Wait-free Queues with Polylogarithmic Step Complexity

Hossein Naderibeni

supervised by Eric Ruppert

October 5, 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 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.

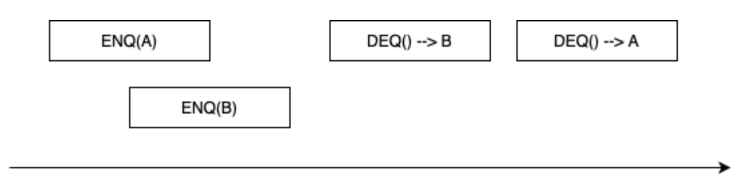


Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

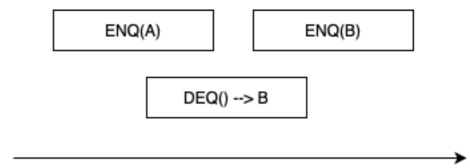


Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A` or nothing.

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.

In section 2 we talk about previous queues and their common problems. We also talk about polylogarithmic construction of shared objects.

Jayanti [9] proved an $\Omega(\log p)$ lower bound on the worst-case shared-access time complexity of p-process universal constructions. He also introduced [1] a construction that achieves $O(\log^2 p)$ shared accesses. Here, we first introduce a universal construction using $O(\log p)$ CAS operations [10]. 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 [13] 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.

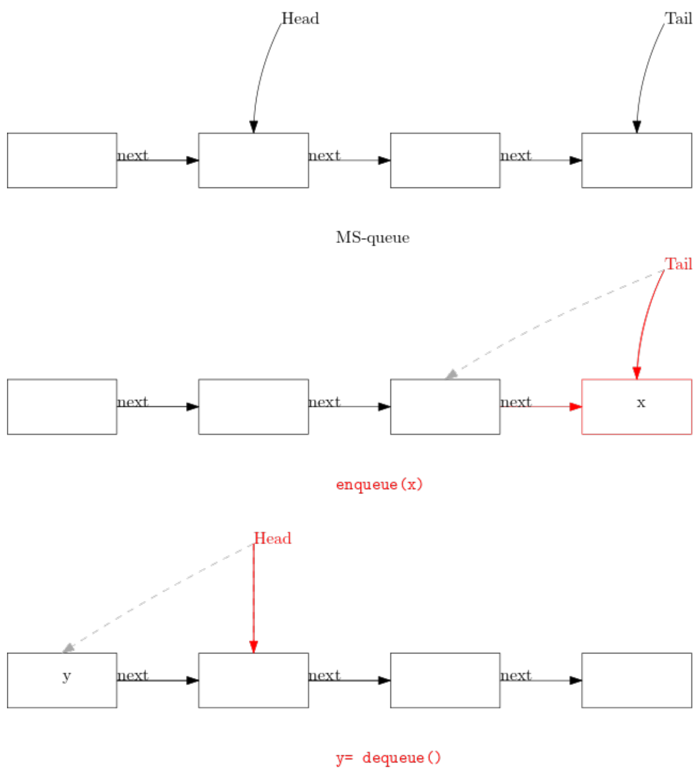


Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

Moir, Nussbaum, and Shalev [14] 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 [8] 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.

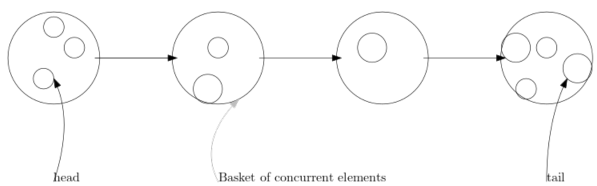


Figure 4: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do `CAS`. To order the operations in a basket, the mechanism in the algorithm for processes is to `CAS` again. The successful process will be the next one in the basket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

Ladan-Mozes and Shavit [12] 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. [6] 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 [4] 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 $\Omega(p)$.

