A Wait-free Queue with Polylogarithmic Step Complexity

Hossein Naderibeni

supervised by Eric Ruppert

October 17, 2022

Abstract

In this work, we are going to introduce a novel lock-free queue implementation. Linearizability and lock-freedom are standard requirements for designing shared data structures. All existing linearizable, lock-free queues in the literature have a common problem in their worst case called CAS Retry Problem. Our contribution is solving this problem while outperforming the previous algorithms.

Contents

1 Introduction **3**

2 Related Work **5**

2.1 List-based Queues 5

2.2 Universal Constructions 7

2.3 Attiya Fourier Lower Bound 8

3 Queue Implementation **9**

3.1 Details of the Implementation 14

3.2 Pseudocode 18

4 Proof of Correctness **23**

4.1 Basic Properties 23

4.2 Ordering Operations 27

4.3 Propagating Operations to the Root 30

4.4 Correctness of GetEnqueue 33

4.5 Correctness of IndexDequeue 36

4.6 Linearizability 39

5 Analysis **42**

6 Future Directions **44**

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. This is why distributed algorithms have become important. It is not hard to see why multiple processes cannot update sequential data structures designed for one process. For example, consider two processes trying to insert some values into 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 after that q does the same. In this run, p's update is overwritten. One solution is to use locks; whenever a process wants to do an update or query on a data structure, the process locks it, and others cannot use it until the lock is released. 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 one process holding the lock can make progress.

The question that may arise is, "What properties matter for a lock-free 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.

The standard safety condition is called *linearizability*, 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 is 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.

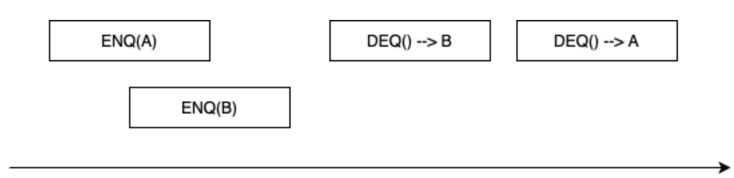

Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

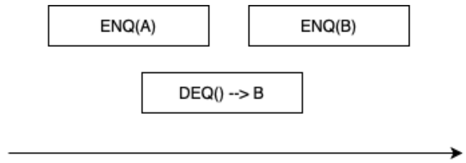

Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A` or nothing.

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.

A queue stores a sequence of elements and supports two operations, enqueue and dequeue. `Enqueue(e)` appends element `e` to the sequence stored. `Dequeue()` removes and returns the first element among in the sequence. If the queue is empty it returns `null`. In section 2 we talk about previous queues and their common problems. We also talk about polylogarithmic construction of shared objects.

Jayanti [11] proved an $\Omega(\log p)$ lower bound on the worst-case shared-access time complexity of p-process universal constructions. He also introduced [2] a construction that achieves $O(\log^2 p)$ shared accesses. Here, we first introduce a universal construction using $O(\log p)$ CAS operations [12]. In section 3 we introduce a polylogarithmic step wait-free universal construction. Our main ideas in of the universal construction also appear in our Queue Algorithm (3.2). The main short come of our universal construction is using big CAS objects. We use the universal construction as a stepping stone towards our queue algorithm, so we will not explain it in too much detail.

In section 4 we introduce a concurrent wait-free datastructure, to agree on the order of the operations invoked on some processes.

In section 5 we introduce our main work, the queue; prove its linearizability and wait-freeness.

2 Related Work

2.1 List-based Queues

In the following paragraphs, we look at previous lock-free queues. Michael and Scott [16] 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). Head points to the first node in the linked list that has not been dequeued, and Tail points to the last element in the queue. To insert a node into the linked list, they use atomic primitive operations like `LL/SC` or `CAS`. If p processes try to enqueue simultaneously, only one can succeed, and the others have to retry. This makes the amortized number of steps to be $\Omega(p)$ per enqueue. Similarly, dequeue can take $\Omega(p)$ steps.

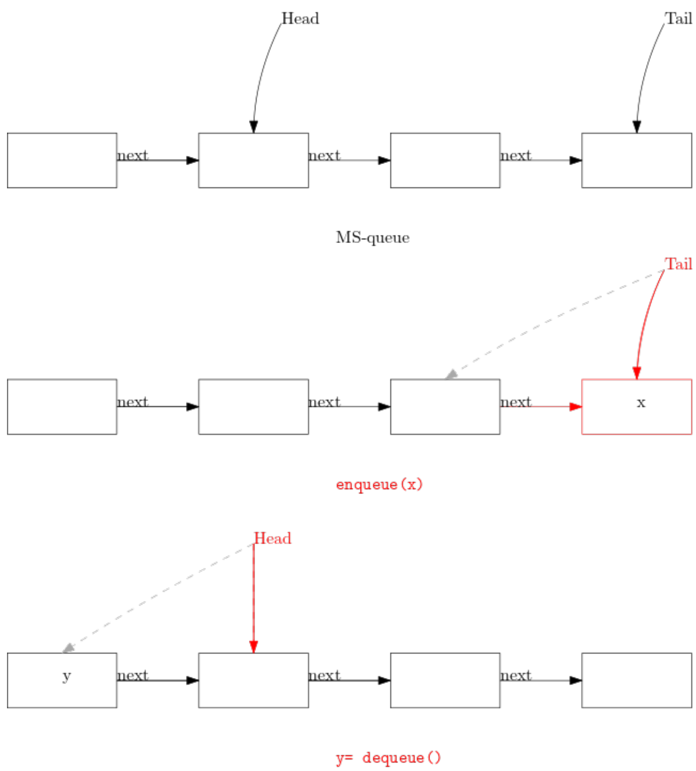

Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

Moir, Nussbaum, and Shalev [17] 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 [10] tried to make the MS-queue more parallel by introducing the Baskets Queue. Their idea is to allow more parallelism by treating the simultaneous enqueue operations as a basket. Each basket has a time interval in which all its nodes' enqueue operations overlap. Since the operations in a basket are concurrent, we can order them in any way. Enqueues in a basket try to find their order in the basket one by one by using `CAS` operations. However, like the previous algorithms, if there are still p concurrent enqueue operations in a basket, the amortized step complexity remains $\Omega(p)$ per operation.

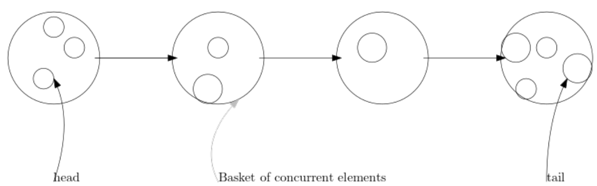

Figure 4: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do `CAS`. To order the operations in a basket, the mechanism in the algorithm for processes is to `CAS` again. The successful process will be the next one in the basket and so on.

Ladan-Mozes and Shavit [15] presented an Optimistic Approach to Lock-Free FIFO Queues. They use a doubly-linked list and do fewer `CAS` operations than MS-queue. But as before, the worst case is when there are p concurrent enqueues which have to be enqueued one by one. The amortized worst-case complexity is still $\Omega(p)$ `CAS`es.

Hendler et al. [8] proposed a new paradigm called flat combining. Their queue is linearizable but not lock-free. Their main idea is that with knowledge of all the history of operations, it might be possible to answer queries faster than doing them one by one. In our work we also maintain the whole history. They present experiments that show their algorithm performs well in some situations.

Gidenstam, Sundell, and Tsigas [6] introduced a new algorithm using a linked list of arrays. Global head and tail pointers point to arrays containing the first and last elements in the queue. Global pointers are up to date, but head and tail pointers may be behind in time. An enqueue or a dequeue searches in the head array or tail array to find the first unmarked element or last written element (see Figure 5). Their data structure is lock-free. Still, if the head array is empty and p processes try to enqueue simultaneously, the step complexity remains $\Omega(p)$.

