Wait-free Queues with Polylogarithmic Step Complexity

Hossein Naderibeni

supervised by Eric Ruppert

October 11, 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

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 **??** 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 **??** is not linearizable since `A` has been enqueued before `B`, so it has to be dequeued first.

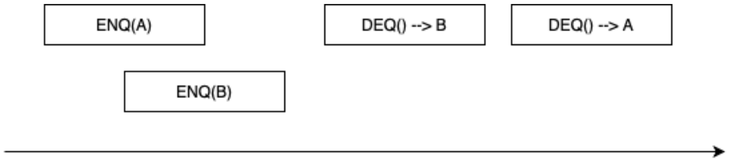


Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

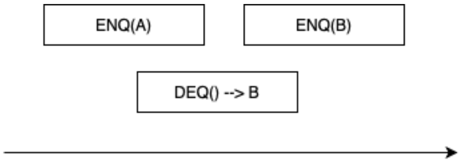


Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A` or nothing.

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 [**?**] proved an $\Omega(\log p)$ lower bound on the worst-case shared-access time complexity of p-process universal constructions. He also introduced [**?**] a construction that achieves $O(\log^2 p)$ shared accesses. Here, we first introduce a universal construction using $O(\log p)$ CAS operations [**?**]. 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 (**??**). 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.

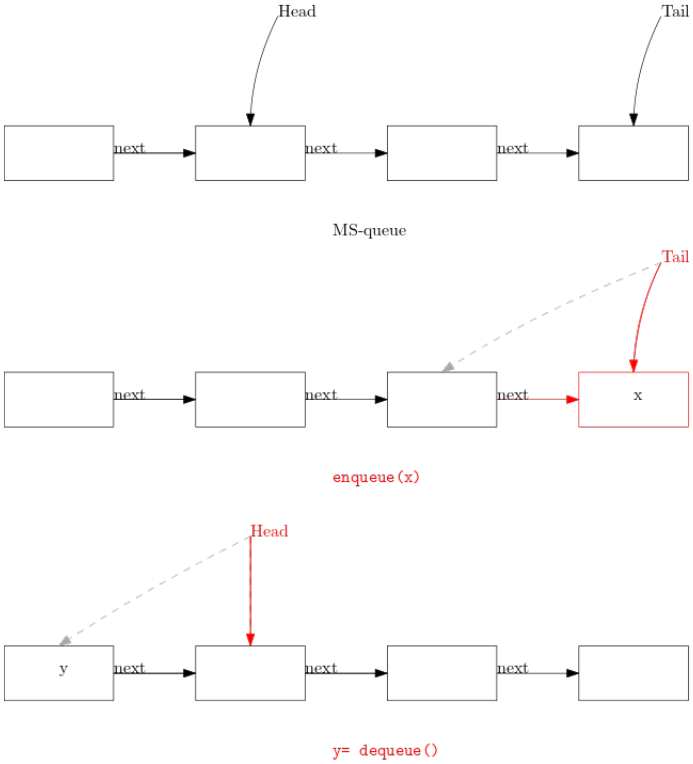
In section 4 we introduce a concurrent wait-free datastructure, to agree on the order of the operations invoked on some processes.

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 [?] 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 **??**). 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.



MS-queue

`enqueue(x)`

`y= dequeue()`

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 [?] 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 [?] 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.

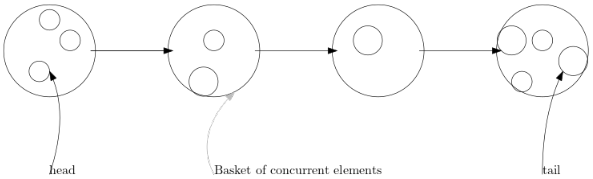


Figure 4: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do CAS. To order the operations in a basket, the mechanism in the algorithm for processes is to CAS again. The successful process will be the next one in the basket and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

Ladan-Mozes and Shavit [?] 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 $\Omega(p)$ CASes.

Hendler et al. [?] 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 [?] 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 ??). Their data structure is lock-free. Still, if the head array is empty and p processes try to enqueue simultaneously, the step complexity remains $\Omega(p)$.

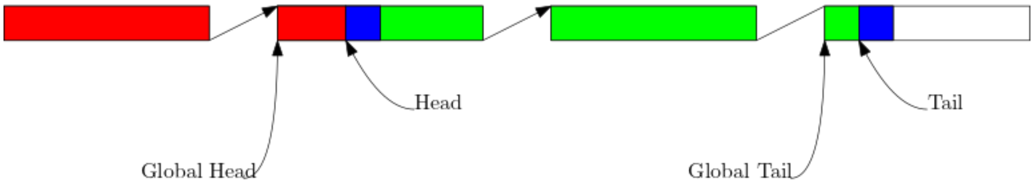


Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

Kogan and Petrank [?] 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* [?]. It is not limited to list-based queues and array-based queues share the CAS retry problem as well [?, ?, ?] . 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 [?]. 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 [?]. He also introduced a construction that achieves $\mathrm{O}(\log^2 p)$ shared accesses [?]. 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 [?]. 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

Paragraph titles will be removed after polishing.

What is a Queue In our model there are p processes doing `Enqueue` and `Dequeue` operations on a queue concurrently. We design a 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 invocation. 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.

How we use Jayanti and why we are better Jayanti and Petrovic introduced a wait-free polylogarithmic multi-enqueuer single-dequeuer queue [**?**]. 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 use `CAS` operations with big words and does not put a limit on the number of concurrent operations.

Introduce tree There is a shared tree among the processes (see Figure **??**) 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 linearization ordering. *TODO::Add sequence to nodes*

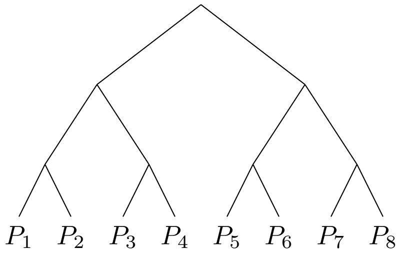
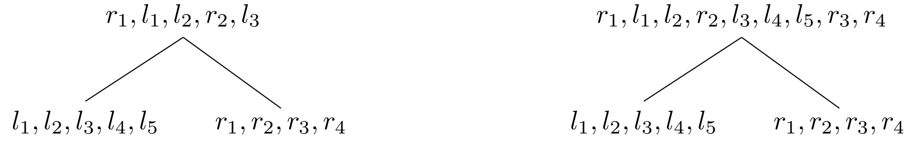


Figure 6: Each of the processes $P_1, P_2 ... P_p$ has a leaf and in each node there is an ordering of operations stored. Each process tries to propagate its operations up to the root, which stores a total ordering of all operations.

Introduce Double Refresh To do a propagate step on node n in the tree, a process observes the operations in both of n's children that are not already in n, and then tries to append them to the sequence stored in n. We call this procedure n.`Refresh()`. A process doing a `Refresh` on n with successful append helps other processes doing `Refresh` on n concurrently to propagate their operations up to the parent. The key idea is 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 the node after the second failed `Refresh`. This is because if both of the `Refresh`es on n fail to append then there is another instance of `Refresh` in between which has gathered its new operations after the first failed `Refresh` and succeeded to do an append. This `Refresh` appends the operations that the first failed `Refresh` was trying to append.


r_1, l_1, l_2, r_2, l_3
l_1, l_2, l_3, l_4, l_5
r_1, r_2, r_3, r_4
$r_1, l_1, l_2, r_2, l_3, l_4, l_5, r_3, r_4$
l_1, l_2, l_3, l_4, l_5
r_1, r_2, r_3, r_4

(a) Before the `Refresh`.

(b) New operations are appended.

Figure 7: Before and after of a n.`Refresh` with successful append. Operations propagating from the left child are numbered with l and from the right child by r.