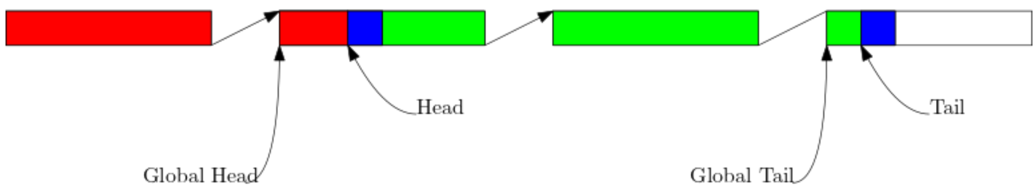


Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

Kogan and Petrank [11] 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* [15]. It is not limited to list-based queues and array-based queues share the CAS retry problem as well [17, 16, 2] . 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 [7]. 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 [9]. He also introduced a construction that achieves $\mathrm{O}(\log^2 p)$ shared accesses [1]. 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 [3]. 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 Our Queue

Jayanti and Petrovic introduced a wait-free polylogarithmic multi-enqueuer single-dequeuer queue [10]. We benefit from some ideas of their work to design a a polylogarithmic multi-enqueuer multi-dequeuer queue. Our algorithm despite them does not use `CAS` operations with big words and does not put a limit on the number of concurrent operations. I our model there are p procesess doing `Enqueue(),Dequeue()` operations concurrently. We use a shared tree among the processes (see Figure 6) to agree on one total ordering on the operations invoked by processes. Each process has a leaf which the order of operations invoked by the process is stored in it. When a process wishes to do an operation it appends the operation to its leaf and then tries to propagate its new operation up to the tree's root. In each node the ordering of operations propagated up to it is stored. All processes agree on the sequence stored in the root and it is defined to be the linearization ordering.

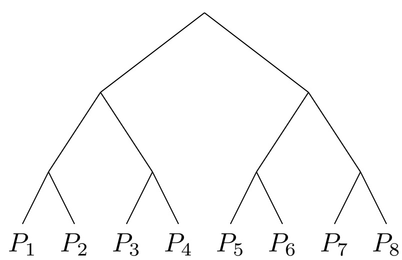
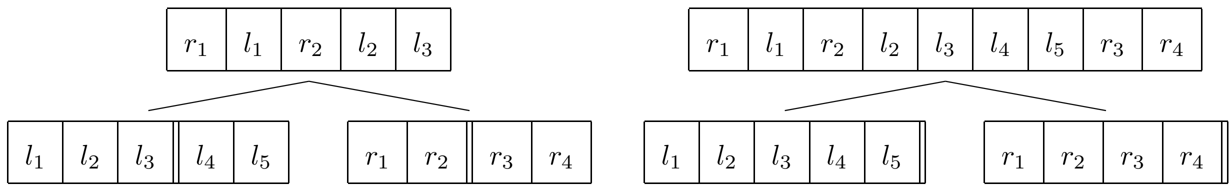


Figure 6: Each process has a leaf and in each node there is an ordering of operations stored. Each node tries to propagate its operations up to the root, which stores the total ordering of all operations.

Add sequence to nodes

We could implement the sequence stored in each node using an array of the queue operations and append some operations to the sequence by doing `k-CAS` operation on the end of the array. To do a propagate step on node n in the tree, we aggregate the operations from node n's both children (that have not already been propagated to n) and try to append them into n. We call this procedure Refresh(n). The main idea is that if we call Refresh(n) twice, the operations in n's children before the first Refresh(n) are guaranteed to be in n. Because if both of the Refresh(n)s fail to do `k-CAS` then there is another instance of `Refresh` in between which has succeeded to do `CAS` and has already appended the operations that the first `Refresh` was trying to append. This mechanism makes us overcome the `CAS` Retry Problem.

fix ||



(a) Operations after || are new. (b) New operations are added to the parent node.

Figure 7: Before and after of a Refresh(n) with successful `CAS`. Operations propagating from the left child are numbered with l and from the right child by r and the operations in children after || are new.

