent. Each block consists of a pair (left, right) indicating the number of operations from the left and the right child, respectively. For example, (12,6) in the root contains (10,2) from the 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).

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

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

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

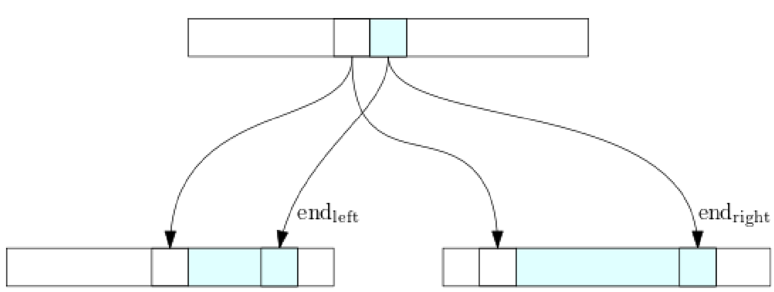

Figure 11: Each block stores the index of its last subblock in each child.

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

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

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

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

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

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

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

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

$$\max\Big(\text{size after } (b-1)\text{th block} + \#\texttt{Enqueue}\text{s in } b\text{th block} - \#\texttt{Dequeue}\text{s in } b\text{th block}, 0\Big).$$

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

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

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

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

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

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 `Block`s called `blocks` and the index `head` of the first empty entry in `blocks`.

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

$\texttt{end}_{\texttt{left}}$ and $\texttt{end}_{\texttt{right}}$ field store the index of the last subblock of a block in the left and the right child, respectively. The approximate index of the superblock of non-root blocks is stored in their `super` field. The `size` field in a block in the root node stores the size of the queue after the operations in the block have been performed.

We now describe the routines used in the implementation.

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

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

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

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

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

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

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

Since the index of the superblock of B is not known until B is propagated, R_n cannot set the `super` field of B while creating B. One approach for R_{par} is to set the `super` field of B after propagating B to par. This solution would not be efficient because there might be up to p

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

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

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

`GetEnqueue(`b, i`)` and `DoublingSearch(`e, end`)` We can describe an operation in a node in two ways: the rank of the operation among all the operations in the node or the index of the block containing the operation in the node and the rank of the operation within that block. If we know the block and rank within the block of an operation, we can find the subblock containing the operation and the operation's rank within that subblock in poly-log time. To find the response of a `Dequeue`, we know about the rank of the response `Enqueue` in the root (`e` in Line 22). We also know the `e`th `Enqueue` is in `root.blocks[1..end]`. `DoublingSearch` 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.`GetEnqueue`(b, i) returns the `element` of the ith enqueue in the bth block of n. The range of subblocks of a block is determined using the $\texttt{end}_{\texttt{left}}$ and $\texttt{end}_{\texttt{right}}$ fields of the block and its previous block. Then, the subblock is found using binary search on the $\texttt{sum}_{\texttt{enq}}$ field (Lines 99 and 103).

3.2 Pseudocode

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

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

Algorithm Tree Fields Description

♢ *Shared*

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

♢ *Local*

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

▶ *Node*

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

▶ *Block*

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

▶ *InternalBlock* `extends` *Block*

- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: indices of the last subblock of the block in the left and right child
- *int* $\texttt{sum}_{\texttt{enq-left}}$: #enqueues in $\texttt{left.blocks[1..end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-left}}$: #dequeues in $\texttt{left.blocks[1..end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: #enqueues in $\texttt{right.blocks[1..end}_{\texttt{right}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-right}}$: #dequeues in $\texttt{right.blocks[1..end}_{\texttt{right}}\texttt{]}$

▶ *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* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: # enqueue, dequeue operations in this block and its previous blocks in the leaf

▶ *RootBlock* `extends` *InternalBlock*

- *int* `size` : size of the queue after performing all operations in this block and its previous blocks in the root

Algorithm *Queue*

```
void Enqueue(Object e)                    ▷ Creates a block with element e and adds it to the tree.
    block newBlock= new(LeafBlock)
    newBlock.element= e
    newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq+1
    newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq
    leaf.Append(newBlock)
end Enqueue

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

▷ Returns the response to D_i(root, b), the ith Dequeue in root.blocks[b].
element FindResponse(int b, int i)
    if root.blocks[b-1].size + root.blocks[b].num_enq - i < 0 then     ▷ Check if the queue is empty.
        return null
    else                                  ▷ The response is E_e(root), the eth Enqueue in the root.
        e= i + (root.blocks[b-1].sum_enq-root.blocks[b-1].size)
        return root.GetEnqueue(root.DoublingSearch(e, b))
    end if
end FindResponse
```

Algorithm *Node*

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

Algorithm *Root*

```
⇝ Precondition: root.blocks[end].sum_enq ≥ e
▷ Returns <b,i> such that E_e(root) = E_i(root,b), i.e., the eth Enqueue in the root is the ith Enqueue within
▷ the bth block in the root.
<int, int> DoublingSearch(int e, int end)
    start= end-1
    while root.blocks[start].sum_enq>=e do
        start= max(start-(end-start), 0)
    end while
    b= root.BinarySearch(e, start, end)
    i= e- root.blocks[b-1].sum_enq
    return <b,i>
end DoublingSearch
```

Algorithm *Leaf*

```
void Append(block B)                                ▷ Only called by the owner of the leaf.
    blocks[head]= B
    head= head+1
    parent.Propagate()
end Append
```

Algorithm *Node*

```
▷ n.Propagate propagates operations in this.children up to this when it terminates.
void Propagate()
    if not Refresh() then
        Refresh()
    end if
    if this is not root then
        parent.Propagate()
    end if
end Propagate

▷ Creates a block containing new operations of this.children, and then tries to append it to this.
boolean Refresh()
    h= head
    for each dir in {left, right} do
        h_dir= dir.head
        if dir.blocks[h_dir]!=null then
            dir.Advance(h_dir)
        end if
    end for
    new= CreateBlock(h)
    if new.num==0 then return true
    end if
    result= blocks[h].CAS(null, new)
    this.Advance(h)
    return result
end Refresh
```

Algorithm *Node*

```
void Advance(int h)                         ▷ Sets blocks[h].super and increments head from h to h+1.
    h_p= parent.head
    blocks[h].super.CAS(null, h_p)
    head.CAS(h, h+1)
end Advance


Block CreateBlock(int i)                    ▷ Creates and returns the block to be installed in blocks[i].
    block new= new(InternalBlock)
    for each dir in {left, right} do
        index_prev= blocks[i-1].end_dir
        new.end_dir= dir.head-1             ▷ new contains dir.blocks[blocks[i-1].end_dir..dir.head-1].
        block_prev= dir.blocks[index_prev]
        block_last= dir.blocks[new.end_dir]
        new.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq - block_prev.sum_enq
        new.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq - block_prev.sum_deq
    end for
    if this is root then
        new.type= InternalBlock-->RootBlock
        new.size= max(root.blocks[i-1].size  + new.num_enq- new.num_deq, 0)
    end if
    return new
end CreateBlock
```

Algorithm *Node*

```
⇝ Precondition: blocks[b].num_enq ≥ i ≥ 1
element GetEnqueue(int b, int i)                    ▷ Returns the element of E_i(this, b).
    if this is leaf then
        return blocks[b].element
    else if i <= blocks[b].num_enq-left then        ▷ E_i(this, b) is in the left child of this node.
        subblockIndex= left.BinarySearch(i+blocks[b-1].sum_enq-left, blocks[b-1].end_left+1,
                        blocks[b].end_left)
        return left.GetEnqueue(subblockIndex, i)
    else
        i= i-blocks[b].num_enq-left
        subblockIndex= right.BinarySearch(i+blocks[b-1].sum_enq-right, blocks[b-1].end_right+1,
                        blocks[b].end_right)
        return right.GetEnqueue(subblockIndex, i)
    end if
end GetEnqueue

⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num_deq ≥ i.
<int, int> IndexDequeue(int b, int i)               ▷ Returns <x, y> if D_i(this, b) = D_y(root, x).
    if this is root then
        return <b, i>
    else
        dir= (parent.left==n ? left: right)
        superblockIndex= parent.blocks[blocks[b].super].sum_deq-dir > blocks[b].sum_deq ?
                          blocks[b].super: blocks[b].super+1
        if dir is left then
            i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-left
        else
            i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-right
            i+= parent.blocks[superblockIndex].num_deq-left
        end if
        return this.parent.IndexDequeue(superblockIndex, i)
    end if
end IndexDequeue
```

3.3 Example Execution

Here we want to demonstrate how four processes help one another to propagate their operations up to the root and then compute their responses.