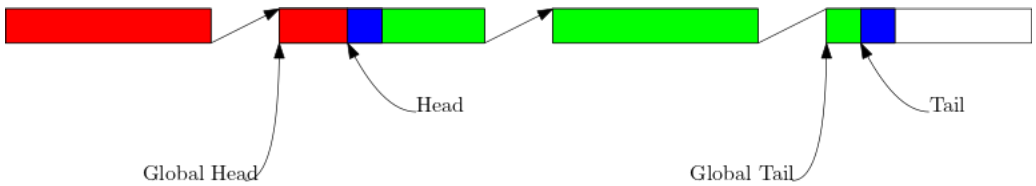

Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

Kogan and Petrank [14] introduced wait-free queues based on the MS-queue and use Herlihy's helping technique to achieve wait-freedom. Their step complexity is $\Omega(p)$ because of the helping mechanism.

In the worst-case step complexity of all the list-based queues discussed above, there is a p term that comes from the case all p processes try to do an enqueue simultaneously. Morrison and Afek call this the *CAS retry problem* [18]. It is not limited to list-based queues and array-based queues share the CAS retry problem as well [20, 19, 3] . We are focusing on seeing if we can implement a queue in sublinear steps in terms of p or not.

2.2 Universal Constructions

Herlihy discussed the possibility of implementing shared objects from other objects [9]. A *universal construction* is an algorithm that can implement a shared version of any given sequential object. 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 [11]. He also introduced a construction that achieves $O(\log^2 p)$ shared accesses [2]. His universal construction can be used to create any data structure, but its implementation is not practical because of using unreasonably large-sized `CAS` operations.

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 [4]. Their idea has similarities to Jayanti's construction, and they represent the value of the Fetch&Inc using the history of successful operations.

2.3 Attiya Fourier Lower Bound

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.

Jayanti and Petrovic introduced a wait-free poly-logarithmic multi-enqueuer single-dequeuer queue [12]. We use their idea of having a tournament tree among processes to agree on the linearization of operations to design a a polylogarithmic multi-enqueuer multi-dequeuer queue. Unlike their work, our algorithm does not put a limit on the number of concurrent dequeuers.

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

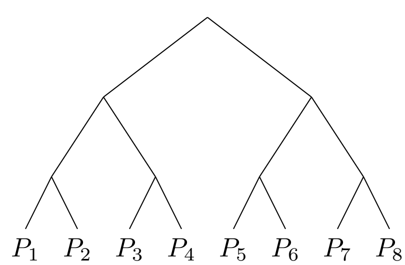

Figure 6: Each of the processes $P_1, P_2, ..., P_p$ has a leaf and in each node there is an ordering of operations stored. Each process tries to propagate its operations up to the root, which stores a total ordering of all operations.

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

n's children before the first `Refresh` are guaranteed to be in n after the second failed `Refresh`. We sketch the argument here.

(a) Before the `Refresh`. (b) New operations are appended.

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.

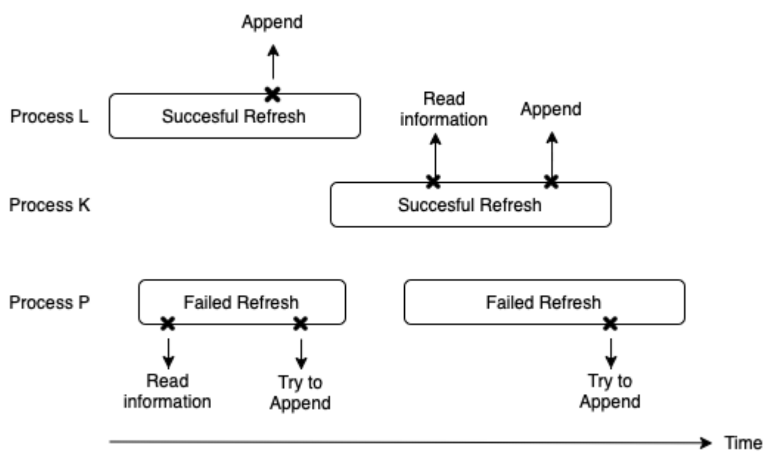

Figure 8: Time relations between the concurrent successful `Refresh`es and the two consecutive `Refresh`es.

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 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 store the sequence of operations in each node explicitly 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 `CAS` 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 children into a new block and puts it in the parent's blocks. We call the aggregated blocks *subblocks* of the new block and the new block the *superblock* of them. In each `Refresh` there is at most one operation from each process trying to be propagated, because one operation 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 on the operations.

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

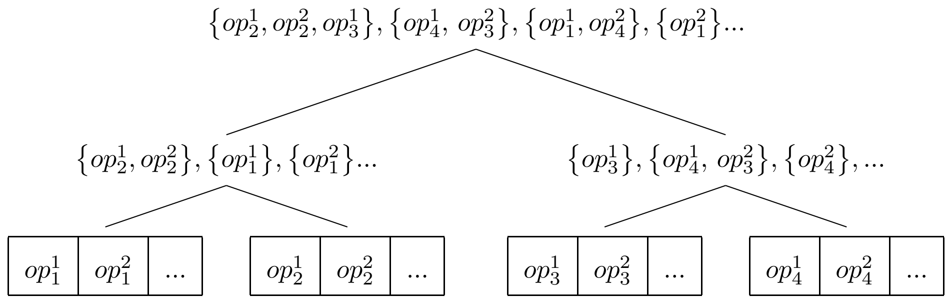

Figure 9: Leaves are for processes P_1 to P_4 from left to right. In each internal node one can arbitrarily linearize the sets of concurrent operations propagated together in a `Refresh`. For example op_4^1 and op_3^2 have propagated together in one `Propagate` step and they will be propagated up to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

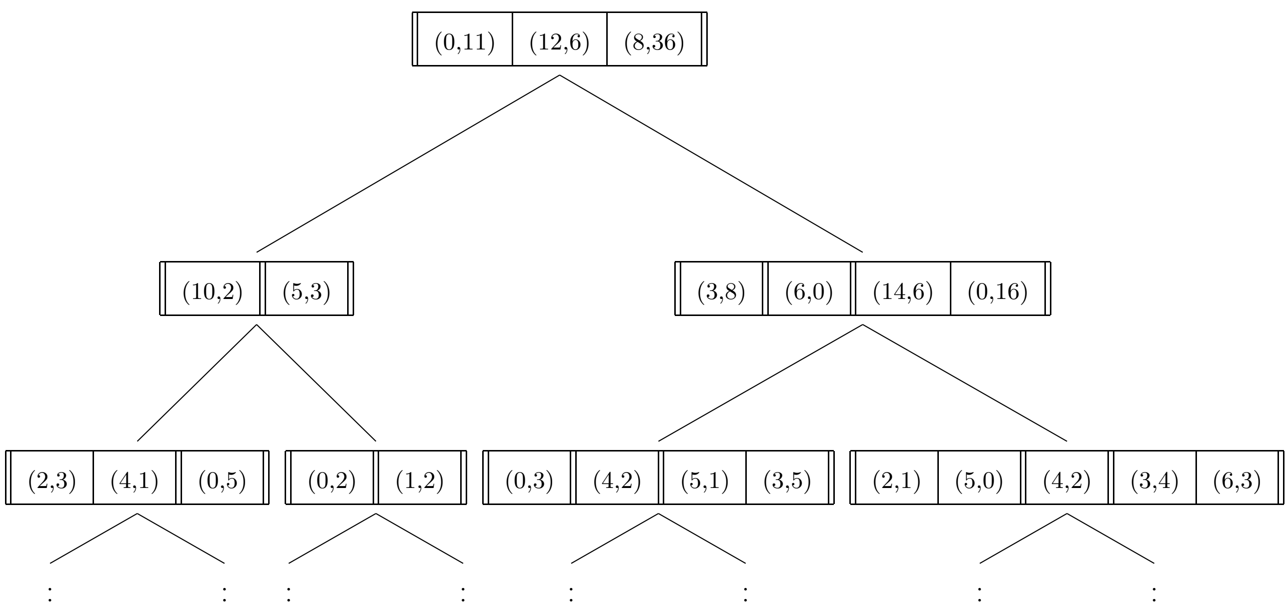

Figure 10: Using blocks to represent operations. Blocks between two lines || are propagated together to the parent. Each block consists of a pair (left, right) indicating the number of operations from the left and the right child, respectively. For example, (12,6) in the root contains (10,2) from the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

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 pointers to the last left and right subblock of it (see Figure 11). We know a block size is at most p, so binary search takes at most $\mathrm{O}(\log p)$ time, since the pointers of a block and its previous block reduce the search range size to $\mathrm{O}(p)$.