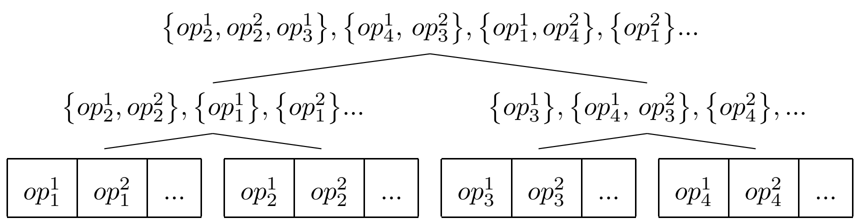


Figure 8: In each internal node, we store the set of all the operations propagated together, and one can arbitrarily linearize the sets of concurrent operations among themselves. Since we linearize operations when they are added to the root, ordering the blocks in the root is important.

The solution for implementing the orderings in the tree told above is not efficient, because there are big `CAS`es and operations information are copied all the way up to the root. Instead of storing operations explicitly in the nodes, we can keep track of some statistics of them. This allows us to `CAS` fixed-size objects in each Refresh(n). To do that, we introduce blocks that only contain the number of operations from the left and the right child in a `Refresh`() procedure and only propagate the statistics block of the new operations. 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. Note that if two operations are in read one 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.

A process may wish to know the ith propagated operation or the rank of a propagated operation in the linearization. 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 its 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 and using the rank of the dequeue among deqeueues and some information stored in the root we compute which enqueue is the answer to the dequeue or if the answer is null. If the answer was some enqueue we find the enqueue using `DSearch(i)` and `GetEnqueue(n,b,i)`. `DSearch(i)` finds the block containing the ith enqueue in the root and `GetEnqueue(n,b,i)` finds its sub-block recursively to reach a leaf. `Index()` is similar but more complicated, finding super-blocks from a leaf to the root. The main challenge in each level of `Get(i)` and `Index(op)` is that it should take polylogarithmic steps with respect to p. After appending operation `op` to the root, processes can find out information about the linearization ordering using `Get(i)` and `Index(op)`. 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.

Implementing Queue using Block Tree In this work, we design a queue with $O(\log^2 p + \log n)$ steps per operation, where n is the number of total operations invoked. We avoid the $\Omega(p)$ worst-case step complexity of existing shared queues based on linked lists or arrays (CAS Retry Problem). 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`. Knowing index i is the tail of the queue, we can return the dequeue response using `Get(i)`. So in the rest we modify block tree to compute `i` for each `Dequeue()` to achieve a FIFO queue.

GetIndex(i) returns the ith operation stored in the block tree sequence. We do that by finding the block b_i containing ith element in the root, and then recursively finding the subblock of b_i which contains ith element. To make this recursive search faster, instead of iterating over all elements in sequence of blocks we store prefix sum of number of elements in the blocks sequence and pointers to make BinarySearch faster.

Furthermore, in each block, we store the prefix sum of left and right elements. Moreover, for each block, we store two pointers to the last left and right subblock of it (see fig 11 and 10).

Starting from the root, GetIndex(i) BinarySearches i in the prefix sum array to find block containing ith operation, then continues recursively calling GetElement(b, i) to find ith element of block b. From lemma 28 we know a block size is at most p. So BinarySearch takes at most (O)($\log p$), since with knowing pointers