Initially, each node in the tree contains one empty block. In Figure 12, processes $P1, P2, P4$ have appended the blocks containing ENQ_1, DEQ_2, DEQ_4. Line 47 appends a block to the process's leaf.

Then, as shown in Figure 13, $P2$ does a successful `Refresh` on the node $n5$ that propagates both the operations pf $P1$ and $P2$. $P4$ 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 $P2$'s successful `CAS` that propagated ENQ_1 and ENQ_2. So the block created by $P1$ is empty as there is nothing to propagate ($P1$'s Line 60 is after $P2$'s Line 70).

Since the block that $P1$ 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 propagates DEQ_4 to the `root` even before $P2$ puts its block into $n5$.

Then, as shown in Figure 15, $P4$ 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 $P3$ into the `root`. $P1$ fails to do `Refresh` on the `root` two times, but it is guaranteed that its operation is propagated before its second failed `CAS` on `root.blocks`.

All the fields of selected blocks from Figure 15 are shown in Figure 16. $P1$ and $P3$'s operations terminated after that they appended the `Enqueue`s to the root. $P2$ needs to compute the index of the block that its `Dequeue` is in. It computes the superblock of its leaf block, which is the second block in $n5$. Then, $P2$ computes the superblock of the block containing ENQ_1, DEQ_2. As we can see the `super` field of ENQ_1, DEQ_2 is 1 but the index of its superblock is 2. The queue size after the previous block is 0 so DEQ_2's response is ENQ_1's element. Then $P2$ tries to get ENQ_1's element. ENQ_1 is in the left subblock of block ENQ_1, ENQ_3, DEQ_2 because 1 is less than the $\texttt{sum}_{\texttt{enq-left}}$ field of the block containing ENQ_1, ENQ_3, DEQ_2. Similarly, $P2$ determines that ENQ_1 came from the left child of $n5$. P_2 therefore reaches the leaf $n1$ and returns ENQ_1's element. $P4$ similarly finds the block in the `root` that contains DEQ_4. Since it is the first operation in the block and the size of the queue after the previous (empty) block is 0, DEQ_4 returns `null`.

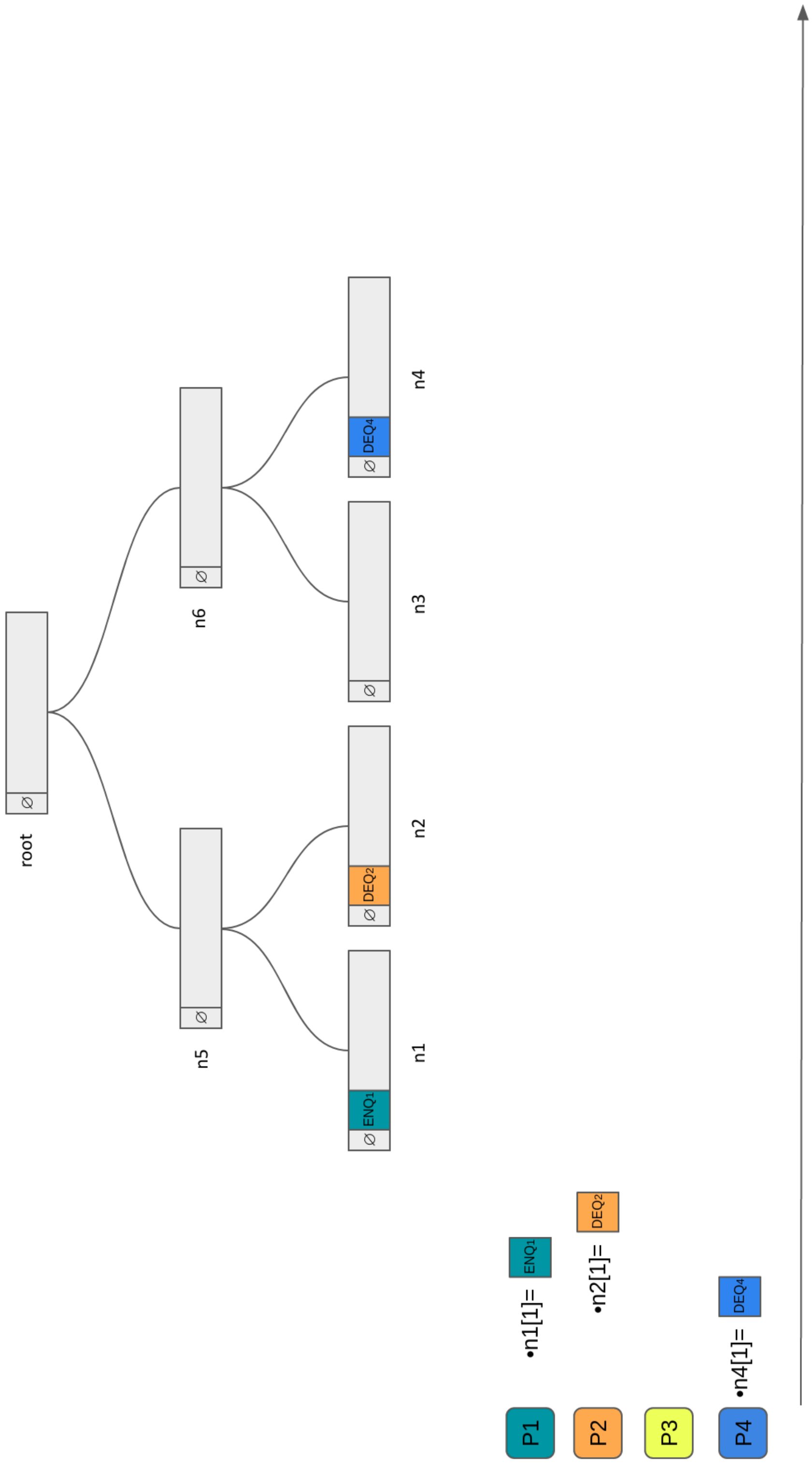

Figure 12: $P1, P2, P4$ append operations to their leaves.