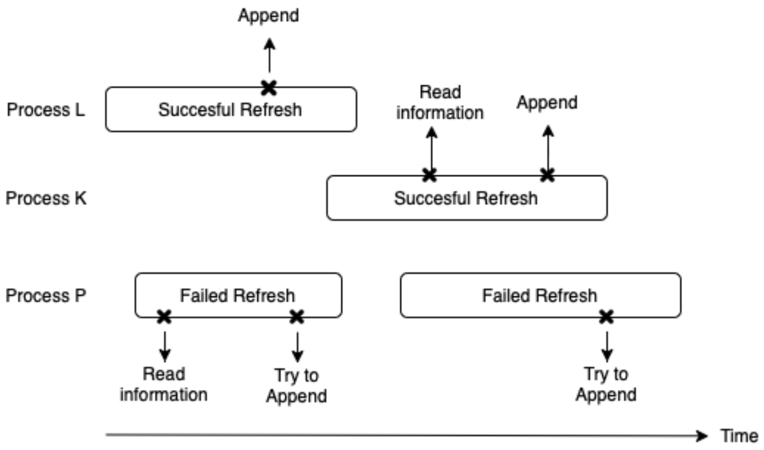


Figure 8: The second failed Refresh is assuredly concurrent with a successful `Refresh`.

Introduce CAS We use `CAS` (Compare & Swap) instructions to implement the `Refresh`'s Try to Append mechanism in described in the previous paragraph. After a process appends its operation into its leaf it can call `Refresh` on the path up to root two times on each node. So with $O(\log p)$ `CAS`es per operations we can have a tree agreeing on the linearization. This cooperative solution allows us to overcome the CAS Retry Problem.

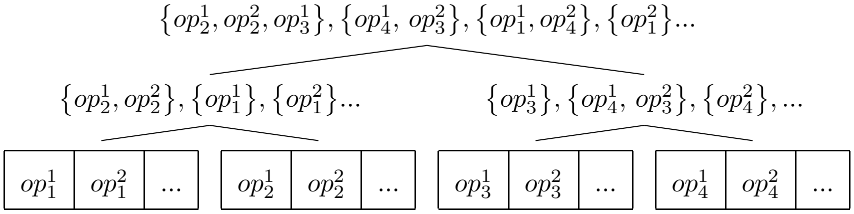

$\{op_2^1, op_2^2, op_3^1\}, \{op_4^1, op_3^2\}, \{op_1^1, op_4^2\}, \{op_1^2\}$...
$\{op_2^1, op_2^2\}, \{op_1^1\}, \{op_1^2\}$...
$\{op_3^1\}, \{op_4^1, op_3^2\}, \{op_4^2\}$, ...
op_1^1
op_1^2
...
op_2^1
op_2^2
...
op_3^1
op_3^2
...
op_4^1
op_4^2
...

Figure 9: Leaves are for processes P_1 to P_4 from left to right. In each internal node and one can arbitrarily linearize the sets of concurrent operations propagated together in a `Refresh` among themselves. 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 window have range in common they can be linearized in any order between themselves.

Introduce Blocks 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 no be poly-logarithmic time. Instead we use 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. We can only keep track of set of operations in each `Refresh` and then define the linearization only in root (see Figure **??**). 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*, a piece of information about the operations in a `Refresh` step. A block contain the number of operations from the left and the right child in a `Refresh` procedure. A node stores an array of blocks of operations propagted up to it. A propagate step aggregates the new blocks in children into a new block and puts it in the parent. In each `Refresh` there is at most one operation from each process trying to be propagated, because one operation cannot invoke two operations concurrently. 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.

Introduce Get and Index 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 and operation op we can apply all the operations till op in the root on a sequential instance of the object and then return the response. But Having this agreement by itself is not enough for an efficient queue. A process may wish to know (1) the ith propagated operation or (2) the rank of a propagated operation in the linearization. We will explain how to use (1) and (2) to construct a queue in the next paragraph.

Explain pieces of a block helping to do Get and Index After propagating an operation `op` to the root, processes can find out information about the linearization ordering using (1) and (2). Each block stores an extra constant amount of information (like prefix sums) to allow binary searches to find the required block in a node quickly.

To get the ith enqueue in the root, we can find the block B containing ith element in the root, and then recursively find the subblock of B that contains ith element. To make this recursive 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. We also store pointers to determine the range of subblocks of a block to make binary search faster. In each block, we store the prefix sum of operations from the left child and the right child. Moreover, for each block, we store two pointers to the last left and right subblock of it (see Figures **??** and **??**). We know a block size is at most p, so binary search takes at most $O(\log p)$ time, since the pointers of a block and its previous block reduce the search range size to $O(p)$.

Introduce how to create Queue using Get and Index In our case of implementing a queue, we can make an assumption that one process only wishes to know the rank of a `Dequeue` and one tries to get an `Enqueue` with an specific rank. `Enqueue`s and `Dequeue`s are appended to the tree and when we want to find the response to a `Dequeue`, we compute the place of the `dequeue` in the linearization. Then, using the rank of the `Dequeue` among `Deqeueue`s and the information about the size of the queue stored in the root we compute which `Enqueue` is the answer to the `Dequeue` or if the answer is null.

A non-null `Dequeue` is one that returns a non-null value. If the queue is non-empty and r `Dequeue` operations have returned a non-null response, then ith `Dequeue` returns the input of the $r+1$th `Enqueue`.

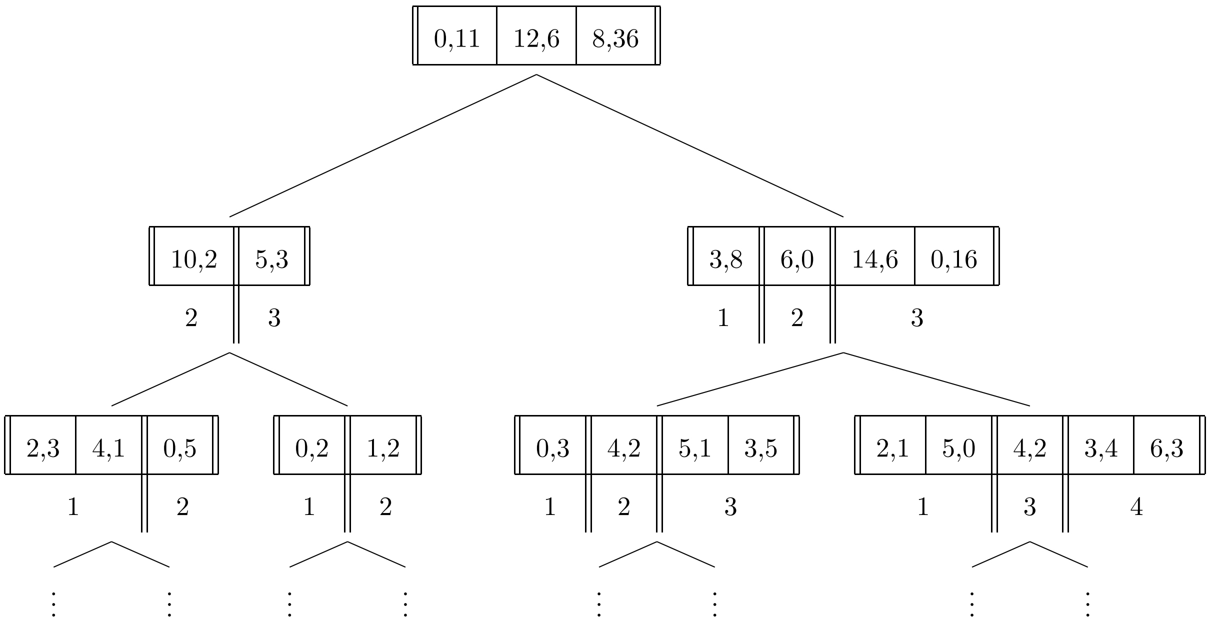
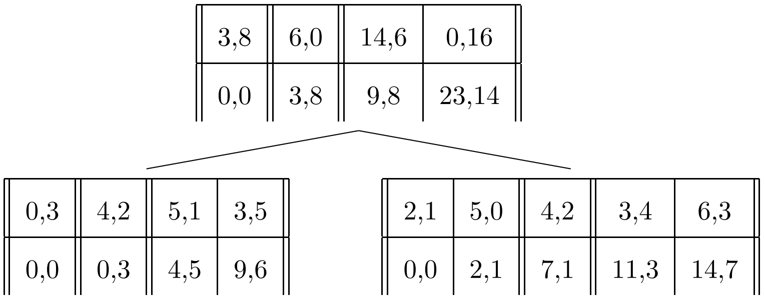