To compute the rank in the root of an operation in the leaf, we need to find the superblock of the block that 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.

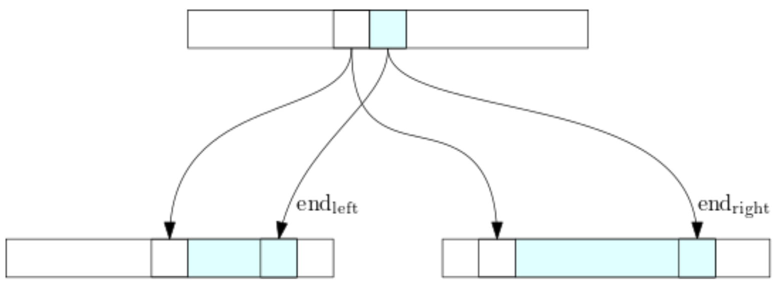

Figure 11: Each block stores the index of its last subblock in each child.

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 size of the queue after the previous operation is 0. We refer to such a dequeue as a *null dequeue*. If the 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 up to block B. 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 rank. 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.

The `Dequeue` operations return `null, 5, 2, 1, 3, 4, null` respectively. Now, we claimed that by knowing

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

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 happens, 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`
#`Enq`s	0	3	1	1
#`Deq`s	1	1	1	4
Size at end	0	2	2	0

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

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

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

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

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

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

3.1 Details of the Implementation

Pseudocode for the queue implementation is given in Section 3.2. It uses the following two types of objects.

`Node` In each `Node` we store pointers to its parent and children, 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 simply 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 block prior to B ended. 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 last subblock of a block in the left and the right child, respectively. The approximate index of the superblock of non-root blocks is stored in their `super` field. The `size` field in a block in the root node stores the size of the queue after the operations in the block have been performed.

Enqueue(e) An `Enqueue` operation does not return a response, so it is sufficient to just 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`.

FindResponse(b, i) To compute the response of the ith `Dequeue` in the bth block of the root Line 219 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 222 computes the rank e of the `Enqueue` whose argument is 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 `DSearch` and after that `GetEnqueue` finds its `element`.

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

Propagate() `Propagate` on node n uses the double refresh idea described in Section 3 and invokes two `Refresh`es on n in Lines 302 and 303. Then, it invokes `Propagate` on n`.parent` recursively until it reaches the root.

Refresh() 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 310. The new block created by Refresh will be inserted into n.blocks[h]. Lines 311–316 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 recent appended blocks. The reason behind this helping is explained later when we discuss IndexDequeue. After helping to Advance the children, a new block called new is created in Line 317. 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 320 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 321.

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 333 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 sum of the 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.

GetEnqueue(b, i) and DSearch(e, end) We can describe an operation in a node in two ways: the rank of the operation among all the operations in the node or the index of the block containing the operation in the node and the rank of the operation within that block. If we know the block and rank within the block of an operation we can find the subblock containing the operation and the operation's rank within that subblock in poly-log time. To find the response of a Dequeue, we know about the rank of the response Enqueue in the root (e in Line 222). We also know the eth Enqueue is in root.blocks[1..end]. DSearch uses doubling to find the range that contains the answer block (Lines 802–805) and then tries to find the required indices with a binary search (Line 806). A call to n.GetEnqueue(b, i) returns the element of the ith enqueue in bth the 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

405 and 409).

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 a 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 in its created block in Line 317. Let j be the index of the block that R_{par} put in par.blocks.

Since the index of the superblock of B is not known until B is propagated, R_n cannot set the super field of B while creating it. One approach for R_{par} is to set the super field of B after propagating B to par. This solution would not be efficient because there might be up to p subblocks in the block R_{par} propagated needing updates 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 Refreshes 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 326), R_n might go to sleep at some time after installing B and before setting B.super. In this case, the next Refreshes on n and par help fill in the value of B.super.

Block B is appended to n.blocks[b] on Line 320. After appending B, B.super is set on Line 326 of a call to Advance from n.Refresh by the same process or another process or by Line 314 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.

3.2 Pseudocode

Algorithm Tree Fields Description

◇ *Shared*

- A binary tree of `Node`s 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

▶ *RootBlock* extends *InternalBlock*

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

▶ *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 the prefix for the block

Abbreviations used in the code and the proof of correctness.

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

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

Object Dequeue()          ▷ Creates a block withnull value element, appends it to the tree and returns its response.
    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

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

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 block b of the root.
<int, int> DSearch(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 DSearch
```

Algorithm Leaf

```
void Append(block B)                    ▷ Only called by the owner of the leaf.
    blocks[head]= B
    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 containig 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

void Advance(int h)
    h_p= parent.head
    blocks[h].super.CAS(null, h_p
    head.CAS(h, h+1)
end Advance

    ⇝ Precondition: blocks[start..end] contains a block with sum_enq
    greater than or equal to i
    ▷ Does a binary search for the value i of sum_enq field. Returns the
    index of the leftmost block in blocks[start..end] whose sum_enq
    is ≥ i.
int BinarySearch( int i, int start, int end)
    return min{j:  blocks[j].sum_enq≥i}
end BinarySearch