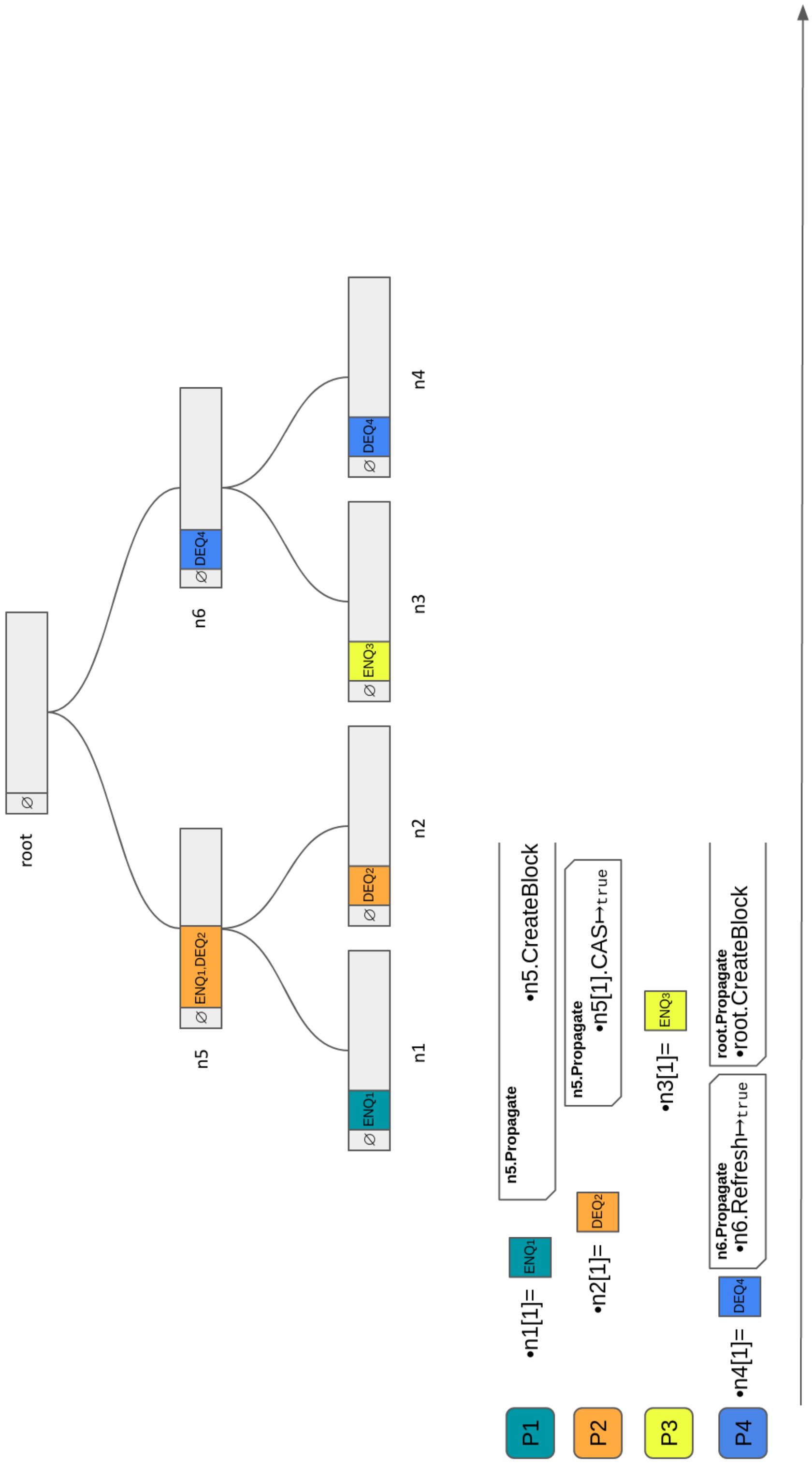

Figure 13: In the diagram, we use $n[i]$ as an abbreviation for $n.\texttt{blocks}[i]$. $P2$ helps $P1$. $P4$ is faster than $P3$.

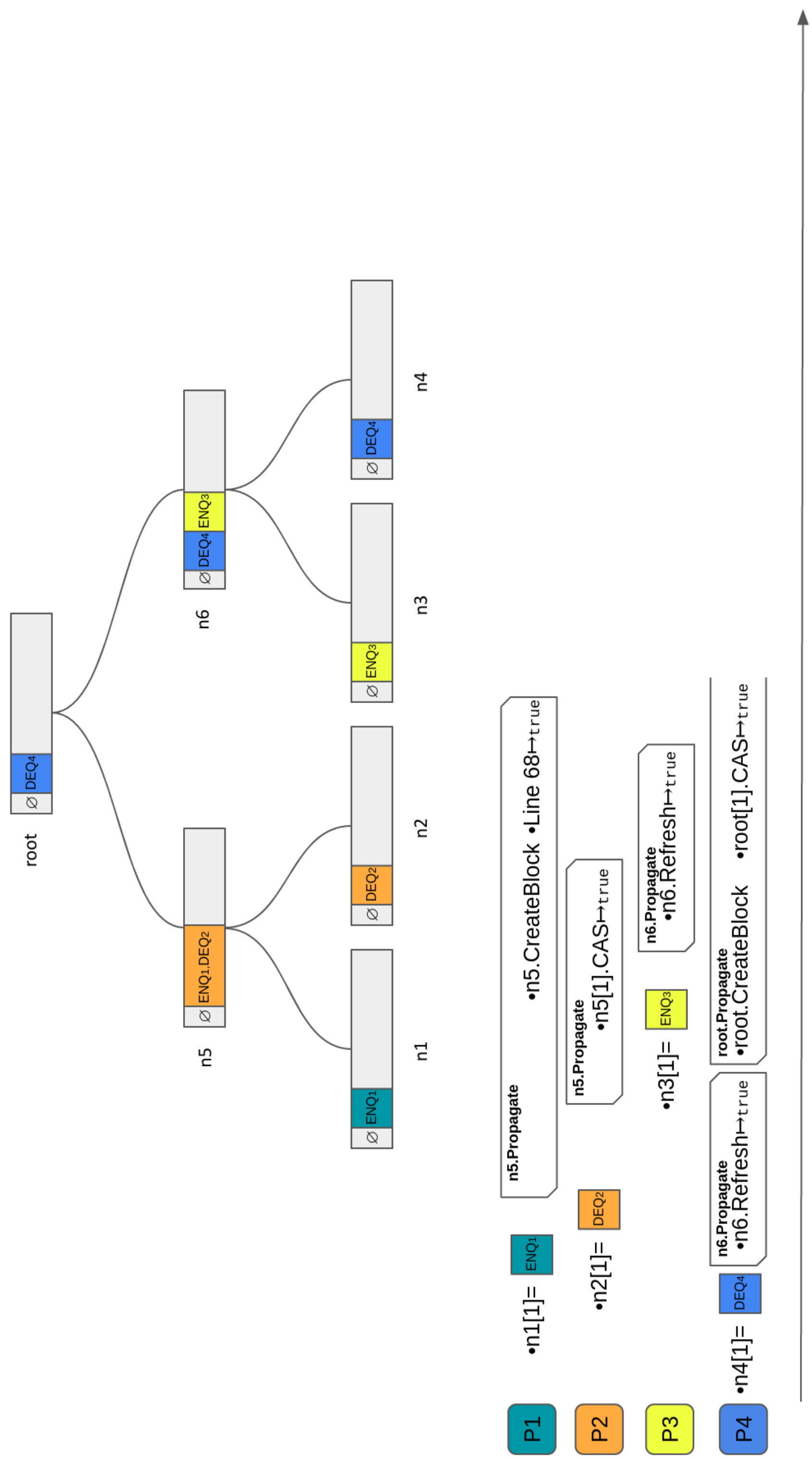

Figure 14: $P4$ puts its block into the `root` before $P2$ appends its block to $n5$.

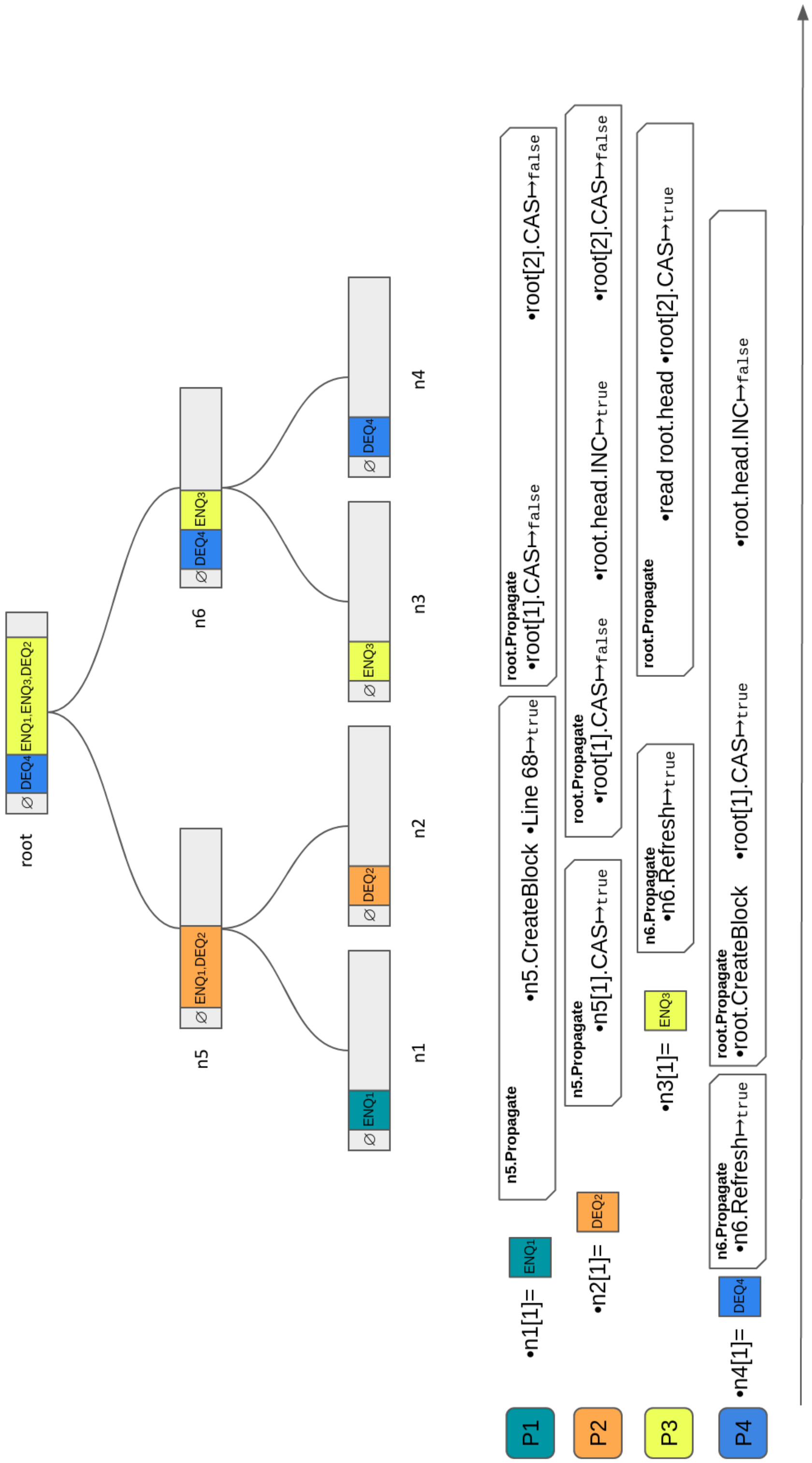

Figure 15: $P3$'s `CAS` on `root.blocks` succeeds. $P1$ and $P2$'s operations are done, each after two failed `CAS` attempts on `root.blocks`.