Figure 10: Showing concurrent operation sets with blocks. Each block consists of a pair(left, right) indicating the number of operations from the left and the right child, respectively. Block (12,6) in the root contains blocks (10,2) from the left child and (6,0) from the right child. Blocks between two lines || are propagated together to the parent. For example, Blocks (2,3) and (4,1) from the leftmost leaf and (0,2) from its sibling are propagated together into the block (10,2) in their parent. The number underneath a group of blocks in a node indicates which block in the node's parent those blocks were propagated to. Each block b in node n is the aggregation of blocks in the children of n that are newly read by thePropagate() step that created block b. For example, the third block in the root (8,36) is created by merging block (5,3) from the left child and (14,6) and (0,16) from the right child. Block (5,3) also points to elements from blocks (0,5) and (1,2). We choose to linearize operations in a block from the left child before those from the right child as a convention. Operations within a block of the root can be ordered in any way that is convenient. In effect, this means that if there are concurrent new blocks in a Refresh() step from several processes we linearize them in the order of their process ids. So for example operations aggregated in block (10,2) are in the order (2,3),(4,1),(0,2). All blocks from the left child with come before the right child and the order of blocks of each child is preserved among themselves.



3,8	6,0	14,6	0,16
0,0	3,8	9,8	23,14

0,3	4,2	5,1	3,5
0,0	0,3	4,5	9,6

2,1	5,0	4,2	3,4	6,3
0,0	2,1	7,1	11,3	14,7

Figure 11: Using Prefix sums in blocks. When we want to find block b elements in its children, we can use binary search. The number below each block shows the count of elements in the previous blocks.

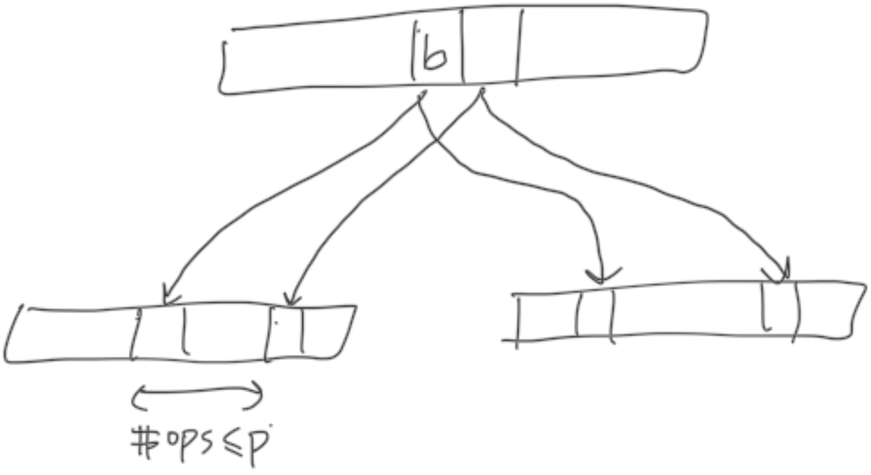


Figure 12: Block have pointers to the starting block of theirs for each child.

So in order to answer a `Dequeue`, it iss sufficient to know whether the queue is empty or not and the number of previous non-null `dequeues`.

We know we can linearize operations in a block in any order; here, we choose to decide to put `Enqueue` operations in a block before `Dequeue` operations. In the next example, operations in a cell are concurrent. `Dequeue()` operations return `null, 5, 2, 1, 3, 4, null` respectively. We will next describe how these values can be computed efficiently.

Now, we claimed that by knowing the size of the queue, we can compute the rank of the resulting `Enqueue()`. We apply this approach to blocks; if we store the size of the queue after each block of op-

DEQ	ENQ(5), ENQ(2), ENQ(1), DEQ	ENQ(3), DEQ	ENQ(4), DEQ, DEQ, DEQ, DEQ

Table 1: An example history of operation blocks on the queue. Enqueues are shown with ENQ and Dequeues are shown with DEQ.

erations happens, we can compute each Dequeue's index of result in O(1) steps.

	DEQ	ENQ(5), ENQ(2), ENQ(1), DEQ	ENQ(3), DEQ	ENQ(4), DEQ, DEQ, DEQ, DEQ
#ENQs	0	3	1	1
#DEQs	1	1	1	4
size	0	2	2	0

Table 2: Augmented history of operation blocks on the queue

Size of the queue the *b*th block could be computed using these equations.

```
size[b] = max(size[b-1] + enqueues[b] - dequeues[b], 0)
```

```
non-null dequeues[b] = enqueues[b] - size[b]
```

Given DEQ is in block B, response of DEQ is the Enqueue with index non-null Dequeus[b-1]+ index of DEQ in the block's Dequeus in the root, if (size[b-1]- index of DEQ in the B's Dequeus >=0). Otherwise it would be null

3.1 Details of Implementation

Block Block b contains subblocks in the left and right children. The left subblocks of b are some consecutive blocks in the left child starting from where the block prior to b ended. See Figure **??** .

We store ordering among operations in the tournament tree constructed by nodes. In each node we store pointers to its parent and children, an array of blocks and the index head of the first empty entry in blocks. Furthermore, in leaf nodes there is an array of operations where each operation is stored in one cell with the same index in blocks. There is a counter in each node incrementing after a successful Refresh step. It means after that some bunch of blocks in a node have propagated into the parent then the counter increases. Each new block added to a node sets its time regarding counter. This helps us to know which blocks have aggregated together to a block, not precisely though. We also store the index of the aggregated block of a block with time i in super[i].

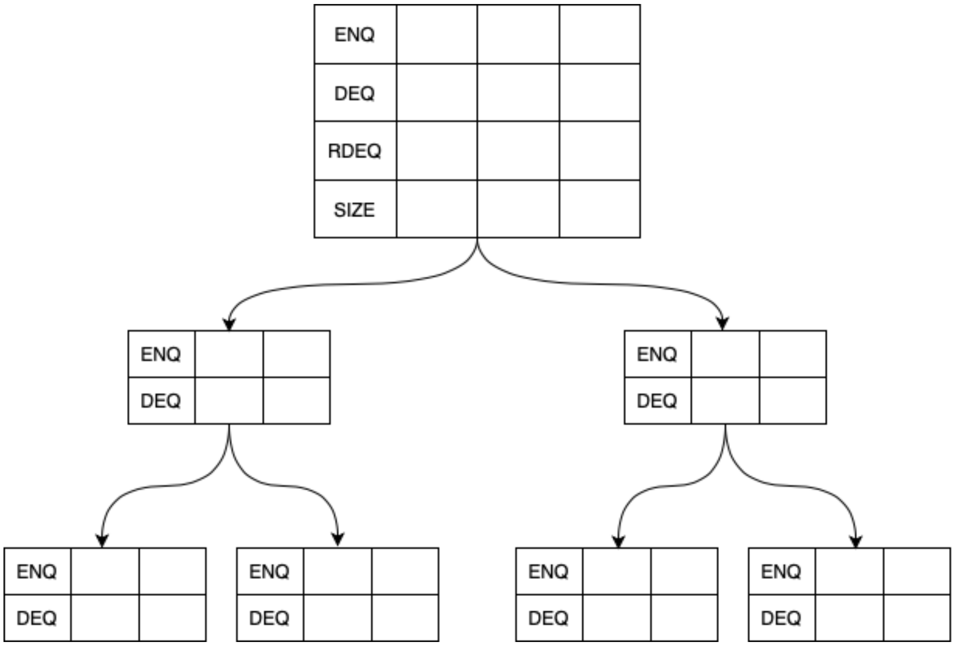


Figure 13: Fields stored in the Queue nodes.