    ▷ Creates and returns the block to be installed in blocks[i].
    Created block includes left.blocks[index_prev+1..index_last] and
    right.blocks[index_prev+1..index_last].
Block CreateBlock(int i)
    block new= new(block)
    for each dir in {left, right} do
        index_prev= blocks[i-1].end_dir
        new.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.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.
        subBlock= left.BinarySearch( i+blocks[b-1].sum_enq-left, blocks[b-1].end_left+1, blocks[b].end_left)
        return left.GetEnqueue(subBlock, i)
    else
        i= i-blocks[b].num_enq-left
        subBlock= right.BinarySearch( i+blocks[b-1].sum_enq-right, blocks[b-1].end_right+1, blocks[b].end_right)
        return right.GetEnqueue(subBlock, 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)
        sb= (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[sb-1].sum_deq-left
        else
            i+= blocks[b-1].sum_deq-parent.blocks[sb-1].sum_deq-right
            i+= parent.blocks[sb].num_deq-left
        end if
        return this.parent.IndexDequeue(sb, i)
    end if
end IndexDequeue
```

4 Proof of Correctness

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

4.1 Basic Properties

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

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

Definition 1 (Ordering of a block in a node)**.** Let b be n.`blocks[`i`]` 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 $i < j$. We define *the prefix for* block b in node n to be the blocks in n.`blocks[0..`i`]`.

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

Observation 2. *For each node n, n.*`head` *is non-decreasing over time.*

Proof. The claim follows trivially from the code since `head` is only changed by incrementing in Line 327 of `Advance`. ☐

Lemma 3. *Let R be an instance of* `Refresh` *on a node n. After R terminates, n.*`head` *is greater than the value read in line 310 of R.*

Proof. If the `CAS` in Line 327 is successful then the claim holds. Otherwise n.`head` has changed from the value that was read in Line 310. 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 320, which updates n.blocks[h] where h is the value read from n.head in Line 310. Since the CAS in Line 320 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 327 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 314 and 321. For Line 314, n.blocks[h] $\neq$ null because of the if condition in Line 313. For Line 321, Line 320 was finished before doing 321. Whether Line 320 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 is *propagated* to node n if b is in *n.blocks* or is a subblock of a block in n.blocks.

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

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

Proof. Consider the block b written into `n.blocks[`i`]` by `CAS` at Line 320. Block b is created by the `CreateBlock(`i`)` called at Line 317. Prior to this call to `CreateBlock(`i`)`, `n.head` $= i$ at Line 310, so `n.blocks[`$i-1$`]` is already a non-null value b' by Invariant 4. Thus, the `CreateBlock(`$i-1$`)` that created b' terminated before the `CreateBlock(`i`)` that creates b is invoked. The value written into $b.\texttt{end}_{\texttt{left}}$ at Line 336 of `CreateBlock(`i`)` was one less than the value read at Line 336 of `CreateBlock(`i`)`. Similarly, the value in `n.blocks[`$i-1$`]`$.\texttt{end}_{\texttt{left}}$ was one less than the value read from `n.left.head` during the call to `CreateBlock(`$i-1$`)`. 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. *Subblocks of any two blocks in node n do not overlap.*

Proof. We are going to prove the lemma by contradiction. Consider the lowest `node` n in the tree that violates the claim. Then subblocks of n`.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`]`$.\texttt{end}_{\texttt{left}} \leq n$`.blocks[`$j-1$`]`$.\texttt{end}_{\texttt{left}}$, so the ranges $[n.\texttt{blocks[}i-1\texttt{]}.\texttt{end}_{\texttt{left}}{+}1 \cdots n.\texttt{blocks[}i\texttt{]}.\texttt{end}_{\texttt{left}}]$ and $[n.\texttt{blocks[}j-1\texttt{]}.\texttt{end}_{\texttt{left}}{+}1 \cdots n.\texttt{blocks[}j\texttt{]}.\texttt{end}_{\texttt{left}}]$ cannot overlap. Therefore, direct subblocks of n`.blocks[`i`]` and n`.blocks[`j`]` cannot overlap. ☐

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 B by $ops(B)$. We also say b contains op if $op \in ops(b)$.

Operations are distinct `Enqueue`s and `Dequeue`s invoked by processes. The next lemma proves that each operation appears at most once in the blocks of a node.

Lemma 11. *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 using Lemma 7. Assume *op* is in the subblocks of both $n.blocks[i]$ and $n.blocks[j]$. From Corollary 7 we know that the subblocks of these blocks are different, so there are two leaf blocks containing *op*. Since each process puts each operation in only one block of its leaf then *op* cannot be in two leaf blocks. This is a contradiction. □

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

Now we want to say the 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 blocks of a node after the `CAS` contains the the set of blocks before the `CAS`. □

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

Proof. From Observations 2, 13. □

4.2 Ordering Operations

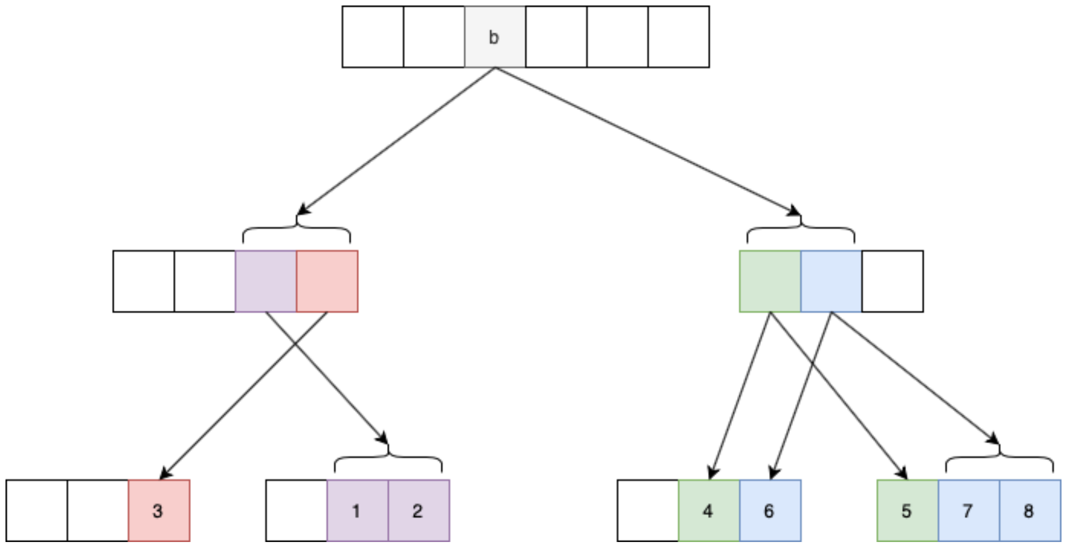

Figure 12: Order of operations in b. Operations in the leaves are ordered with numerical order shown in the drawing.

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

Definition 15 (Ordering of operations inside the nodes)**.**

- $E(n,b)$ is the sequence of enqueue operations in $ops(n\texttt{.blocks[}b\texttt{]})$ defined recursively as follows. $E(leaf,b)$ is the single enqueue operation in $ops(leaf\texttt{.blocks[}b\texttt{]})$ or an empty sequence if $leaf\texttt{.blocks[}b\texttt{]}$ 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 $ops(n\texttt{.blocks[}b\texttt{]})$ defined recursively as follows. $D(leaf,b)$ is the single dequeue operation in $ops(leaf\texttt{.blocks[}b\texttt{]})$ or an empty sequence if $leaf\texttt{.blocks[}b\texttt{]}$ 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 that operations appear in the blocks in the root.

Definition 16 (Linearization)**.**

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

Observation 17. *For any node n and indices $i<j$ of* `blocks` *in in, 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 `x` *in* $\{\texttt{enq}, \texttt{deq}, \texttt{enq-left}, \texttt{enq-right}, \texttt{deq-left}, \texttt{deq-right}\}$.

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

Lemma 18. *Let B, B' be $n\texttt{.blocks[}b\texttt{]}$, $n\texttt{.blocks[}b-1\texttt{]}$ respectively.*