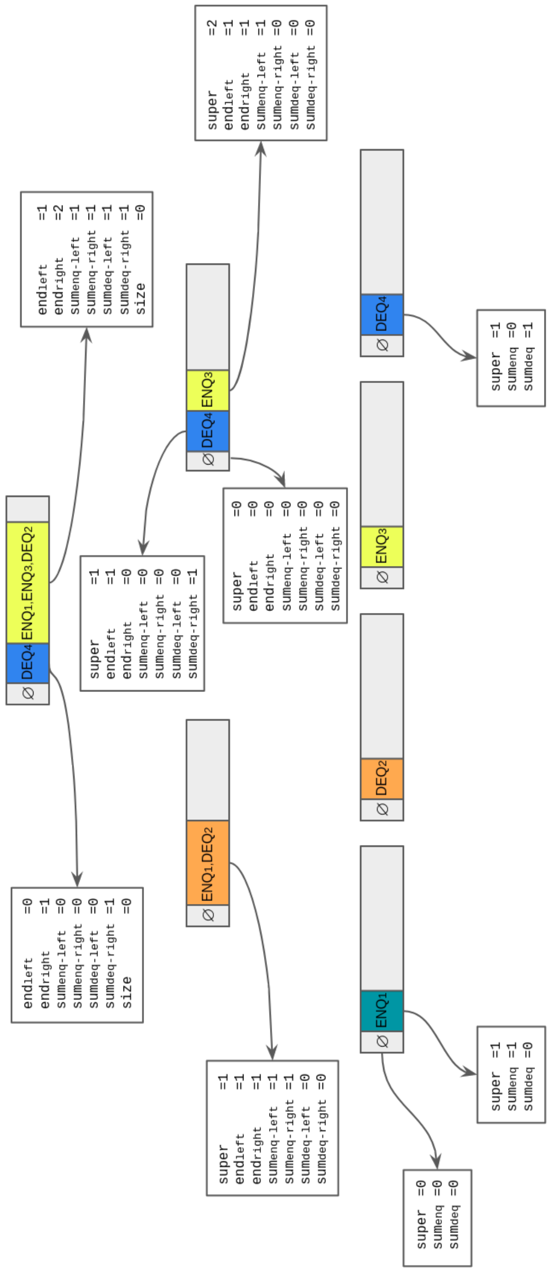

Figure 16: Figure 15 with completed fields.

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 linearization order, operations get the same results as in our queue. The proof is structured like this. First, we define and prove some facts about blocks and the node's `head` field. Then, we introduce the linearization ordering formally. Next, we prove double `Refresh` on a node is enough to propagate its children's new operations up to the node, which is used to prove (1). After this, we prove some claims about the size and operations of each block, which we use to prove the correctness of `DoublingSearch()`, `GetEnqueue()` and `IndexDequeue()`. Finally, we prove the correctness of the way we compute the response of a dequeue, which establishes (2).

4.1 Basic Properties

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

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.`blocks[`b`]` and B' be n.`blocks[`j`]`. We call i the *index* of block B. Block B is *before* block B' in node n if and only if $i < j$.

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.*`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 creates a non-empty block* `new` *(*`new.sum` $\neq 0$*). After R terminates, n*`.head` *is greater than the value read in Line 60 of R.*

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

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

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

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

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

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

We define the subblocks of a block recursively.

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

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

or in

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

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

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

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

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

Lemma 7. *The sets of subblocks of any two blocks in a node are disjoint.*

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

Definition 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. □

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

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

Lemma 11. *For any node n, if op is in n.*blocks[i] *then there is no $j \neq i$ such that op is in* n.blocks[j]*.*

Proof. We prove this claim by contradiction using Lemma 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, op cannot be in two leaf blocks. This is a contradiction. □

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_n^t is the set of established operations in node n at time t.

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

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

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

4.2 Ordering Operations

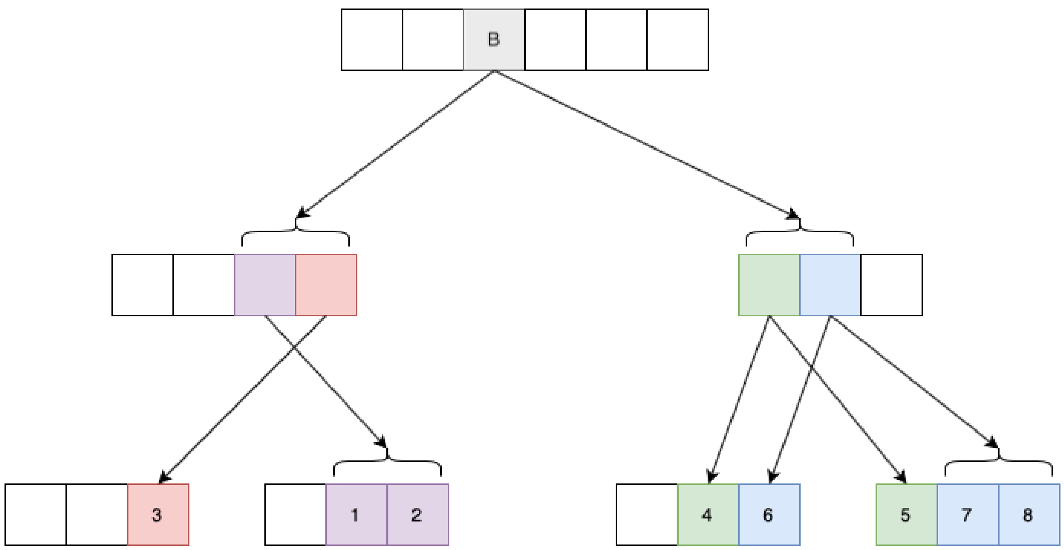

Figure 17: Order of operations in the block B. Operations in the leaves are ordered in the numerical order shown in the drawing.

Now we define the ordering of operations stored in each node. In the non-root nodes, we only need to order operations of a type among themselves (that is, we order the `Enqueue`s in the node and order the `Dequeue`s in the node separately). Processes are numbered from 1 to p, and leaves of the tree are assigned from left to right. We will show in Lemma 27 that there is at most one operation from each process in a given block.

Definition 14 (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

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

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

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

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

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

Definition 15 (Linearization)**.**

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

The following observation follows from the Definition of $\texttt{num}_{\texttt{x}}$ on page 22.

Observation 16. *For any node n and indices $i<j$ of* blocks *in n, we have*

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

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

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

Lemma 17. *Let B and B' be n.*`blocks`*$[b]$ and n.*`blocks`*$[b-1]$, respectively.*

If n is an internal node, then

(1) $B.\mathtt{num}_{\mathtt{enq\text{-}left}} = \Big|E(n.\mathtt{left}, B'.\mathtt{end}_{\mathtt{left}}+1)\cdots E(n.\mathtt{left}, B.\mathtt{end}_{\mathtt{left}})\Big|$.

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

And for every node n, we have

(3) $B.\mathtt{num}_{\mathtt{enq}} = \Big|E(n,b)\Big|$.

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

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

The first line follows from the Definition of $\mathtt{num}_{\mathtt{enq}}$. The second line is similar to the way $\mathtt{sum}_{\mathtt{enq\text{-}left}}$ is computed in the `CreateBlock` routine. Observation 16 implies the third line and the last line holds because of the induction hypothesis (3). (2) is similar to (1). Now we prove (3) starting from the definition of $E(n,b)$.

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

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

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

Corollary 18. *Let B be n.*`blocks[`b`]`*.*

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

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

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

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

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

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

4.3 Propagating Operations to the Root

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

Definition 19. Let t^{op} be the time *op* is invoked, ${}^{op}t$ be the time *op* terminates, t_l^{op} be the time immediately before running Line l of operation *op* and ${}^{op}_{l}t$ be the time immediately after running Line l of operation *op*. We sometimes suppress *op* and write t_l or ${}_lt$ if *op* is clear from the context. In the text, v_l is the value of variable v immediately after line l for the process we are talking about and v_t is the value of variable v at time t.