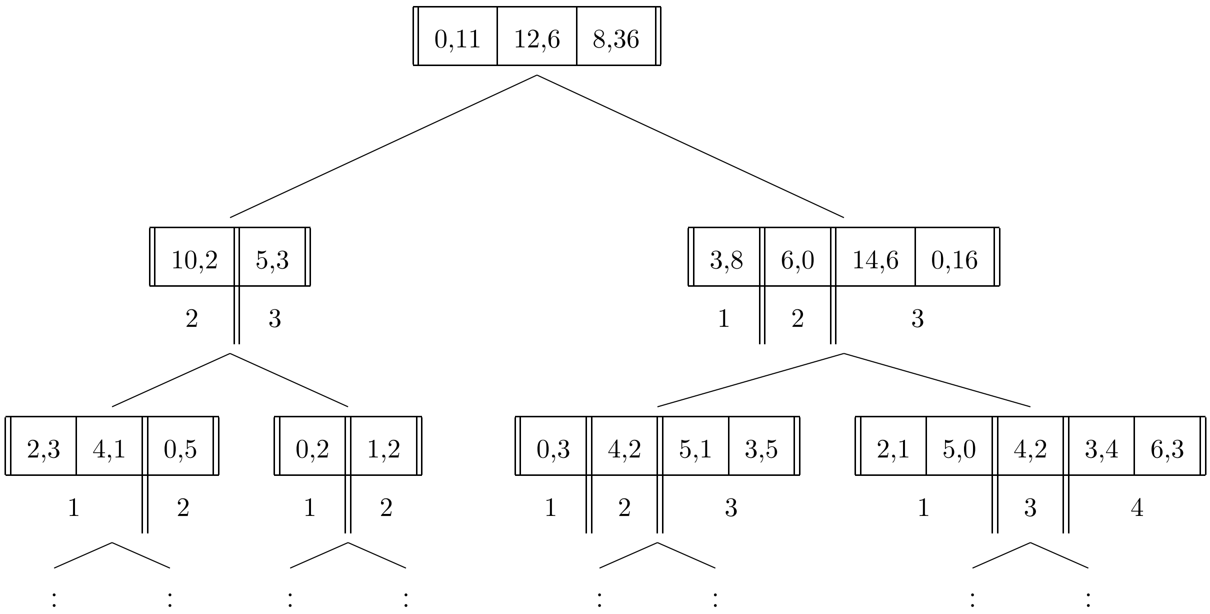


Figure 9: Showing concurrent operation sets with blocks. Each block consists of a pair(left, right) indicating the number of operations from the left and the right child, respectively. Block (12,6) in the root contains blocks (10,2) from the left child and (6,0) from the right child. Blocks between two 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 10: 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.

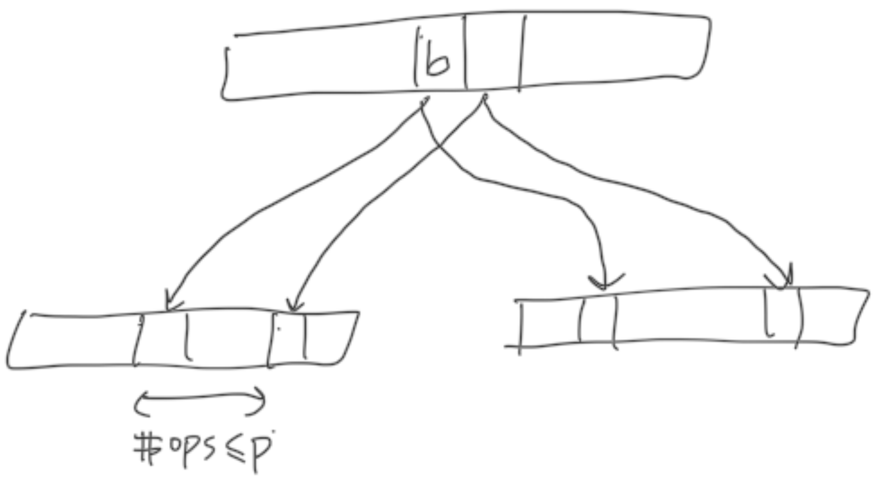


Figure 11: Block have pointers to the starting block of theirs for each child.

of a block and its previous block we can determine the base `(domain ?)` to search and its size is $O(p)$.

`CreateBlock` `CreateBlock(n)` returns a block containing new operations of `n`'s children. $\texttt{b}'.\texttt{end}_{\texttt{left}}$ stores the index of the rightmost subblock of left child of `b`'s previous block. Other attributes are assigned values followed by definition.