(1) If n is an internal node $B\texttt{.num}_{\texttt{enq-left}} = \Big|E(n\texttt{.left}, B'\texttt{.end}_{\texttt{left}}+1)\cdots E(n\texttt{.left}, B\texttt{.end}_{\texttt{left}})\Big|$.

(2) If n is an internal node $B\texttt{.num}_{\texttt{enq-right}} = \Big|E(n\texttt{.right}, B'\texttt{.end}_{\texttt{right}}+1)\cdots E(n\texttt{.right}, B\texttt{.end}_{\texttt{right}})\Big|$.

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

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

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

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

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

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

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

Corollary 19. *Let B be $n\texttt{.blocks[}b\texttt{]}$ and* `enq` *be in* $\{\texttt{enq}, \texttt{deq}\}$.

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

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

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

4.3 Propagating Operations to the Root

In this section we explain why two `Refresh`es are enough to propagate a nodes operations to its parent.

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

Definition 21 (Successful Refresh)**.** An instance of `Refresh` is *successful* if its `CAS` in Line 320 returns `true`. If a successful instance of `Refresh` terminates, we say it is *complete.*

In the next two results we show 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 320.

Lemma 22. *If R is a successful instance of n.*`Refresh`*, then we have* $EST^{t^R}_{n.\texttt{left}} \cup EST^{t^R}_{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{320})$.

Proof. We show

$$EST^{t^R}_{n.\texttt{left}} = ops(n.\texttt{left.blocks[0..n.left.head}_{309}-1\texttt{]})$$
$$\subseteq ops(n.\texttt{blocks}_{320}) = ops(n.\texttt{blocks[0..}n.\texttt{head}_{320}\texttt{]}).$$

Line 320 stores a block `new` in n that has $\texttt{end}_{\texttt{left}} = n.\texttt{left.head}_{336}-1$. Therefore, by Definition 5, after the successful `CAS` in Line 320 we know all blocks in $n.\texttt{left.blocks[}1\cdots n.\texttt{left.head}_{336}-1\texttt{]}$ are subblocks of $\texttt{n.blocks[}1\cdots n.\texttt{head}_{310}\texttt{]}$. Because of Lemma 2 we have $n.\texttt{left.head}_{309}-1 < n.\texttt{left.head}_{336}-1$ and $n.\texttt{head}_{310} < n.\texttt{head}_{320}$. From Observation 13 the claim follows. The proof for the right child is the same. ◻

Corollary 23. *If R is a complete instance n.*`Refresh`*, then we have* $EST^{t^R}_{n.\texttt{left}} \cup EST^{t^R}_{n.\texttt{right}} \subseteq EST^{{}^Rt}_{n}$.

Proof. The left hand side is the same as Lemma 22, so it is sufficient to show when R terminates the established blocks in n are a superset of $n.\texttt{blocks}_{320}$. Line 320 writes the block `new` in $n.\texttt{blocks[}h\texttt{]}$ where h is value of $n.\texttt{head}$ read at Line 310. Because of Lemma 3 we are sure that $n.\texttt{head} > h$ when R terminates. So the block `new` appended to n at Line 320 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 24. *Consider two consecutive terminating instances R_1, R_2 of* Refresh *on internal node n by process p. If neither R_1 nor R_2 is a successful* Refresh*, then we have $EST_{n.\texttt{left}}^{t^{R_1}} \cup EST_{n.\texttt{right}}^{t^{R_1}} \subseteq EST_n^{R_2 t}$.*

Proof. Let R_1 read i from n.head at Line 310. By Lemma 3, R_1 and R_2 both cannot read the same value i. By Observation 2, R_2 reads a larger value of n.head than R_1.

Consider the case where R_1 reads i and R_2 reads $i+1$ from Line 310. As R_2's CAS in Line 320 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 310) and R_1 reads the value i from n.head. By Observation 2 we have ${}^{R_1}t < t_{310}^{R_1} < t_{310}^{R2'}$ (see Figure 13). By Lemma 23 we have $EST_{{}^{R_2'}_{310}t}^{n.\texttt{left}} \cup EST_{{}^{R_2'}_{310}t}^{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{t_{320}^{R_2'}})$. 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' to n is established by the time R_2 terminates. To summarize, ${}^{R_1}t$ is before R_2''s read of n.head ($t_{310}^{R_2'}$) and R_2''s successful CAS ($t_{320}^{R_2'}$) is before R_2's termination (t^{R_2}), so by Observation and Lemma 3 we have 13 $EST_{n.\texttt{left}}^{t^{R_1}} \cup EST_{n.\texttt{right}}^{t^{R_1}} \subseteq ops(n.\texttt{blocks}_{t_{320}^{R_2'}}) \subseteq EST_{n.\texttt{left}}^{R_2 t}$.

If R_2 reads some value greater than $i+1$ in Line 310 it means n.head has been incremented more than two times since ${}^{R_1}_{310}t$. By Lemma 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 310 and has put its block in n.blocks[$i+1$] before R_2's read of n.head at Line 310. So we have $t^{R_1} <_{310}^{R_3} t <_{320}^{R_3} t < t_{310}^{R_2} <_2^R t$. From Observation 13 on the operations before and after R_3's CAS and Lemmas 22, 3 on R_3 the claim holds. ◻

Corollary 25. $EST_{n.\texttt{left}}^{302t} \cup EST_{n.\texttt{right}}^{302t} \subseteq EST_n^{t_{303}}$

Proof. If the first Refresh in line 302 returns true then by Lemma 23 the claim holds. If the first Refresh failed and the second Refresh succeeded the claim still holds by Lemma 23. Otherwise both failed and the claim is satisfied by Lemma 24. ◻

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

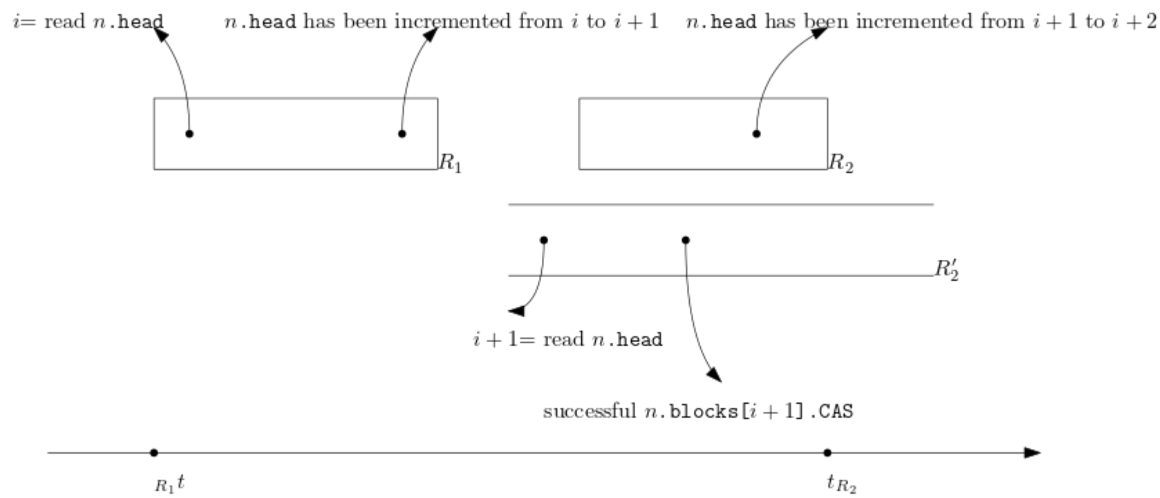

Figure 13: ${}_{R_1}t < t_{310}^{R_1} <$ incrementing n.`head` from i to $i+1 < t_{310}^{R'_2} < t_{320}^{R'_2} <$ incrementing n.`head` from $i+1$ to $i+2 < t_{R_2}$

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

Proof. A adds b to the assigned leaf of the process, establishes it at Line 603 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 25 and then calls n.`parent.Propagate` untill n is `root`. ☐

Corollary 27. *After* $l.$`Append`(b) *finishes,* b *is subblock of exactly one block in each node along the path from* l *to the* `root`*.*

Proof. By the previous corollary and Lemma 27 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 28. *Each block contains at most one operation of each process, and therefore at most p operations in total.*

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

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

Proof. The claim follows directly from Lemma 28 and the observation that each block appended to an internal node contains at least one operation, due to the test on Line 318. We can also see the blocks in the leaves have exactly one operation in the `Enqueue` and `Dequeue` routines. □

`DSearch(`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 entire sequence stored in the root.

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