In each `block` we store 4 essential stats that implicitly summarize which operations are in the block $\texttt{num}_{\texttt{enq-left}}$, $\texttt{num}_{\texttt{deq-left}}$, $\texttt{num}_{\texttt{enq-right}}$, $\texttt{num}_{\texttt{deq-right}}$. In order to make `BinarySearch()`es faster we store prefix sums as well and there are some more general stats that help to make pseudocode more readable but not necessary.

To compute the head of the `queue` before a `dequeue` two more fields are stored in the root `size` and $\texttt{sum}_{\texttt{non-null deq}}$. `size` in a `block` shows the number of elements after the `block` has finished and $\texttt{sum}_{\texttt{non-null deq}}$ is the total number of non-null dequeues till the `block`.

Enqueue(e) An `Enqueue` operation does not return a response, so to do an `enqueue` operation in our algorithm it is sufficient to just append the `Enqueue` operation to the root, 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.

I believe the code is readable and we do not need references to code.

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

FindResponse(b,i) To compute the response of the ith `dequeue` in bth block of the root Line **??** computes whether the queue is empty or not. If there are more dequeues than enqueues the queue would become empty before the requeusted dequeue. If the queue was not empty, Line **??** 222 computes the position of the eqneueue which is response to the dqeueue. Knowing the response is the eth enqueue in the root which is before the bth block we find the block and position contaiing the qnueue operation using `DSearch` and after that `GetEnqueue` finds its `element`.

Append(blk) `head` field is the index of the first empty index in `blocks` array in a `LeafBlock`. `Append(`B`)` adds B to the end of the `blocks` 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 subblock of a block in the `root`. 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.

n.Propagate() `Propagate(`n`)` uses the double refresh idea and invokes two `Refresh`es on n in Lines **??** and **??**. Then invokes the `Propagate` on n`.parent` recursivly till it reaches the root.

n.Refresh() n`.Refresh` goal is to create a block of n's children new blocks and append it to n`.blocks`. From Lines 311 to 316 **??** n`.Refresh` helps to `Advance` n's children. `Advance` increments the children's `head` and set the `super` field of the last block in n. The reason behind helping `super` field is explained later in `IndexDequeue` paragraph. After helping to `Advance` children, the value h from n`.head` is read at Line **??** and `new` block is created in Line **??**. `new` is going to be inserted into n`.block[h]`. The reason why reading `head` and `CreateBlock` are in two lines is that maybe the `head` value changes from reading it till creating a block to be inserted to `head`. Then if `new` is empty `Refresh` returns `true` because there is no new operations to propagate an it is unnecessary to add an empty block to the tree. Later we will assume all blocks contain at least one operation. Line **??** tries to install `new`, if it was succsuful all is good, if not it means someone else has put a block earlier into `head` or maybe the `head` was not empty at first point. Anyway `Refresh` tries to update the `head` and `super` field of n`.blocks[h]` at Line **??**.

CreateBlock n`.CreateBlock(h)` returns a block containing new operations of `n`'s children. The `new` block created in Line 333's fields are filled with symmtery for both left and right directions. $\texttt{index}_{\texttt{last}}$ is the index of the last subblock in the direction and $\texttt{index}_{\texttt{prev}}$ is the index of the prvious block of the first direct subblock to be aggregated into `new`. With this said, n`.blocks[`i'`]`$\texttt{.end}_{\texttt{dir}}$ stores the index of the rightmost direct subblock of direction `dir` child of n.`blocks`$[i-1]$. Then $\texttt{sum}_{\texttt{enq-dir}}$ is computed with the sum of the from computing the number of enqueue operations in the `new` block and the value stored in n.`blocks`$[i-1]$.$\texttt{sum}_{\texttt{enq-dir}}$. $\texttt{sum}_{\texttt{deq-dir}}$ is also the same. Then if `new` block is going to be installed in the `root`, the `size` field is computed.

should I get into details of computing size and sumenq?

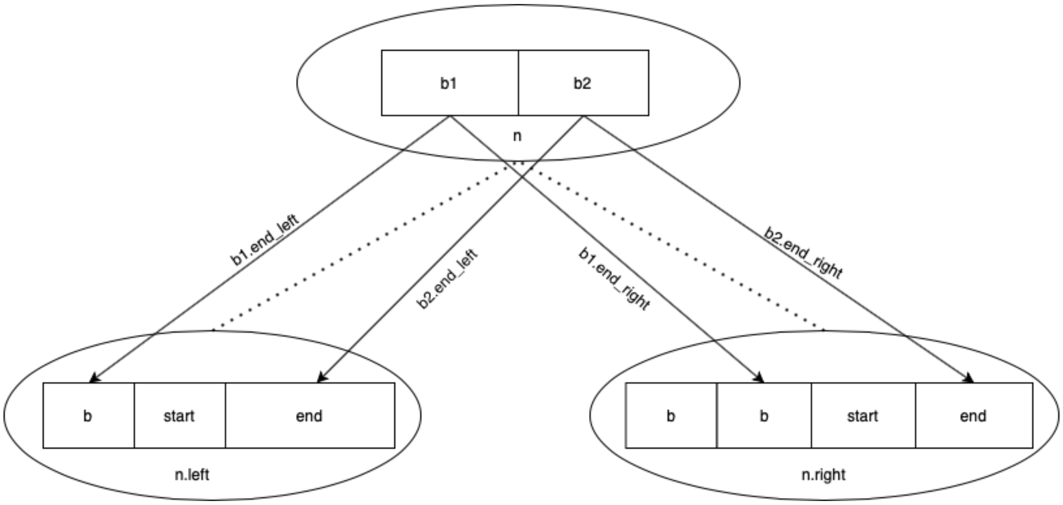


Figure 14: Snapshot of a CreateBlock

DSearch(e,end) We can show an operation in a node in two formats, 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 in that block. If we know tuple of block and rank in the block of an operation (E) we can find the this tuple in the subblock containig the operation in poly log time. To find the response of a dequeue, the information we know about the response enquue is like the first way of showing the operation in the root (E_i(`root`)). So we have to find the block in the root containing the `e`th enqueue in the root. We also know `end` which the `e`th enqueu is before or in the `root.blocks[end]`. `DSearch` uses the Doubling idea to find the range that the answer block is there (Lines **??**-**??**) and then tries to find the response with binary search (Line 806**??**).

GetEnqueue(b,i) n.`GetEnqueue(b,i)` returns the `element` of `i`th enqueue in b th block of n. The range of subblocks of a block can be determined with $\texttt{end}_{\texttt{dir}}$ fields of the block and its previous block. Then the subblock can be found usig binary search on $\texttt{sum}_{\texttt{enq}}$ field.

IndexDequeue(b,i) Let i be the value R_n, a successful instance of Refresh on node n reads from n.head. R_n does a successful CAS(null, b) into n.blocks[i]. Let p be n.parent. Without loss of generality for the rest of this section assume n is the left child of p. From Lemma ?? we know there could be only one p.Refresh propagating b. Let R_p be the first successful p.Refresh that reads some value greater than i for left.head and contains b in its created block in Line ??. Let the index of the block R_p put in p.blocks be j.

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 is to set the super field of b by R_n after propagating b to p. This solution would not be efficient because there might be p subblocks in the block R_p propagated needing to update the super field. However intuitively, once b is installed, its superblock is going to be close to n.parent.head at the time of installation. One idea is that if we know the approximate position of the superblock of b then we can search for the real superblock when we wished to know the superblock of b i.e. 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 n.parent.head while creating b, but the problem is that there might be many p.Refreshes that could happen after reading p.head by R_n and before propagating b to p. If we set b.super to p.head after appending b to n.blocks (Line ??), 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 n.parent help fill in the value of b.super.