Definition 20 (Successful Refresh)**.** An instance of `Refresh` is *successful* if its `CAS` in Line 70 returns `true`.

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

Lemma 21. *If R is a successful instance of n*.`Refresh`*, then we have* $EST^{t^R}_{n.\mathtt{left}} \cup EST^{t^R}_{n.\mathtt{right}} \subseteq ops(n.\mathtt{blocks}_{70})$.

Proof. We show

$$EST^{t^R}_{n.\mathtt{left}} = ops(n.\mathtt{left.blocks[0..n.left.head}_{t^R} - 1\mathtt{]})$$
$$\subseteq ops(n.\mathtt{blocks}_{70}) = ops(n.\mathtt{blocks[0..}n.\mathtt{head}_{70}\mathtt{]}).$$

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

Corollary 22. *If R is a successful instance of n*.`Refresh` *that terminates, then we have*

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

Proof. The left-hand side is the same as Lemma 21, so it is sufficient to show when R terminates the established blocks in n are a superset of $n.\mathtt{blocks}_{70}$. Line 70 writes the block `new` in $n.\mathtt{blocks[}h\mathtt{]}$ where h is value of $n.\mathtt{head}$ read at Line 60. Because of Lemma 3 we are sure that $n.\mathtt{head} > h$ when R terminates. So the block `new` appended to n at Line 70 is established at ${}^Rt$. □

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

Proof. Let R_1 read i from $n.\mathtt{head}$ at Line 60. By Lemma 3, R_1 and R_2 cannot both read the same value i. By Observation 2, R_2 reads a larger value of $n.\mathtt{head}$ 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.`Refresh` that has done a `CAS` successfully into n.`blocks[`i+1`]` before R_2 tries to `CAS`. R_2' creates its block `new` after reading the value i+1 from n.`head` (Line 60) and R_1 reads the value i from n.`head`. By Observation 2 we have ${}^{R_1}t < t^{R_1}_{60} < t^{R2'}_{60}$ (see Figure 18). By Lemma 21 we have $EST^{n.\texttt{left}}_{{}^{R_2'}_{60}t} \cup EST^{n.\texttt{right}}_{{}^{R_2'}_{60}t} \subseteq ops(n.\texttt{blocks}_{t^{R_2'}_{70}})$. Also by Lemma 3 on R_2, the value of n.`head` is more than i+1 after R_2 terminates, so the block appended by R_2' into n.`blocks[`i`]` is established by the time R_2 terminates. To summarize, ${}^{R_1}t$ is before R_2''s read of n.`head` ($t^{R_2'}_{60}$), so we have $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{t^{R_2'}_{70}})$. R_2''s successful `CAS` ($t^{R_2'}_{70}$) is before R_2's termination (t^{R_2}), so by Lemma 3 n.`head` has been incremented when R_2 terminates and the block R_2' put into n is established by then. So we have $ops(n.\texttt{blocks}_{t^{R_2'}_{70}}) \subseteq EST^{{}^{R_2}t}_{n}$.

If R_2 reads some value greater than $i+1$ in Line 60 it means n.`head` has been incremented at least two times since ${}^{R_1}_{60}t$. By Invariant 4, when n.`head` is incremented from $i+1$ to $i+2$, n.`blocks[`$i+1$`]` is non-null. Let R_3 be the `Refresh` on n that has put the block in n.`blocks[`i+1`]`. R_3 read n.`head` $= i+1$ at Line 60 and has put its block in n.`blocks[`$i+1$`]` before R_2's read of n.`head` at Line 60. So we have $t^{R_1} <^{R_3}_{60} t <^{R_3}_{70} t < t^{R_2}_{60} <^{R_2} t$. From Observation 13 on the operations before and after R_3's `CAS` and Lemmas 21 and 3 on R_3 the claim holds. $\square$

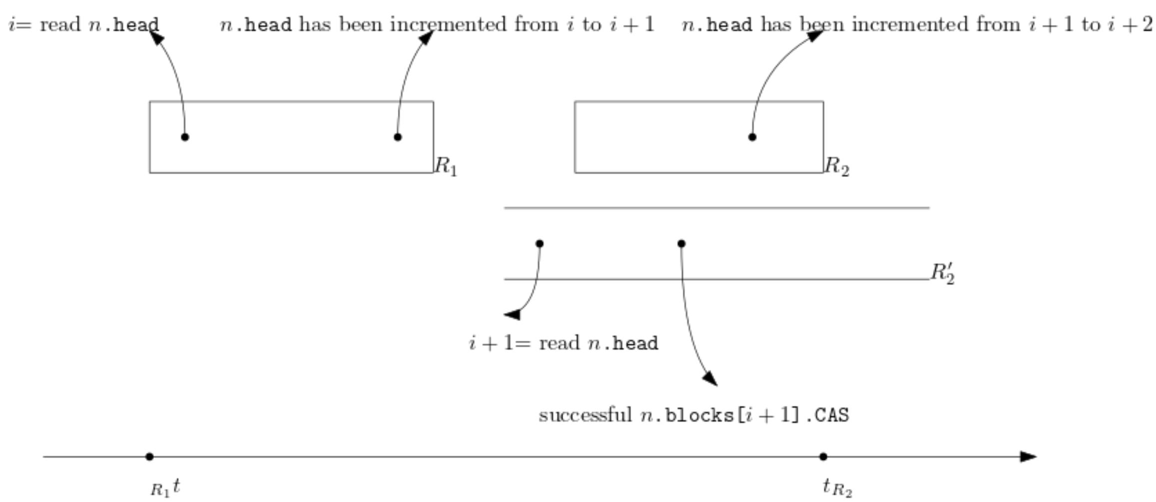

Figure 18: ${}_{R_1}t < t^{R_1}_{60} <$ incrementing n.`head` from i to $i+1 < t^{R_2'}_{60} < t^{R_2'}_{70} <$ incrementing n.`head` from $i+1$ to $i+2 < t_{R_2}$

Corollary 24. $EST^{52t}_{n.\texttt{left}} \cup EST^{52t}_{n.\texttt{right}} \subseteq EST^{t_{53}}_{n}$

Proof. If the first `Refresh` in line 52 returns `true`, then by Corollary 22 the claim holds. If the first `Refresh` failed and the second `Refresh` succeeded, the claim still holds by Corollary 22. Otherwise, both failed, and the claim is implied by Lemma 23. □

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

Corollary 25. *For* $A = l$`.Append(`B`)` *we have* $ops(b) \subseteq EST^{t^A}_{n}$ *for each node* n *in the path from* l *to* `root`*.*

Proof. A adds B to the assigned leaf of the process, establishes it at Line 48 and then calls `Propagate` on the parent of the leaf where it appended B. For every node n, n`.Propagate` appends B to n, establishes it in n by Corollary 24 and then calls n`.parent.Propagate` until n is `root`. □

Corollary 26. *After* l`.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. □

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

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process P in block B in node n. Without loss of generality op_1 is invoked earlier than op_2. Process P cannot invoke more than one operation concurrently, so op_1 has to be finished before op_2 begins. By Corollary 26, 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. □

Lemma 28. *Each block contains at most c operations, where c is the maximum number of concurrent operations at any time in the whole execution ($c \leq p$).*

Proof. There is a time that all the operations in a block are concurrent, because otherwise if there is an operation in a block that has ended before another operation in that block starts, then by Corollary 26 these two operations couldn't be in the same block. From the definition of c we know at any time in the execution there cannot be more than c concurrent operations, and from the previous lemma we know a process has at most one operation in a block, so there cannot be a block with more than c operations. □

Lemma 29. *Each block has at most c direct subblocks, where c is the maximum number of concurrent operations at any time in the whole execution ($c \leq p$).*

Proof. From Definition 10 we know the operations in a block are the union of the operations in the direct subblocks of the block. We can see that each block appended to an internal node contains at least one operation due to the test on Line 68. Also, blocks in the leaves contain only one `Enqueue` or `Dequeue` operation. By Lemma 28 each block in an internal node contains at most c operations and each one of its direct subblocks has at least one operation, so by pigeonhole principle the number of direct subblocks in a block is at most c. □

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

Lemma 30 (`DoublingSearch` correctness)**.** *If* $1 \leq e \leq$ `root.blocks[`end`].sum`$_{\texttt{enq}}$, *then* `Doubling-Search(`e`, `end`)` *returns* `<`b`, `i`>` *such that* $E_i(root, b) = E_e(root)$.