Proof. From Lines 339 and 340 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 `root.blocks[`$0 \cdot \cdot end$`]` are also non-decreasing. Furthermore, since `root.blocks[0].sum`$_{\texttt{enq}} = 0$ and `root.blocks[`end`].sum`$_{\texttt{enq}} \geq e$, there is a b such that `root.blocks[`b`].sum`$_{\texttt{enq}} \geq e$ and `root.blocks[`$b-1$`].sum`$_{\texttt{enq}} < e$ by Lemma 19. Block `root.blocks[`b`]` contains $E_i(root, b)$. Lines 802–805 doubles the search range in Line 804 and will eventually reach `start` such that `root.blocks[start].sum`$_{\texttt{enq}} \leq e \leq$ `root.blocks[end].sum`$_{\texttt{enq}}$. Then, in Line 806, the binary search finds the b such that `root.blocks[`$b-1$`].sum`$_{\texttt{enq}} < e \leq$ `root.blocks[`b`].sum`$_{\texttt{enq}}$. By Corollary 19, `root.blocks[`b`]` is the block that contains $E_e(root)$. Finally i is computed using the definition of $\texttt{sum}_{\texttt{enq}}$ and Corollary 19. □

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

Proof. We are going to prove this lemma by induction on the height of node n. For the base case, suppose n is a leaf. Leaf blocks each contain exactly one operation, n`.blocks[`b`].sum`$_{\texttt{enq}} \le 1$, which means only n`.GetEnqueue(b,1)` can be called when n is a leaf. Line 403 of n`.GetEnqueue(`b`, 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`.child.GetEnqueue(`b'`, `i`)` returns $E_i(n.\texttt{child}, b')$ then n`.GetEnqueue(`b`, `i`)` returns $E_i(n,b)$. From Definition 15 of $E(n,b)$, so operations from the left subblocks come before the operations from the right subblocks in a block (see Figure 14). By Observation 18, the `num`$_{\texttt{enq-left}}$ field in n`.blocks[`b`]` is the number of `Enqueue` operations from the blocks's subblocks in the left child of n. So the ith `Enqueue` operation in n`.blocks[`b`]` is propagated from the right child if and only if i is greater than n`.blocks[`b`].num`$_{\texttt{enq-left}}$. Line 404 decides whether the ith enqueue in the bth block of internal node n is in the left child or right child subblocks of n`.blocks[`b`]`. By Definitions 5 and 10 to find an operation in the subblocks of n`.blocks[`i`]` we need to search in the range

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

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

First we consider the case where the `Enqueue` we are looking for is in the left child. There are $eb = n.$`blocks[`$b-1$`].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 $i+eb$th enqueue in `n.left`, where i' is $i + eb - n$`.left.blocks[`$b'-1$`].sum`$_{\texttt{enq}}$. The parameters in Line 405 are for searching $E_{i+eb}(n.left)$ 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 `Enqueue`s from the right subblocks, there are `n.blocks[b].num`$_{\texttt{enq-left}}$ enqueues ahead of $E_i(n,b)$ from the left child. So we need to search for $i - n$`.blocks[`b`].num`$_{\texttt{enq-left}} + n$`.blocks[`$b-$ `1].sum`$_{\texttt{enq-right}}$ in the right child (Line 409). 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 405 or 409. Finally, n`.child.GetEnqueue(`*subblock*`, `i`)` is invoked on the subblock containing $E_i(n,b)$ and it returns $E_i(n,b)$`.element` by the hypothesis of the induction. ☐

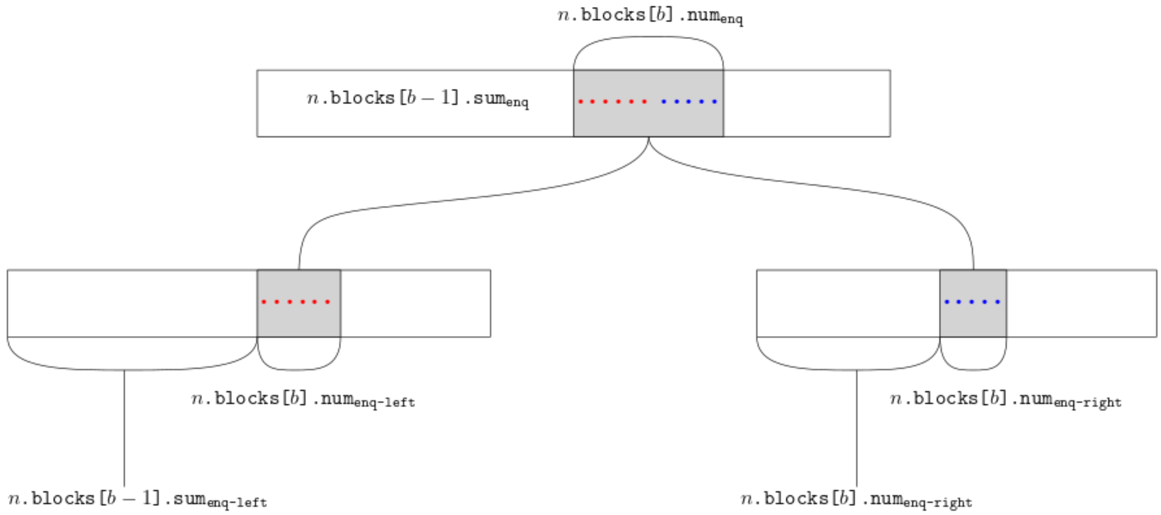

Figure 14: The number and ordering of the enqueue operations propagated from the left and the right child to n.`blocks[`b`]`. Both n.`blocks[`b`]` and its subblocks are shown in grey. `Enqueue` operations from the left subblocks (colored red), are ordered before the `Enqueue` operations from the right child (colored blue).

4.5 Correctness of IndexDequeue

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

Definition 32. If a `Refresh` instance R_1 does its `CAS` at Line 320 earlier than `Refresh` instance R_2 we say R_1 has *happened before* R_2.

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 the n`.head` changes only by the increment on Line 327. Before an instance of `Advance` increments `n.head` on Line 327, Line 326 ensures that `n.blocks[head].super` was set at Line 326. □

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

Proof. By Lemma 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 336.*

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

Proof. Let R_p be the instance of n`.parent.Refresh` that stores n`.parent.blocks[`sb`]`. By 35 if R_p propagates B it has to read a greater value than b from n`.head`, which means n`.head` was incremented from b to $b+1$ in Line 327. By Observation 33 B`.super` was already set in Line 326. The value written in B`.super` was read in Line 325,s before the `CAS` that sets B`.super`. From Observation 2 we know n`.parent.head` is non-decreasing so B`.super` $\leq sb$, since n`.parent.head` is still equal to sb when R_p executes its `CAS` at Line 320 by Invariant 6. The reader may wonder when the case b`.super` $= sb$ happens. This can happen when n`.parent.blocks[`B`.super]` $=$ `null` when B`.super` is written and R_p puts its created block into n`.parent.blocks[`B`.super]` afterwards. □

Lemma 37. *Let R_n be a* `Refresh` *that puts B in n*`.blocks[`b`]` *at Line 320. Then, the block created by one of the next two successful n*`.parent.Refresh`*es according to Definition 32 contains B and B*`.super` *is set when the second successful n*`.parent.Refresh` *reaches Line 317.*

Proof. Let R_{p1} be the first successful n`.parent.Refresh` after R_n and R_{p2} be the second next successful n`.parent.Refresh`. To derive a contradiction assume B was not propagated to n`.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`.head` in Line 336 is at most b. From Observation 2 the value R_{p2} reads in Line 312 is also at most b.

R_n puts B into n`.blocks[`b`]` so R_n reads the value b from n`.head`. Since R_{p2}'s `CAS` into n`.parent.blocks` is successful there should be a `Refresh` instance R'_p on n`.parent` that increments n`.parent` (Line 327) after R_{p1}'s Line 320 and before R_{p2}'s Line 310. We assumed $t_{320}^{R_n} < t_{320}^{R_{p1}} < t_{320}^{R_{p2}}$ by Definition 32. Finally, Line 312 is after Line 310 and R_{p2}'s 310 is after R'_p's Line 327, which is after R_n's n`.blocks.CAS`.

$$\left.\begin{array}{r}{}^{R_n}_{320}t <^{R_{p1}}_{320} t\\ {}^{R_{p1}}_{320}t <^{R_{p'}}_{327} t <^{R_{p2}}_{310} t\\ {}^{R_{p2}}_{310}t <^{R_{p2}}_{312} t\end{array}\right\} \Longrightarrow {}^{R_n}_{320} t <^{R_{p2}}_{312} t$$

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

Therefore R_{p2} reads n`.head` $= b$. R_{p2} calls n`.Advance` at Line Line 314, which ensures n`.head` is incremented from b. So the value R_{p2} reads in Line 336 of `CreateBlock` is greater than b and R_{p2}'s created block contains B. This is contradiction with our hypothesis.

Furthermore, if B`.super` was not set earlier it is set by R_{p2} call to n`.Advance` invoked from Line 314.

□