Block b is appended to n.blocks[h] on Line ??. After appending b, b.super is set on Line ?? of a call to Advance from n.Refresh by the same process or another process or maybe an n.parent.Refresh. We want to bound how far b.super is from the index of b's superblock, which is created by a successful n.parent.Refresh that propagates b.

4 Pseudocode

Algorithm Tree Fields Description

◇ *Shared*

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

◇ *Local*

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

▶ *Node*

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

▶ *Block*

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

▶ *RootBlock* extends *InternalBlock*

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

▶ *InternalBlock* extends *Block*

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

▶ *LeafBlock* extends *Block*

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

Abbreviations used in the code and the proof of correctness.

- blocks[b].sum_{x}=blocks[b].$\text{sum}_{\text{x-left}}$+blocks[b].$\text{sum}_{\text{x-right}}$ (for b≥0 and x ∈ {enq, deq})
- blocks[b].num_{x}=blocks[b].sum_{x}-blocks[b-1].sum_{x}

  (for 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 with null 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

<int, int> FindResponse(int b, int i)                                   ▷ Returns the response to $D_{root,b,i}$.
    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$_{enq}$                ▷ $E_e(root)$ is the response.
        return root.GetEnqueue(root.DSearch(e, b))
    end if
end FindResponse
```

Algorithm Root

```
    ⇝ Precondition: root.blocks[end].sum$_{enq}$ $\geq$ e
<int, int> DSearch(int e, int end)                ▷ Returns <b,i> such that $E_e(root) = E_i(root, b)$.
    start= end-1
    while root.blocks[start].sum$_{enq}$$\geq$e do
        start= max(start-(end-start), 0)
    end while
    b= root.BinarySearch(sum$_{enq}$, e, start, end)
    i= e- root.blocks[b-1].sum$_{enq}$
    return <b,i>
end DSearch
```

Algorithm Leaf

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

Algorithm *Node*

```
void Propagate()
    if not Refresh() then
        Refresh()
    end if
    if this is not root then
        parent.Propagate()
    end if
end Propagate