Proof. From Lines 86 and 87 we know the $\texttt{sum}_{\texttt{enq-left}}$, and $\texttt{sum}_{\texttt{enq-right}}$ fields of `blocks` in each node are sorted in non-decreasing order. Since $\texttt{sum}_{\texttt{enq}} = \texttt{sum}_{\texttt{enq-left}} + \texttt{sum}_{\texttt{enq-right}}$, the $\texttt{sum}_{\texttt{enq}}$ values of $\texttt{root.blocks[}0 \cdot \cdot end\texttt{]}$ are also non-decreasing. By Corollary 18 we know that the $\texttt{sum}_{\texttt{enq}}$ field in a block is the sum of the number of `Enqueue` operations in that block and the all blocks before that block in the node. Furthermore, since $\texttt{root.blocks[0].sum}_{\texttt{enq}} = 0$ and $\texttt{root.blocks[}end\texttt{].sum}_{\texttt{enq}} \geq e$, there is a b such that $\texttt{root.blocks[}b-1\texttt{].sum}_{\texttt{enq}} < e$ and $e \leq \texttt{root.blocks[}b\texttt{].sum}_{\texttt{enq}}$. Block $\texttt{root.blocks[}b\texttt{]}$ contains $E_i(root, b)$. Lines 38–41 doubles the search range in Line 40 and will eventually reach `start` such that $\texttt{root.blocks[start].sum}_{\texttt{enq}} \leq e \leq \texttt{root.blocks[end].sum}_{\texttt{enq}}$. Then, in Line 42, the binary search finds the b such that $\texttt{root.blocks[}b-1\texttt{].sum}_{\texttt{enq}} < e \leq \texttt{root.blocks[}b\texttt{].sum}_{\texttt{enq}}$. By Corollary 18, $\texttt{root.blocks[}b\texttt{]}$ is the block that contains $E_e(root)$. Finally, i is computed using the definition of $\texttt{sum}_{\texttt{enq}}$ and Corollary 18. □

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

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

For the induction step, we prove if $n\texttt{.dir.GetEnqueue(}b'\texttt{, }i\texttt{)}$ returns $E_i(n.\texttt{dir}, b')$ then $n\texttt{.GetEnqueue(}b\texttt{, }i\texttt{)}$ returns $E_i(n, b)$. From Definition 14 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 17, the $\texttt{num}_{\texttt{enq-left}}$ field in $n\texttt{.blocks[}b\texttt{]}$ is the number of `Enqueue` operations from the blocks' subblocks in the left child of n. So the ith `Enqueue` operation in $n\texttt{.blocks[}b\texttt{]}$ is propagated from the right child if and only if i is greater than $n\texttt{.blocks[}b\texttt{].num}_{\texttt{enq-left}}$. Line 98 decides whether the ith enqueue in the bth block of internal node n is in the left child or right child subblocks of $n\texttt{.blocks[}b\texttt{]}$. By Definitions 5 and 10, to find an operation in the subblocks of $n\texttt{.blocks[}b\texttt{]}$

we need to search in the range

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

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

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

Otherwise, the `Enqueue` we are looking for is in the right child. Because `Enqueue`s from the left subblocks are ordered before the ones from the right subblocks, there are $\texttt{n.blocks[b].num}_{\texttt{enq-left}}$ enqueues ahead of $E_i(n,b)$ from the left child. So we need to search for $i-n\texttt{.blocks[}b\texttt{].num}_{\texttt{enq-left}}+$ $n\texttt{.blocks[}b-1\texttt{].sum}_{\texttt{enq-right}}$ 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. *subblockIndex* is the index of the block in n`.dir` containing $E_i(n,b)$. Finally, n`.child.GetEnqueue(`*subblockIndex*`,` i`)` is invoked and it returns $E_i(n,b)$`.element` by the hypothesis of the induction. □

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.

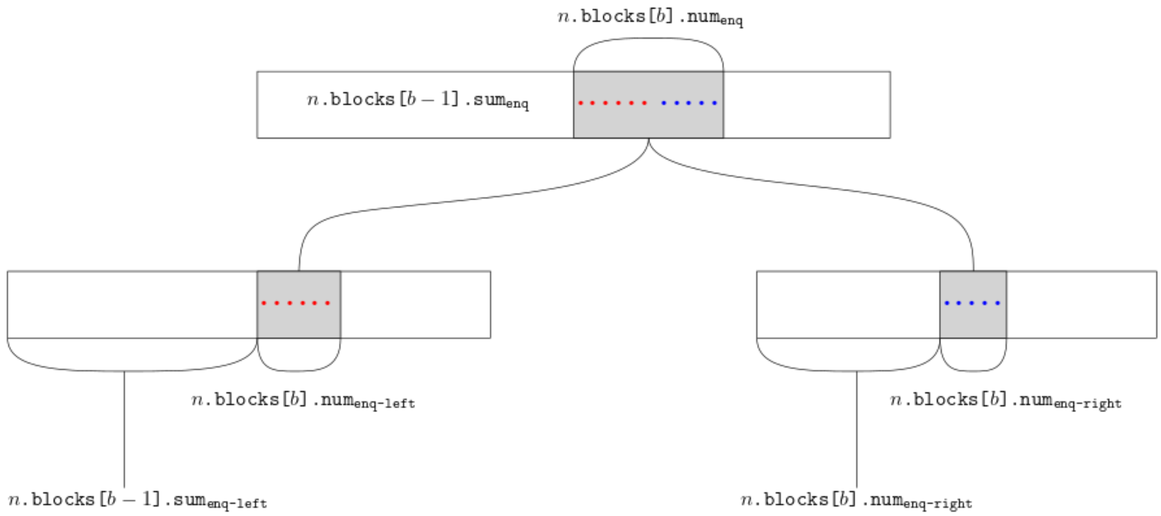

Figure 19: The number and order of the `Enqueue` operations propagated from the left and the right child to n.`blocks`[b]. Both n.`blocks`[b] and its subblocks are shown in grey. `Enqueue` operations from the left child (colored red), are ordered before the `Enqueue` operations from the right child (colored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

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

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

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

Proof. By Invariant 4 and Observation 33. □

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

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

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

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

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

Proof. Let R_{p1} and R_{p2} be the next two successful $n.\texttt{parent.Refresh}$es after R_n. To derive a contradiction assume B was neither propagated to $n.\texttt{parent}$ 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 $n.\texttt{head}$ 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 $n.\texttt{blocks[}b\texttt{]}$ so R_n reads the value b from $n.\texttt{head}$. Since R_{p2}'s $\texttt{CAS}$ into $n.\texttt{parent.blocks}$ is successful there should be a $\texttt{Refresh}$ instance R'_p on $n.\texttt{parent}$ that increments $n.\texttt{parent.head}$ (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.`blocks.CAS`.

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

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

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

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

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

Proof. Let R_n be the n.`Refresh` that put B in n.`blocks` and let R_{p1} be the first successful n.`parent.Refresh` after R_n and R_{p2} be the second next successful n.`parent.Refresh`. Before B can be propagated to n's `parent`, n.`head` must be greater than b, so by Observation 33 B.`super` is set. From Lemma 37 we know that B is propagated by the second next successful `Refresh`'s `CAS` on n.`parent.blocks`. To summarize, we have $n.\texttt{parent.head}_{{}^{R_{p2}}_{70}t} = n.\texttt{parent.head}_{{}^{R_{p1}}_{70}t} + 1$ and $n.\texttt{parent.head}_{{}^{R_{p1}}_{70}t} \leq n.\texttt{parent.head}_{{}^{R_n}_{70}t}$ from Definition 32 and Observation 2. The value that is set in B.`super` is read from n.`parent.head` after ${}^{R_n}_{70}t$. So B.`super` is equal to or one less than the index of the superblock of B. □

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

Lemma 39 (`IndexDequeue` correctness)**.** *If $1 \leq i \leq n.\texttt{blocks[}b\texttt{].num}_{\texttt{deq}}$ then n.`IndexDequeue(`b`,`i`)` returns $< x, y >$ such that $D_i(n, b) = D_y(\texttt{root}, x)$.*

Proof. We will prove this by induction on the distance of n from the `root`. The base case where n is `root` is trivial (see Line 109). For the non-root nodes n.`IndexDequeue(`b`, `i`)` computes *superblockIndex*, the index of the superblock of the bth block in n, in Line 112 by Corollary 38. After that, the position of $D_i(n, b)$ in $D(n.\texttt{parent}, superblockIndex)$ is computed in Lines 113–118. By Definition 14, `Dequeues` in a block are ordered based on the order of its subblocks from left to right. If $D_i(n, b)$ was propagated from the left child, the number of dequeues in the left subblocks of n`.parent.blocks[`*superblockIndex*`]` before n`.blocks[`b`]` is considered in Line 114 (see Figure 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`.parent` recursively, and it returns the correct response by the induction hypothesis. □