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 thr previous Lemma we know that B is propagated by second next successful `Refresh`'s `CAS` on n`.parent.blocks`. To summarize we have $n.\texttt{parent.head}_{{}^{R_{p2}}_{320}t} = n.\texttt{parent.head}_{{}^{R_{p1}}_{320}t} + 1$ and by Definition 32 and Observation 2

n.`parent.head`$_{{}^{R_{p1}}_{320}t} \leq n$.`parent.head`$_{{}^{R_n}_{320}t}$. The value that is set in B.`super` is read from n.`parent.head` after ${}^{R_n}_{320}t$. So B.`super` is equal to or one less than the index of the superblock of B. □

Now using Corollary 38 on each step of the `IndexDequeue` we prove its correctness.

Lemma 39 (`IndexDequeue` correctness)**.** *If* $1 \leq i \leq n$.`blocks[`b`].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 415). For the non-root nodes n.`IndexDequeue(`b`, `i`)` computes sb, the index of the superblock of the bth block in n, in Line 418 by Corollary 38. After that, the position of $D_i(n, b)$ in $D(n.\texttt{parent}, sb)$ is computed in Lines 419–424. By Definition 15, `Dequeue`s 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 dequeus in the left subblocks of n.`parent.blocks[`sb`]` before n.`blocks[`b`]` is considered in Line 420 (see Figure 15). 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 421) (see Figure 16). Finally `IndexDequeue` is called on n.`parent` recursively and it returns the correct response by induction hypothesis. □

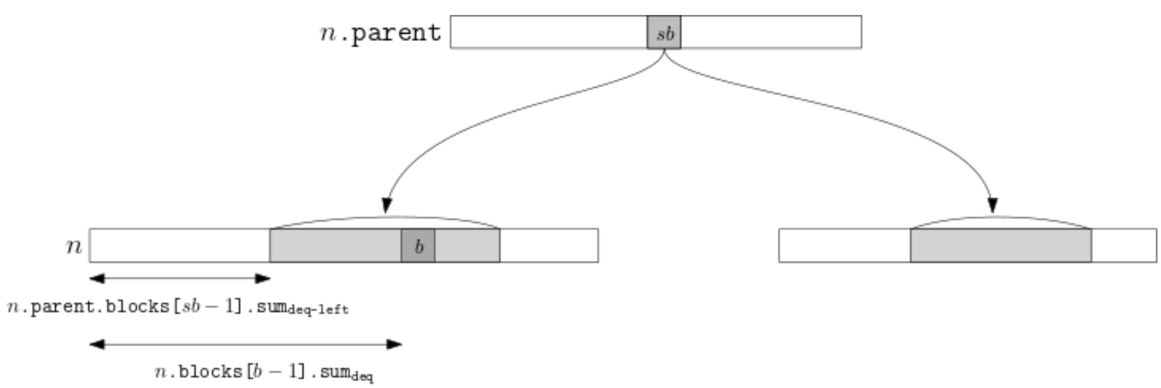

Figure 15: The number of `Dequeue` operations before $E_i(n, b)$ shown in the case where n is a left child.

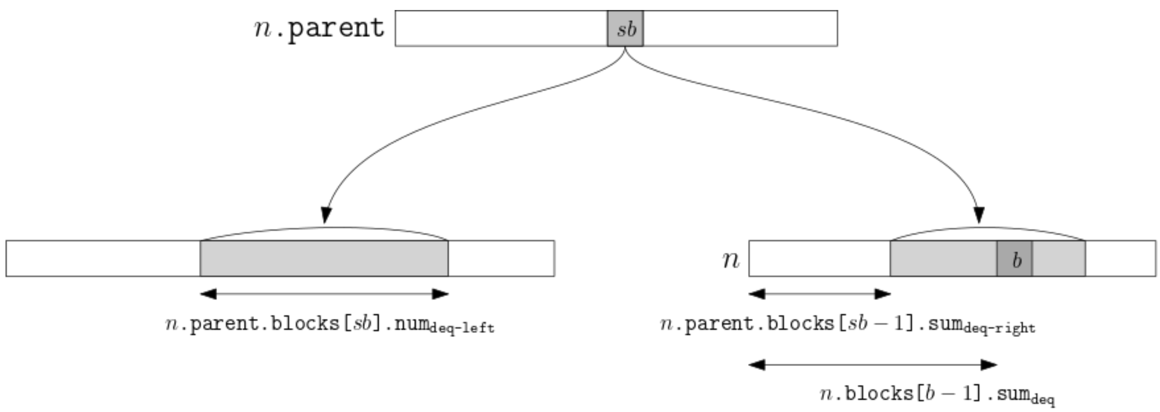

Figure 16: The number of `Dequeue` operations before $E_i(n, b)$ shown in the case where n is a right child.

4.6 Linearizability

We now prove the two properties needed for linearizability.

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

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

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 `Eequeue`s (see Definition 15).

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

Next we define the responses that `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 if 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`].size` $= 0$. We are going to show the correctness when $b = i$ assuming correctness when $b = i-1$. By Definition 15 `Enqueue` operations come before `Dequeue` operations in a block. By Lemma 18 $\texttt{num}_{\texttt{enq}}$ and $\texttt{num}_{\texttt{deq}}$ fields in a block show ther number of `Enqueue` and `Dequeue` operations in it. If there are more than `root.blocks[`$i-1$`].size` $+$ `root.blocks[`i`].`$\texttt{num}_{\texttt{enq}}$ dequeue operations in `root.blocks[`i`]` then the queue would become empty after `root.blocks[`i`]`. Otherwise the size of the queue after the bth block in the root is `root.blocks[`$b-1$`].size` $+$ `root.blocks[`b`].`$\texttt{num}_{\texttt{enq}}$ $-$ `root.blocks[`b`].`$\texttt{num}_{\texttt{deq}}$. In both cases, this is same as the assignment on Line 343. ☐

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

Lemma 44. *If operations in the root are applied with the order of L, the number of non-null* `Dequeue`*s in* `root.blocks[`$0\cdots b$`]` *is* `root.blocks[`b`].`$\texttt{sum}_{\texttt{enq}}$ $-$ `root.blocks[`b`].size`*.*

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

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

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

Proof. From Corollary 45 and Lemma 18. ☐

Lemma 47. `FindResponse(`b`, `i`)` *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 15 and Lemma 19. $D_i(root, b)$ returns `null` at Line 220 if `root.blocks[`$b-1$`].size` $+$ `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[`$b-1$`].`$\texttt{sum}_{\texttt{enq}}$ $-$ `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 222). See Figure 17.

After computing e at Line 222, the code finds b,i such that $E_i(root, b) = E_e(root)$ using `DSearch` and then finds its `element` using `GetEnqueue` (Line 223). Correctness of `DSearch` and `GetEnqueue` routines are shown in Lemmas 30 and 31. ☐

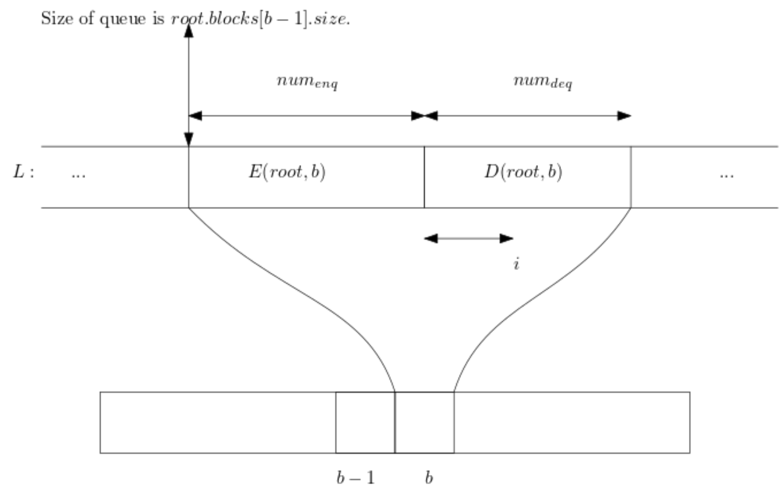

Figure 17: The position of $D_i(root, b)$.

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

Proof. `Enqueue` operations do not return any value. By Lemma 47 response of a `Dequeue` in our algorithm is same as the response from 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 [7]. By Definition 15 the linearization ordering of operations will not change as blocks containing new operations are appended to the root.

5 Analysis

Is this a Lemma or a Result or an Observation of Algorithm? Which is not used in main wait-free theorem.

Lemma 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 7 `CAS`es, one in Line 320 and two from each `Advance` in Line 314 or 321. □

Lemma 51 (`DSearch` 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* `DSearch(`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[`$b+1 \cdots end-1$`]` 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 318 we know that `num` field of the every block in the tree is greater than 0, each block has at least one operation, 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`.

So, the doubling search reaches `start` such that the `root.blocks[start].sum`$_{\texttt{enq}}$ is less than e in $O\big(\log($`root.blocks[`b`].size` $+$ `root.blocks[`end`].size`$)\big)$ steps. See Figure 18. After Line 805, the binary search that finds b also takes $O\big(\log($`root.blocks[`b`].size` $+$ `root.blocks[`end`].size`$)\big)$. Next, `i` is computed via the definition of `sum`$_{\texttt{enq}}$ in constant time (Line 807). □

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

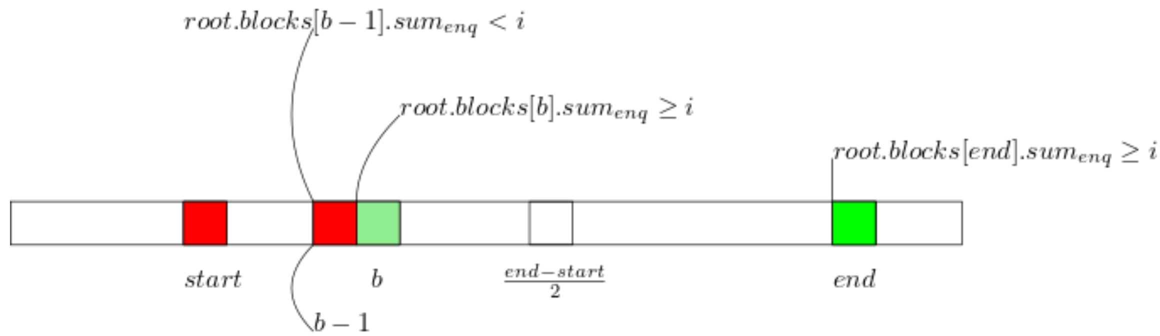

Figure 18: Distance relations between `start`, b, end.

and q_i is the size of the queue at 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 302, 303). 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 $O(logp)$ 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 `DSearch` takes $\Theta(\log($`root.blocks[b].size+root.blocks [end].size`$)$ by Lemma 51, which is $O(\text{log size of the queue when }$`Enqueue`$\text{ is linearized}) + \text{log size of the queue when }$`Dequeue`$\text{ is linearized})$. Each search in `GetEnqueue()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma 29), so `GetEnqueue()` takes $O(\log^2 p)$ steps. □

Lemma 53 (Amortized Time 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 `DSearch` time cost between the corresponding `Enqueue`, `Dequeue`, in each operation takes $O(\log^2 p + q)$ steps. □