boolean Refresh()
    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
    h= head
    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 field f greater than or equal to i
▷ Does a binary search for the value i of the given prefix sum field. Returns the index of the leftmost block in blocks[start..end] whose field f is $\geq$ i.
int BinarySearch(field f, int i, int start, int end)
    return min{j: blocks[j].f$\geq$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
        $index_{last}$= dir.head-1
        $index_{prev}$= blocks[i-1].$end_{dir}$
        new.$end_{dir}$= $index_{last}$
        $block_{last}$= dir.blocks[$index_{last}$]
        $block_{prev}$= dir.blocks[$index_{prev}$]
        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$_{enq}$ $\geq$ i $\geq 1$
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 $\leq$ blocks[b].num$_{enq-left}$ then              ▷ $E_i(this, b)$ is in the left child of this node.
        subBlock= left.BinarySearch(sum$_{enq}$, 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(sum$_{enq}$, i+right.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$_{deq}$ $\geq$ 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
```

5 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).

5.1 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 **??** 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* **??** *of R.*

Proof. If the `CAS` in Line **??** is successful then the claim holds. Otherwise n`.head` has changed from the value that was read in Line **??**. By Observation **??** 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 **??**, which updates n.blocks[h] where h is the value read from n.head in Line **??**. Since the CAS in Line **??** 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 **??** 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 **??** 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 **??** and **??**. For Line **??**, n.blocks[h] $\neq$ null because of the if condition in Line **??**. For Line **??**, Line **??** was finished before doing **??**. Whether Line **??** 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].end$_{\texttt{left}}$ $\geq$ n.blocks[$i-1$].end$_{\texttt{left}}$ *and* n.blocks[i].end$_{\texttt{right}}$ $\geq$ n.blocks[$i-1$].end$_{\texttt{right}}$.

Proof. Consider the block b written into `n.blocks[`i`]` by `CAS` at Line **??**. Block b is created by the `CreateBlock(`i`)` called at Line **??**. Prior to this call to `CreateBlock(`i`)`, `n.head` $= i$ at Line **??**, so `n.blocks[`$i-1$`]` is already a non-null value b' by Invariant **??**. Thus, the `CreateBlock(`$i-1$`)` that created b' terminated before the `CreateBlock(`i`)` that creates b is invoked. The value written into b`.end`$_{\texttt{left}}$ at Line **??** of `CreateBlock(`i`)` was one less than the value read at Line **??** of `CreateBlock(`i`)`. Similarly, the value in `n.blocks[`$i-1$`].end`$_{\texttt{left}}$ was one less than the value read from `n.left.head` during the call to `CreateBlock(`$i-1$`)`. By Observation **??**, n`.left.head` is non-decreasing, so b'`.end`$_{\texttt{left}} \leq b$`.end`$_{\texttt{left}}$. The proof for `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 **??** we have n`.blocks[`i`].end`$_{\texttt{left}} \leq n$`.blocks[`$j-1$`].end`$_{\texttt{left}}$, so the ranges $[n$`.blocks[`$i-1$`].end`$_{\texttt{left}}+1 \cdots n$`.blocks[`$i$`].end`$_{\texttt{left}}]$ and $[n$`.blocks[`$j-1$`].end`$_{\texttt{left}}+1 \cdots n$`.blocks[`$j$`].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 **??**. □

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 **??**. Assume *op* is in the subblocks of both $n.blocks[i]$ and $n.blocks[j]$. From Corollary **??** 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 **??**, **??**. □

5.2 Ordering Operations

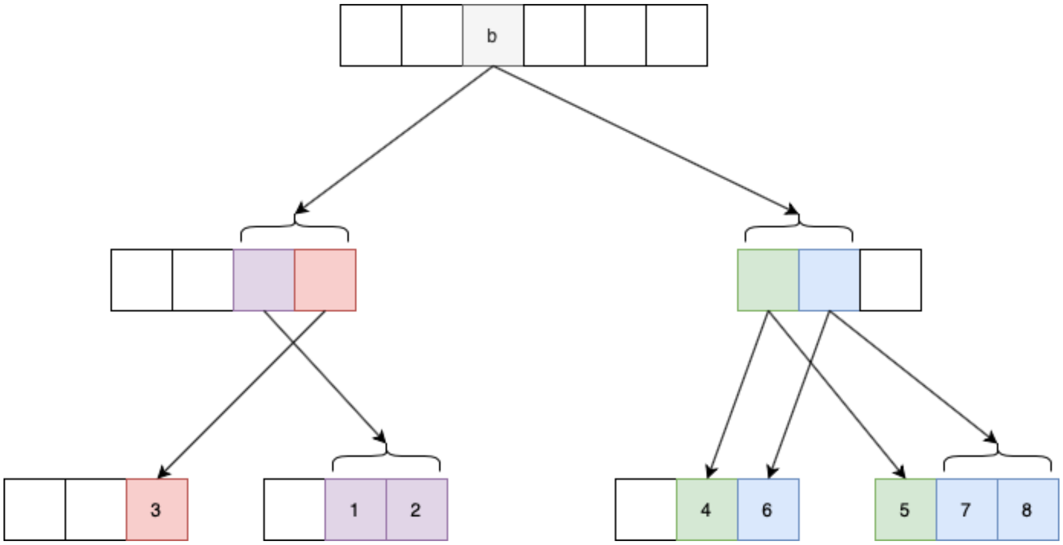


Figure 15: Order of operations in b. Operations in the leaves are ordered with numerical order shown in the drawing.

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 **??** 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

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

- $E_i(n,b)$ 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

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

- $D_i(n, b)$ 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.

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

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*`.blocks[`*b*`]` *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) $B\texttt{.sum}_{\texttt{enq}} = \Big|E(n, 1) \cdot E(n, 2) \cdots E(n, b)\Big|$

5.3 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 ${}^{op}_{l}t$ be the time immediately after running Line l of operation *op*. We sometimes suppress *op* and write t_l or ${}_lt$ if *op* is clear 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 **??** 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 **??**.

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

Proof. We show

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

$$\subseteq ops(n.\texttt{blocks}_{??}) = ops(n.\texttt{blocks[0..}n.\texttt{head}_{??}\texttt{]}).$$

Line **??** stores a block `new` in n that has $\texttt{end}_{\texttt{left}} = n.\texttt{left.head}_{??} - 1$. Therefore, by Definition **??**, after the successful `CAS` in Line **??** we know all blocks in $n.\texttt{left.blocks[}1 \cdots n.\texttt{left.head}_{??} - 1\texttt{]}$ are subblocks of $\texttt{n.blocks[}1 \cdots n.\texttt{head}_{??}\texttt{]}$. Because of Lemma **??** we have $n.\texttt{left.head}_{309} - 1 < n.\texttt{left.head}_{??} - 1$ and $n.\texttt{head}_{??} < n.\texttt{head}_{??}$. From Observation **??** 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 **??**, so it is sufficient to show when R terminates the established blocks in n are a superset of $n.\texttt{blocks}_{??}$. Line **??** writes the block `new` in $n.\texttt{blocks[}h\texttt{]}$ where h is value of $n.\texttt{head}$ read at Line **??**. Because of Lemma **??** we are sure that $n.\texttt{head} > h$ when R terminates. So the block `new` appended to n at Line **??** 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 $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq EST^{R_2 t}_{n}$.*