Computing `Get(n, b, i)`

How `Refresh(n)` **works.**

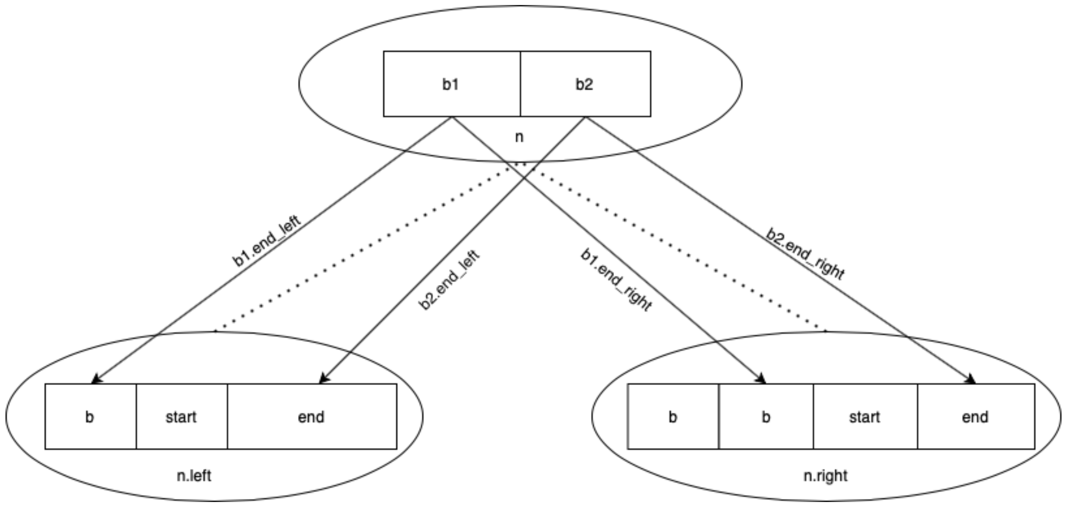


Figure 12: Snapshot of a CreateBlock

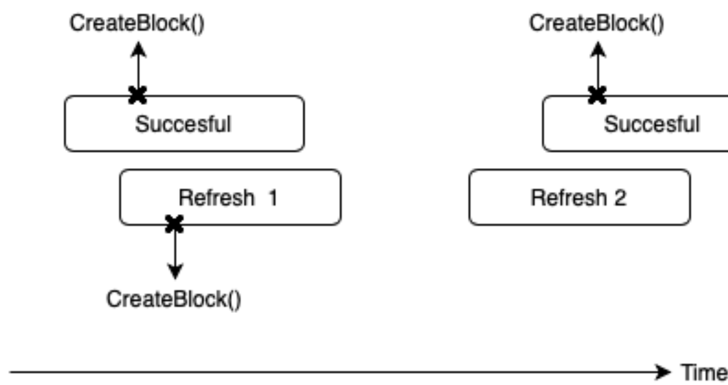


Figure 13: The second failed Refresh is assuredly concurrent to a Successful `Refresh` with `CreateBlock` line after first failed `Refresh`'s `CreateBlock`.

Computing superblock 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 27 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 317. 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 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 n.parent help fill in the value of b.super.

Block b is appended to n.blocks[h] 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 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.

Queue from tree Now, we describe how to use the tree to implement a queue. Consider the following execution of operations. Enqueue(e) appends an operation with input argument e in the block tree. What should a Dequeue() return? To compute the response of a Dequeue(), process p first appends a DEQ operation to the tree. Then p finds the rank of the DEQ using Index(), the rank of the DEQ and the information stored in the root about the queue p computes the rank of the ENQ having the answer of the DEQ. Finally p returns the argument of that ENQ using Get(i).

ENQ(5)	ENQ(2)	DEQ()	ENQ(3)	DEQ()	DEQ()	DEQ()	ENQ(4)	ENQ(6)	DEQ()

Table 1: An example histoy of operations on the queue

A non-null dequeue is one that returns a non-null value. In the example above, Dequeue() operations return 5, 2, 3, null, 4 in order. Before ENQ(4) the queue gets empty so the last DEQ() returns null. 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(). So, in order to answer a Dequeue, it's sufficent to know the size of the queue and the number of previous non-null dequeues.

In the Block Tree, we did not store the sequence of operations explicitly but instead stored blocks of concurrent operations to optimize Propagate steps and increase parallelism. So now the problem is to find the result of each Dequeue. From lemma 28 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. `DEQ()` operations return `null, 5, 2, 1, 3, 4, null` respectively. We will next describe how these values can be computed efficiently.