Theorem 54. *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 an linearization of operations invoked by p processes in an asynchronous model, which we will call *Block Tree*. We also implemented two queries to know information about the ordering agreed in the block tree. Then we used the tree to implement a queue where the number of steps per operation is poly-logarithmic with respect to the size of the queue and the number of processes. Block trees can be used as a mechanism to achieve agreement among processes to construct more poly-logarithmic wait-free linearizable objects. In the next paragraphs we talk about possible improvements on block trees and the data structures that we can implement with block trees.

Reducing Space Usage The `blocks` arrays defined in our algorithm are unbounded. We could instead use the vector of Feldman, Valera-Leon and Damian Segment [5] model to use $O(n)$ space in each node, where n is the total number of operations. Their vector creates an array called `arr` of pointers to array segments. When a process wishes to write into `head` it checks whether `arr`$[\lfloor \log \texttt{head} \rfloor]$ points to an array or not. If not, it creates a shared array with size $2^{\lfloor \log \texttt{head} \rfloor}$ and tries to `CAS` a pointer to the created array into `arr`$[\lfloor \log \texttt{head} \rfloor]$. Whether the `CAS` is successful or not `arr`$[\lfloor \log \texttt{head} \rfloor]$ points to an array. When a process wishes to access the ith element it looks up `arr`$[\lfloor \log i \rfloor][i - 2^{\lfloor \log i \rfloor}]$, which takes $O(1)$ steps. Note that 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 to the `CAS` on `arr`$[i]$. Jayanti and Shun [13] present a way to initialize wait-free arrays in constant steps. The time taken to allocate arrays in an execution containing n operations is $O(p \log n)$ which if $n >> p$ we can ignore the p factor. The vector implementation has also 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}.

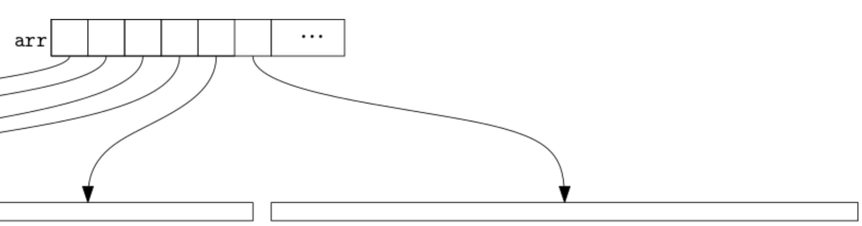

Figure 19: Array Segments

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 of its enqueues have been dequeued and the responses of all of its dequeues have been computed. We call these blocks *finished blocks*. If we help the operations of all processes to compute their responses, then we can say if block B is finished then all blocks before B are also finished. Knowing the most recent finished block in a node we can reclaim the memory taken by finished blocks. To throw the garbage in the blocks away we cannot use arrays (or vectors). We need a data structure that supports `tryAppend()`, `read(i)`, `write(i)` and `split(i)` operations in $O(\log n)$. Where `split(i)` removes all the indices less than `i`. We can use a concurrent implementation of persistent red black trees for this [1].

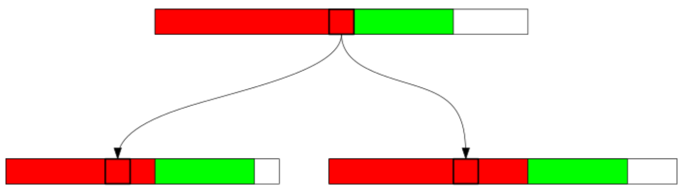

Figure 20: Finished blocks are shown with red color and unfinished blocks are shown with green color. All the subblocks of a finished block are also finished.

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 data structure. `append(e)` is implemented like `Enqueue(e)`, `get(i)` is done by calling `DSearch` with replacing the `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.

Stacks There are two reasons 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` is not very far from it in the sequence of operations: the distance is at most linear in the size of the queue. It may be possible to create wait-free poly-logarithmic implementaion of other objects whose operations satisfy these two conditions.

References

[1] Benyamin Bashari and Philipp Woelfel. An efficient adaptive partial snapshot implementation. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 545–555. ACM, 2021.

[2] Tushar Deepak Chandra, Prasad Jayanti, and King Tan. A polylog time wait-free construction for closed objects. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 287–296. ACM, 1998.

[3] Robert Colvin and Lindsay Groves. Formal verification of an array-based nonblocking queue. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*, pages 507–516. IEEE Computer Society, 2005.

[4] Faith Ellen and Philipp Woelfel. An optimal implementation of fetch-and-increment. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, page 284298, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] Steven Feldman, Carlos Valera-Leon, and Damian Dechev. An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):654–667, 2016.

[6] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, volume 6490 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2010.

[7] Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382. ACM, 2011.

[8] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In Friedhelm Meyer auf der Heide and Cynthia A. Phillips, editors, *SPAA 2010:*

Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010, pages 355–364. ACM, 2010.

[9] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124149, jan 1991.

[10] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal, editors, *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2007.

[11] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 201–210. ACM, 1998.

[12] Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In Ramaswamy Ramanujam and Sandeep Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, volume 3821 of *Lecture Notes in Computer Science*, pages 408–419. Springer, 2005.

[13] Siddhartha Jayanti and Julian Shun. Fast arrays: Atomic arrays with constant time initialization. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[14] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In Calin Cascaval and Pen-Chung Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 223–234. ACM, 2011.

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

[16] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth*

Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996, pages 267–275. ACM, 1996.

[17] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free FIFO queues. In Phillip B. Gibbons and Paul G. Spirakis, editors, *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 253–262. ACM, 2005.

[18] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In Alex Nicolau, Xiaowei Shen, Saman P. Amarasinghe, and Richard W. Vuduc, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 103–112. ACM, 2013.

[19] Niloufar Shafiei. Non-blocking array-based algorithms for stacks and queues. In Vijay K. Garg, Roger Wattenhofer, and Kishore Kothapalli, editors, *Distributed Computing and Networking, 10th International Conference, ICDCN 2009, Hyderabad, India, January 3-6, 2009. Proceedings*, volume 5408 of *Lecture Notes in Computer Science*, pages 55–66. Springer, 2009.

[20] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In Arnold L. Rosenberg, editor, *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2001, Heraklion, Crete Island, Greece, July 4-6, 2001*, pages 134–143. ACM, 2001.