Proof. Let R_1 read i from n`.head` at Line **??**. By Lemma **??**, R_1 and R_2 both cannot read the same value i. By Observation **??**, 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 **??**. As R_2's `CAS` in Line **??** returns `false`, there is another successful instance R_2' of n`.Refresh` that has done a `CAS` successfully into $n.\texttt{blocks}[i+1]$ before R_2 tries to `CAS`. R_2' creates its block `new` after reading the value $i+1$ from n`.head` (Line **??**) and R_1 reads the value i from n`.head`. By Observation **??** we have ${}^{R_1}t < t^{R_1}_{\textbf{??}} < t^{R2'}_{\textbf{??}}$ (see Figure **??**). By Lemma **??** we have $EST^{n.\texttt{left}}_{\substack{R_2' \\ \textbf{??}}t} \cup EST^{n.\texttt{right}}_{\substack{R_2' \\ \textbf{??}}t} \subseteq ops(n.\texttt{blocks}_{t^{R_2'}_{\textbf{??}}})$. Also by Lemma **??** 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` ($t^{R_2'}_{\textbf{??}}$) and R_2''s successful `CAS` ($t^{R_2'}_{\textbf{??}}$) is before R_2's termination (t^{R_2}), so by Observation and Lemma **??** we have **??** $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{t^{R_2'}_{\textbf{??}}}) \subseteq EST^{R_2 t}_{n.\texttt{left}}$.

If R_2 reads some value greater than $i+1$ in Line **??** it means n`.head` has been incremented more than two times since ${}^{R_1}_{\textbf{??}}t$. By Lemma **??**, 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 $n.\texttt{blocks}[i+1]$. R_3 read $n.\texttt{head} = i+1$ at Line **??** and has put its block in $n.\texttt{blocks}[i+1]$ before R_2's read of n`.head` at Line **??**. So we have $t^{R_1} <^{R_3}_{\textbf{??}} t <^{R_3}_{\textbf{??}} t < t^{R_2}_{\textbf{??}} <^{R}_{2} t$. From Observation **??** on the operations before and after R_3's `CAS` and Lemmas **??**, **??** on R_3 the claim holds. □

Corollary 25. $EST^{\textbf{??}t}_{n.\texttt{left}} \cup EST^{\textbf{??}t}_{n.\texttt{right}} \subseteq EST^{t\textbf{??}}_{n}$

Proof. If the first `Refresh` in line **??** returns `true` then by Lemma **??** the claim holds. If the first `Refresh` failed and the second `Refresh` succeeded the claim still holds by Lemma **??**. Otherwise both failed and the claim is satisfied by Lemma **??**. □

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

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

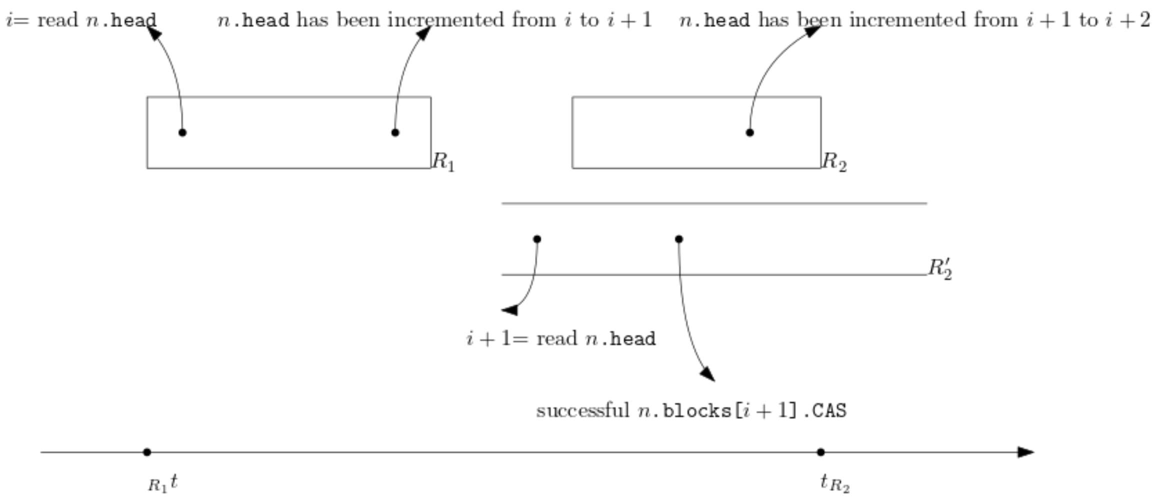

i= read n.head
n.head has been incremented from i to $i+1$
n.head has been incremented from $i+1$ to $i+2$
R_1
R_2
R_2'
$i+1$= read n.head
successful n.blocks[$i+1$].CAS
${}_{R_1}t$
t_{R_2}

Figure 16: ${}_{R_1}t < t^{R_1}_{??} <$ incrementing n.head from i to $i+1 < t^{R_2'}_{??} < t^{R_2'}_{??} <$ incrementing n.head from $i+1$ to $i+2 < t_{R_2}$

Proof. A adds b to the assigned leaf of the process, establishes it at Line **??** 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 **??** 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 **??** there is exactly one block in each node containing b. □

5.4 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 **??**, 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 **??**. □

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

Proof. The claim follows directly from Lemma **??** and the observation that each block appended to an internal node contains at least one operation, due to the test on Line **??**. 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* $\texttt{root.blocks[}end\texttt{]} \neq \texttt{null}$ *and* $1 \leq e \leq \texttt{root.blocks[}end\texttt{].sum}_{\texttt{enq}}$, `DSearch(`$e$`, `$end$`)` *returns* `<`$b$`, `$i$`>` *such that* $E_i(root, b) = E_e(root)$.

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

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 **??** 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 **??** of $E(n,b)$, so operations from the left subblocks come before the operations from the right subblocks in a block (see Figure **??**). By Observation **??**, 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 **??** 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 **??** and **??** to find an operation in the subblocks of n`.blocks[`i`]` we need to search in the range

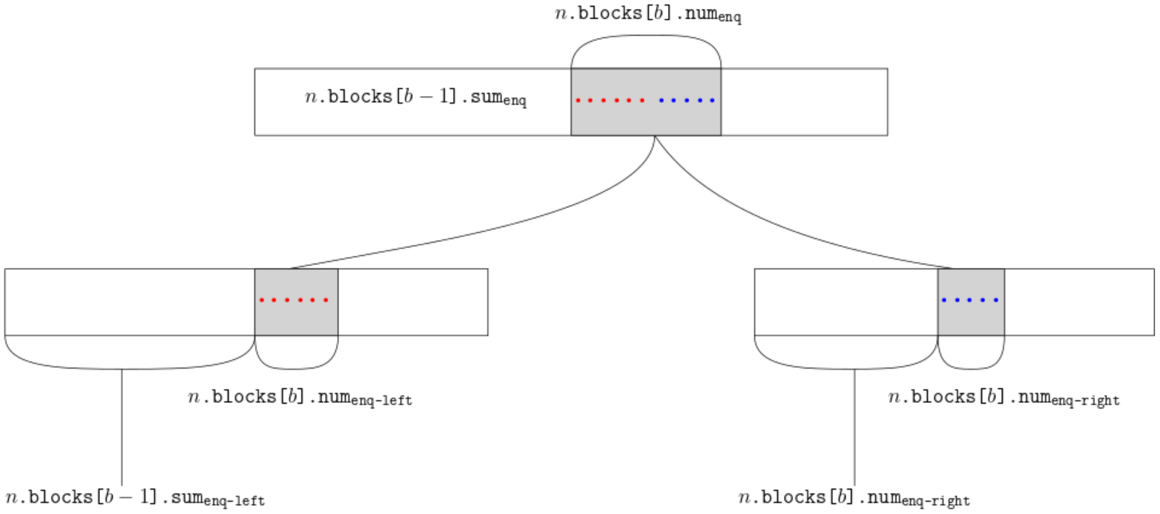
$$n\texttt{.left.blocks[}n\texttt{.blocks[}i\texttt{-1].end}_{\texttt{left}}\texttt{+1..}n\texttt{.blocks[}i\texttt{].end}_{\texttt{left}}\texttt{]} \text{ or}$$

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

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 **??** 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 **??**). 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 **??** or **??**. 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. □



n.blocks[b].num$_{\text{enq}}$
n.blocks[$b-1$].sum$_{\text{enq}}$
n.blocks[b].num$_{\text{enq-left}}$
n.blocks[b].num$_{\text{enq-right}}$
n.blocks[$b-1$].sum$_{\text{enq-left}}$
n.blocks[b].num$_{\text{enq-right}}$

Figure 17: 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).

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

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

Proof. By Lemma **??** and Observation **??**. □

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* **??***.*

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 **??** 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 **??**. By Observation **??** B`.super` was already set in Line **??**. The value written in B`.super` was read in Line **??**,s before the `CAS` that sets B`.super`. From Observation **??** 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 **??** by Invariant **??**. 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* **??***. Then, the block created by one of the next two successful n*`.parent.Refresh`*es according to Definition* **??** *contains B and B*`.super` *is set when the second successful n*`.parent.Refresh` *reaches Line* **??***.*

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 **??** the value R_{p2} reads from n`.head` in Line **??** is at most b. From Observation **??** the value R_{p2} reads in Line **??** 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 **??**) after R_{p1}'s Line **??** and before R_{p2}'s Line **??**. We assumed $t_{\textbf{??}}^{R_n} < t_{\textbf{??}}^{R_{p1}} < t_{\textbf{??}}^{R_{p2}}$ by Definition **??**. Finally, Line **??** is after Line **??** and R_{p2}'s **??** is after R_p''s Line **??**, which is after R_n's n`.blocks.CAS`.

$$\left.\begin{array}{r} {}^{R_n}_{\textbf{??}}t <^{R_{p1}}_{\textbf{??}} t \\ {}^{R_{p1}}_{\textbf{??}}t <^{R_{p'}}_{\textbf{??}} t <^{R_{p2}}_{\textbf{??}} t \\ {}^{R_{p2}}_{\textbf{??}}t <^{R_{p2}}_{\textbf{??}} t \end{array}\right\} \Longrightarrow^{R_n}_{\textbf{??}} t <^{R_{p2}}_{\textbf{??}} t$$

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

Therefore R_{p2} reads n`.head` $= b$. R_{p2} calls n`.Advance` at Line Line **??**, which ensures n`.head` is incremented from b. So the value R_{p2} reads in Line **??** 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 **??**.