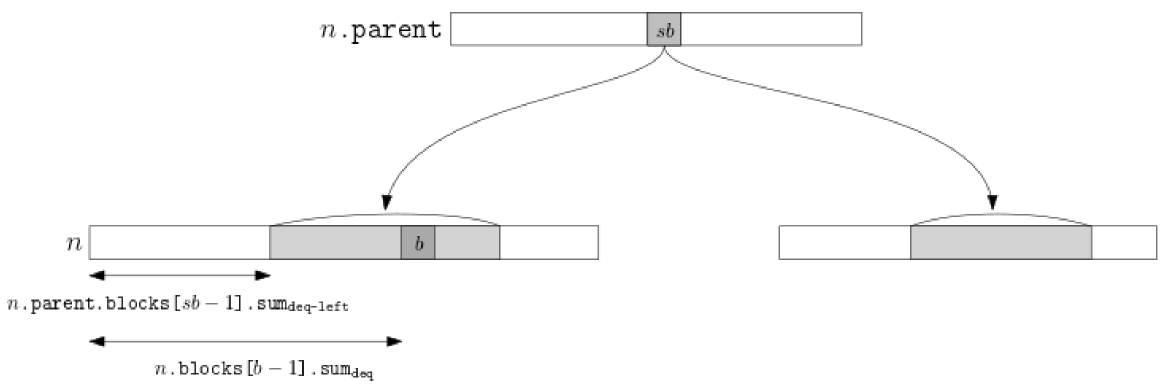

Figure 20: The number of `Dequeue` operations before $D_i(n, b)$ shown in the case where n is a left child. The index of the superblock is shown with sb.

4.6 Linearizability

We now prove the two properties needed for linearizability.

Lemma 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 in starts in the execution, then op_1 is before op_2 in the linearization. The first claim is directly reasoned from Corollary 26. For the latter, if op_1 terminates before

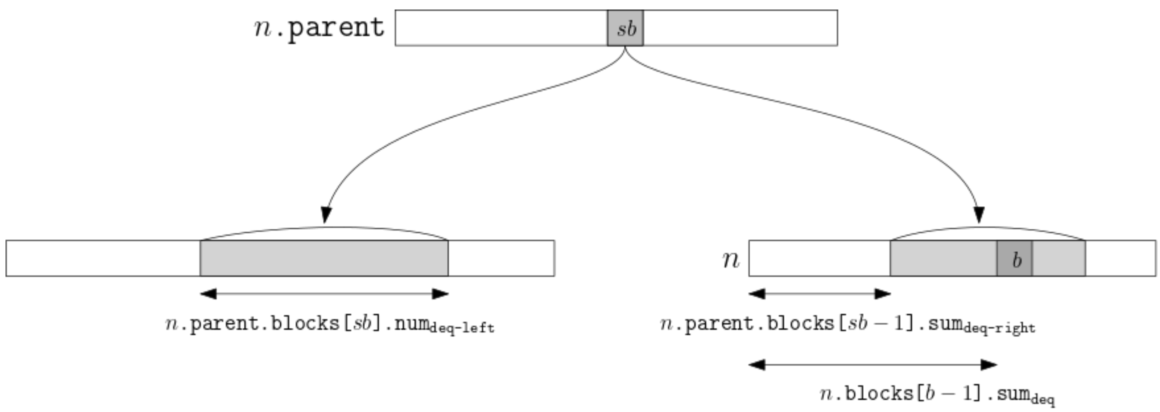

Figure 21: The number of `Dequeue` operations before $D_i(n, b)$ shown in the case where n is a right child. The index of the superblock is shown with *sb*.

op_2 starts, op_1`.Append` has terminated before op_2`.Append` started. From Corollary 25, 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 `Enqueue`s in a block before `Dequeue`s (see Definition 14).

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 `Dequeue`s 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 after the operations in* `root.blocks[`$0 \cdots b$`]` *are applied in the order of L.*

Proof. We prove the claim by induction on b. The base case when $b = 0$ is trivial since the queue is initially empty and `root.blocks[0]` contains an empty block with `size` field equal to 0. We are going to show the correctness when $b = i$ assuming correctness when $b = i - 1$. By Definition 14 `Enqueue` operations come before `Dequeue` operations in a block in L. By Lemma 17 $\texttt{num}_{\texttt{enq}}$ and $\texttt{num}_{\texttt{deq}}$ fields in a block show the number of `Enqueue` and `Dequeue` operations in it. If there are more than $\texttt{root.blocks[}i-1\texttt{].size} + \texttt{root.blocks[}i\texttt{].num}_{\texttt{enq}}$ dequeue operations in $\texttt{root.blocks[}i\texttt{]}$ then the queue would become empty after $\texttt{root.blocks[}i\texttt{]}$. Otherwise, the size of the queue after the bth block in the root is $\texttt{root.blocks[}b-1\texttt{].size} + \texttt{root.blocks[}b\texttt{].num}_{\texttt{enq}} - \texttt{root.blocks[}b\texttt{].num}_{\texttt{deq}}$. In both cases, this is the same as the assignment on Line 91. ☐

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* `Dequeue`*s in* $\texttt{root.blocks[}0\cdots b\texttt{]}$ *is* $\texttt{root.blocks[}b\texttt{].sum}_{\texttt{enq}} - \texttt{root.blocks[}b\texttt{].size}$.

Proof. There are $\texttt{root.blocks[}b\texttt{].sum}_{\texttt{enq}}$ `Enqueue` operations in $\texttt{root.blocks[}0\cdots b\texttt{]}$ by Corollary 18. The size of the queue after doing $\texttt{root.blocks[}0\cdots b\texttt{]}$ in the order of L is the number of *enqueues* in $\texttt{root.blocks[}0\cdots b\texttt{]}$ minus the number of *non-null* `Dequeue`*s* in $\texttt{root.blocks[}0\cdots b\texttt{]}$. By the correctness of the `size` field from Lemma 43 and $\texttt{sum}_{\texttt{enq}}$ field from Lemma 17, the number of *non-null* `Dequeue`*s* is $\texttt{root.blocks[}b\texttt{].sum}_{\texttt{enq}} - \texttt{root.blocks[}b\texttt{].size}$. ☐

Corollary 45. *If operations in the root are applied in the order of L, the number of non-null dequeues in* $\texttt{root.blocks[}b\texttt{]}$ *is* $\texttt{root.blocks[}b\texttt{].num}_{\texttt{enq}} - \texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}b-1\texttt{].size}$.

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

Proof. The claim follows immediately from Corollary 45 and Lemma 17. ☐

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

Proof. $D_i(root, b)$ is $D_{\texttt{root.blocks[}b-1\texttt{].sum}_{\texttt{deq}}+i}(root)$ by Definition 14 and Lemma 18. $D_i(root, b)$ returns `null` at Line 20 if $\texttt{root.blocks[}b-1\texttt{].size} + \texttt{root.blocks[}b\texttt{].num}_{\texttt{enq}} - i < 0$ and

$Resp(D_i(root, b)) =$ `null` in this case by Lemma 46. Otherwise, if $D_i(root, b)$ is the eth non-null `Dequeue` in L it should return the eth enqueued value. By Lemma 44 there are `root.blocks[`$b-1$`].sum`$_{\texttt{enq}}$ $-$ `root.blocks[`$b-1$`].size` non-null `Dequeue` operations in `root.blocks[`$0 \cdots b-1$`]`. The `Dequeues` in `root.blocks[`b`]` before $D_i(root, b)$ are non-null `Dequeues`. So $D_i(root, b)$ is the eth non-null `Dequeue` where $e = i +$ `root.blocks[`$b-1$`].sum`$_{\texttt{deq}}$ $-$ `root.blocks[`$b-1$`].size` (Line 22). See Figure 22.

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

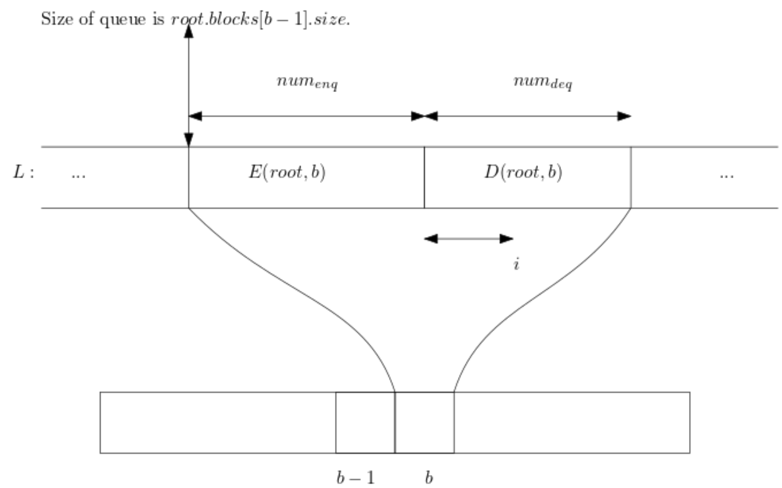

Figure 22: The position of $D_i(root, b)$.