□

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 **??** 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`.parent.head`${}_{{}^{R_{p2}}_{\textbf{??}}t} = n$`.parent.head`${}_{{}^{R_{p1}}_{\textbf{??}}t} + 1$ and by Definition **??** and Observation **??**

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

Now using Corollary **??** 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 **??**). For the non-root nodes n.IndexDequeue(b, i) computes sb, the index of the superblock of the bth block in n, in Line **??** by Corollary **??**. After that, the position of $D_i(n, b)$ in $D(n.\texttt{parent}, sb)$ is computed in Lines **??**–**??**. By Definition **??**, Dequeues in a block are ordered based on the order of its subblocks from left to right. If $D_i(n, b)$ was propagated from the left child, the number of dequeus in the left subblocks of n.parent.blocks[sb] before n.blocks[b] is considered in Line **??** (see Figure **??**). 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 **??**) (see Figure **??**). Finally IndexDequeue is called on n.parent recursively and it returns the correct response by induction hypothesis. □

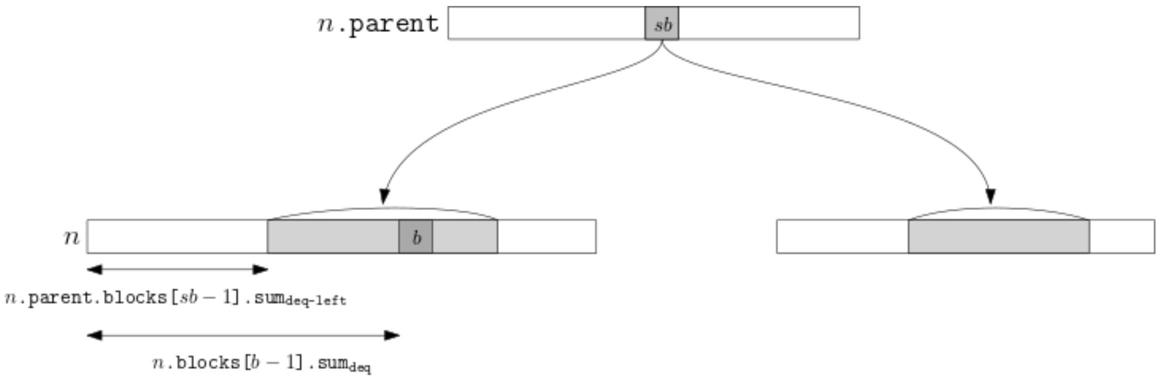


Figure 18: The number of Dequeue operations before $E_i(n, b)$ shown in the case where n is a left child.

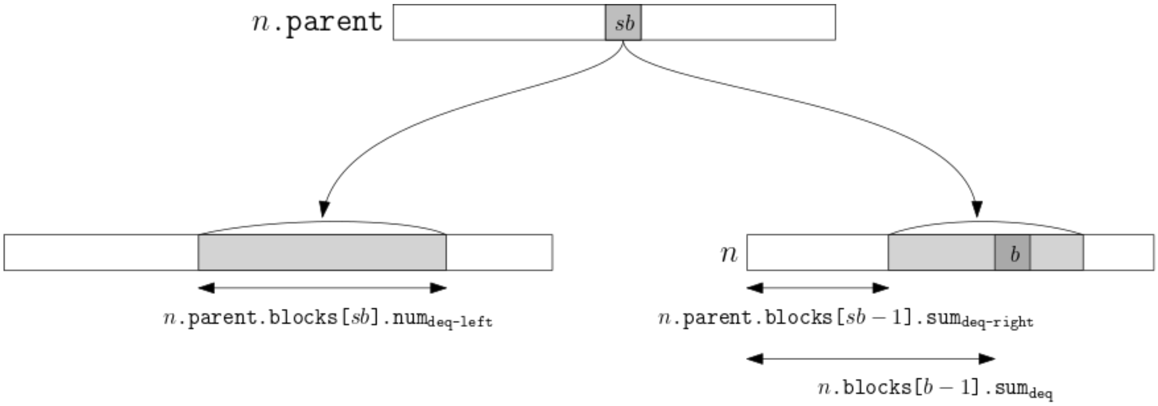

n.parent
sb
n
b
n.parent.blocks[sb].num$_{\texttt{deq-left}}$
n.parent.blocks[$sb-1$].sum$_{\texttt{deq-right}}$
n.blocks[$b-1$].sum$_{\texttt{deq}}$

Figure 19: The number of `Dequeue` operations before $E_i(n,b)$ shown in the case where n is a right child.

5.6 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 **??**. For the latter, if op_1 terminates before op_2 starts, op_1`.Append` has terminated before op_2`.Append` started. From Lemma **??**, 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 **??**).

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

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

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

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

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

Proof. From Corollary **??** and Lemma **??**. □

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

Proof. $D_i(root, b)$ is $D_{\texttt{root.blocks[}b-1\texttt{].sum}_{\texttt{deq}}+i}(root)$ by Definition **??** and Lemma **??**. $D_i(root, b)$ returns `null` at Line **??** if $\texttt{root.blocks[}b-1\texttt{].size} + \texttt{root.blocks[}b\texttt{].num}_{\texttt{enq}} - i < 0$ and *Resp*$(D_i(root, b)) =$ `null` in this case by Lemma **??**. Otherwise, if $D_i(root, b)$ is the eth non-null dequeue in L it should return the eth enqueued value. By Lemma **??** there are $\texttt{root.blocks[}b-1\texttt{].sum}_{\texttt{enq}}$ $-$ $\texttt{root.blocks[}b-1\texttt{].size}$ 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 **??**). See Figure **??**.

After computing e at Line **??**, the code finds b,i such that $E_i(root, b) = E_e(root)$ using DSearch and then finds its element using GetEnqueue (Line **??**). Correctness of DSearch and GetEnqueue routines are shown in Lemmas **??** and **??**. □

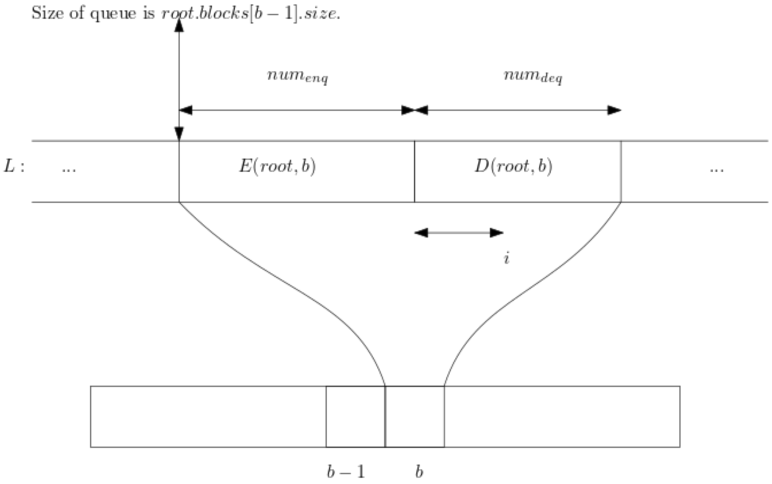


Figure 20: The position of $D_i(root, b)$.

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 **??** 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 **??** and **??**. □

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

6 Analysis

Lemma 50 (Amortized time analysis). `Enqueue` *and* `Dequeue`*, each take* $O(\log^2 p+\log q)$ *steps in amortized analysis. Where* p *is the number of processes and* q *is the size of the queue at the time of invocation of operation.*

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

A `Dequeue` creates a block with null value element, appends it to the tree, computes its order among enqueue operations, and returns the response. The first two part is similar to an `Enqueue` operation. To compute the order of a `dqueue` in $D(n)$ there are some constant steps and `IndexDequeue()` is called. `IndexDequeue` does a search with range p in each level which takes $O(log^2p)$ in the tree. In the `FindResponse()` routine `DSearch()` in the root takes $\Theta($log(`root.blocks[b].size` +`root.blocks[end].size`) by Lemma **??**, which is $O($log size of the queue when `Enqueue` is invoked) + log size of the queue when `Dequeue` is invoked). Each search in `GetEnqueue()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma **??**), so `GetEnqueue()` takes $O(\log^2 p)$ steps.

If we split `DSearch` time cost between the corresponding `Enqueue`, `Dequeue`, in amortized we have `Enqueue` takes $O(\log p + q)$ and `Dequeue` takes $O(\log^2 p + q)$ steps. □

Lemma 51. *An* `Enqueue` *or* `Dequeue` *operation, does at most* $4 \log p$ `CAS` *operations.*

Proof. In each height of the tree at most 2 times `Refresh` is invoked and every `Refresh` invokes at most 3 `CAS`es, one in Line **??** and two from `Advance` in Line **??**. □

Lemma 52 (`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`. Suppose there were more than `root.blocks[b].size` `Dequeue`s in `root.blocks[`$b+1 \cdots end-1$`]`. Then the element in the queue which is the response to the `Dequeue` would become dequeued at some point before `root.blocks[`end`]`'s first `Dequeue`. 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`]`. 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`]`. 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 there cannot be more than `root.blocks[`b`].size`+`root.blocks[`end`].size` `Enqueue`s in `root.blocks[`$b+1\cdots end-1$`]`. Overall there can be at most $2\times$ `root.blocks[`b`].size` + `root.blocks[`end`].size` operations in `root.blocks[`$b+1\cdots end$`]` and since from line **??** 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 **??**. After Line **??**, 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 **??**). So the claim is proved. □

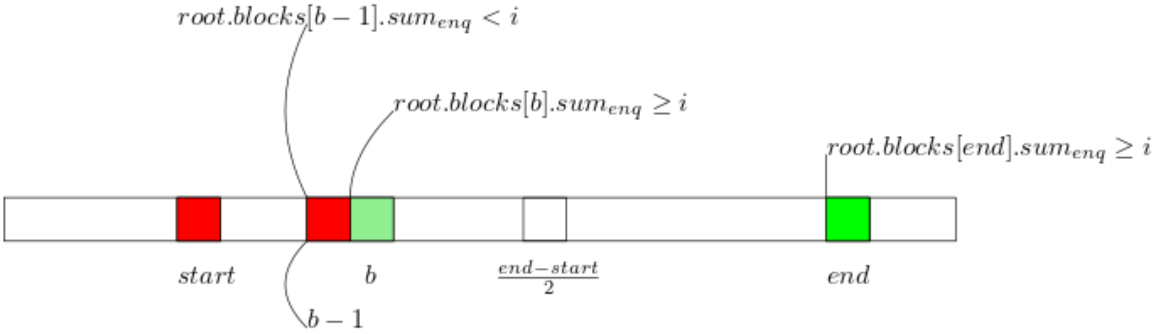

$root.blocks[b-1].sum_{enq} < i$
$root.blocks[b].sum_{enq} \geq i$
$root.blocks[end].sum_{enq} \geq i$
$start$
b
$\frac{end-start}{2}$
end
$b-1$

Figure 21: Distance relations between `start`, b, end.

6.1 Garbage Collection or Getting rid of the infinite Arrays

7 Using Queues to Implement Vectors

Supporting Append, Read, Write in PolyLog time by modifying Get(Enq) Method. Create a Universal Construction Using our vector

8 Conclusion

possible directions for work

Maybe Stacks

Characterize what datastructure can be used for this approach, we already know: queue, fetch & Inc, Vectors

References