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 the same as its response in the sequential execution of L. □

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

Proof. The theorem follows from Lemmas 40 and 48. □

Remark In fact our algorithm is strongly linearizable as defined in [10]. By Definition 14 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 `CAS`es, one in Line 70 and two from each `Advance` in Line 64 or 71. $\square$

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

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

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

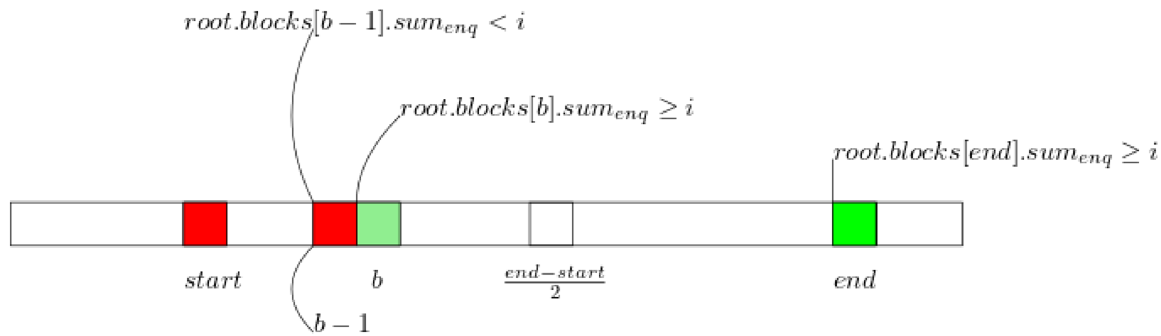

Figure 23: Distance relations between `start`, b, end.

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

Proof. `Enqueue` consists of creating a block and appending it to the tree. The first part takes constant time. To propagate the operation to the root the algorithm tries at most two `Refresh`es 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 $O(1)$ `CAS`es. Since the height of the tree is $\Theta(\log p)$, `Enqueue` takes $O(\log p)$ steps.

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

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

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

Observation 54. *If the maximum number of concurrent processes at any time in an execution is c, then the amortized worst-case step complexity is $O(\log p \log c + \log q)$ per operations. Furthermore, in a sequential, execution where $c = 1$, the step complexity of our algorithm is $\Theta(\log p + \log q)$ per operation.*

Proof. The analysis is similar to the two previous Lemmas, but by Lemma 29 each `BinarySearch` in each call of `GetEnqueue` takes $O(\log c)$ steps. ☐

Theorem 55. *The queue implementation is wait-free.*

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

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 compute 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 $\texttt{arr}[\lfloor \log\ \texttt{head} \rfloor]$ points to an array or not. If not, it creates a shared array of size $2^{\lfloor \log \texttt{head} \rfloor}$ and tries to `CAS` a pointer to the created array into $\texttt{arr}[\lfloor \log\ \texttt{head} \rfloor]$. Whether the `CAS` is successful or not, $\texttt{arr}[\lfloor \log\ \texttt{head} \rfloor]$ points to an array. When a process wishes to access the ith element it looks up $\texttt{arr}[\lfloor \log i \rfloor][i - 2^{\lfloor \log i \rfloor}]$, which takes $O(1)$ steps. The CAS Retry Problem does not happen here because if n elements are appended to the array, then only $O(p \times \log n)$ `CAS` steps have happened on the array `arr`. Furthermore, at most p arrays with size $2^{\lfloor \log i \rfloor}$ are allocated by processes while processes try to do the `CAS` on $\texttt{arr}[i]$. 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 $n >> p$. 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}.

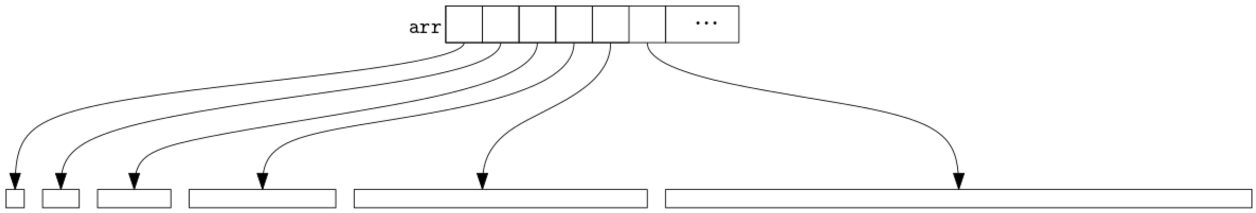

Figure 24: Array segments.

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 `block`s in the `root` whose operations are all terminated, i.e., all enqueues have been dequeued, and the responses of all dequeues have been computed. We call these blocks *finished blocks*. If we help the operations of all processes to compute their responses, then we can say if block B is finished, then all blocks before B are also finished. Knowing the most recent finished block in a node, we can reclaim the memory taken by finished blocks. We cannot use arrays (or vectors) to throw the garbage blocks away. We need a data structure that supports `tryAppend()`, `read(i)`, `write(i)` and `split(i)` operations in $O(\log n)$ time, where `split(i)` removes all the indices less than `i`. If each process tries to do the garbage collection once every p^2 operations on the queue, then the amortized complexity remains the same. We can use a concurrent implementation of a persistent red-black trees for this [26]. Bashari and Woelfel [3] used persistent red-black trees in a similar way.

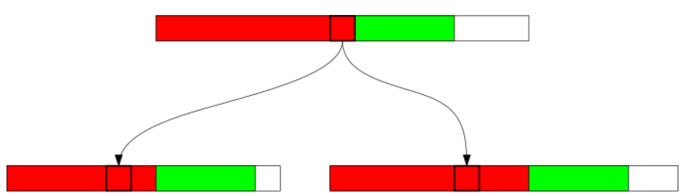

Figure 25: Finished blocks are shown with red color and unfinished blocks are shown with green color. All the subblocks of a finished block are also finished.

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 `DoublingSearch` 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 `append`s 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.

Attiya–Fouren Lower Bound As discussed 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 $\Omega(c)$ amortized complexity [2] when c the number of concurrent processes, is $O(\log\log p)$. Our amortized worst-case step complexity is $\Theta(\log^2 p + \log q)$. It is an open problem to reduce the gap between our algorithm and the Attiya–Fouren lower $\Omega(\log\log p)$ 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* (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* (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 *ACM Symposium on Principles of Distributed Computing* (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* (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* (2005), IEEE Computer Society, pp. 507–516.

[6] David, M. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference* (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* (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* (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* (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 *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures* (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 (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 (1990), 463–492.

[14] Hoffman, M., Shalev, O., and Shavit, N. The baskets queue. In *Principles of Distributed Systems, 11th International Conference* (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* (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 *Foundations of Software Technology and Theoretical Computer Science* (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* (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 enqueuers and dequeuers. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2011), C. Cascaval and P. Yew, Eds., ACM, pp. 223–234.

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

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

[21] 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* (1996), J. E. Burns and Y. Moses, Eds., ACM, pp. 267–275.

[22] 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* (2018), C. Scheideler and J. T. Fineman, Eds., ACM, pp. 99–109.

[23] Moir, M., Nussbaum, D., Shalev, O., and Shavit, N. Using elimination to implement scalable and lock-free FIFO queues. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2005), P. B. Gibbons and P. G. Spirakis, Eds., ACM, pp. 253–262.

[24] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013), A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, Eds., ACM, pp. 103–112.

[25] Nikolaev, R., and Ravindran, B. Wcq: A fast wait-free queue with bounded memory usage. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2022), SPAA '22, Association for Computing Machinery, p. 307–319.

[26] Okasaki, C. Functional data structures. In *Advanced Functional Programming, Second International School* (1996), J. Launchbury, E. Meijer, and T. Sheard, Eds., vol. 1129 of *Lecture Notes in Computer Science*, Springer, pp. 131–158.

[27] Shafiei, N. Non-blocking array-based algorithms for stacks and queues. In *Distributed Computing and Networking, 10th International Conference* (2009), V. K. Garg, R. Wattenhofer, and K. Kothapalli, Eds., vol. 5408 of *Lecture Notes in Computer Science*, Springer, pp. 55–66.

[28] 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* (2001), A. L. Rosenberg, Ed., ACM, pp. 134–143.