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

Table 2: An example history of operation blocks on the queue

Now, we claimed that by knowing the current size of the queue and the number of non-null dequeue operations before the current dequeue, we could compute the index of the resulting `Enqueue()`. We apply this approach to blocks; if we store the size of the queue after each block of operations happens and the number of non-null dequeues dequeues till a block, 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()`
#enqueues	0	3	1	1
#dequeues	1	1	1	4
#non-null dequeues	0	1	2	5
size	0	2	2	0

Table 3: Augmented history of operation blocks on the queue

Size and the number of non-null dequeues for bth block could be computed this way:

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

`non-null dequeues[b]= non-null dequeues[b-1] +dequeues[b] -size[b-1] -enqueues[b]`

Given `DEQ` is in block `b`, `response(DEQ)` would be:

```
(size[b-1]- index of DEQ in the block's dequeus >=0) ?  ENQ[non-null dequeus[b-1]+ index of
DEQ in the block's dequeus] :  null;
```

3.1 Pseudocode description

Specification A Queue is a shared data structure that stores a sequence of elements. It has two methods `Enqueue(e)` and `Dequeue()`. `Enqueue(e)` adds `e` to the end of the sequence. `Dequeue()` returns the first

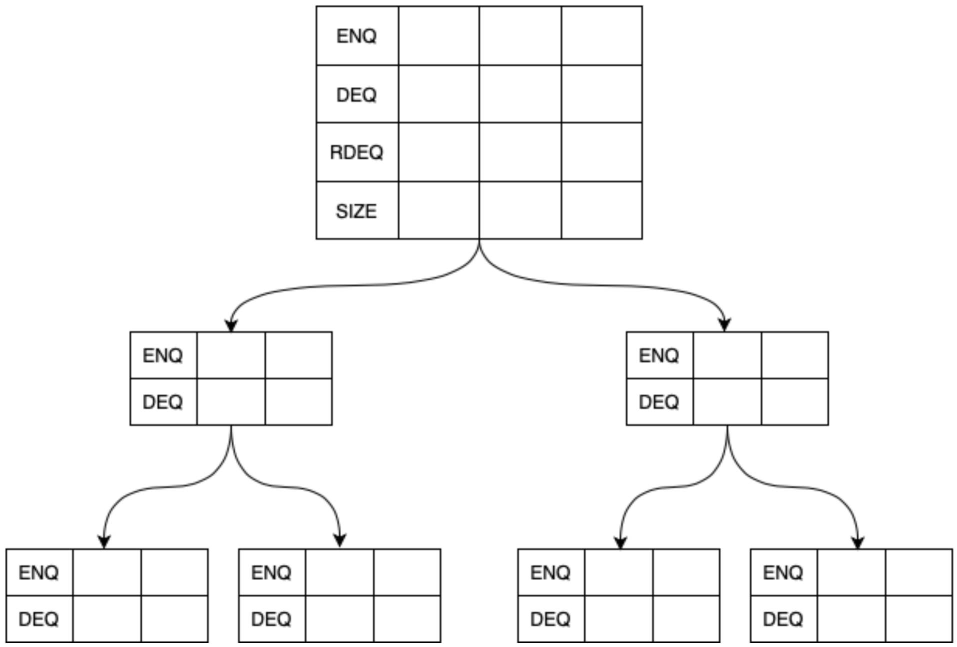


Figure 14: Fields stored in the Queue nodes.

element stored in the sequence and removes it from the sequence.

Tree In order to reach an agreement on the order of operations among p processes, we use a Tournament Tree. Leaf `l_i` is assigned to a process `i`. Each process adds *op* to its leaf. In each internal node an ordering of operations in its subtree is stored. All processes agree on the total ordering of all operations stored in the root. This ordering will be the linearization of the operations.

Implicit Storing Blocks For efficiency, instead of storing explicit sequence of operations in nodes of the Tournament Tree, we use Blocks. A Block is a constant size object that implicitly represents a sequence of operations. In each node there is an array of Blocks.

Block b contains subblocks in the left and right children. WLOG left subblocks of b are some consecutive blocks in the left child starting from where previous block of b has ended to the the end of b. See Figure 12 .

We store ordering among `operation`s in the tournament tree constructed by `node`s. In each `node` we store pointers to its relatives, an array of `block`s and an index to the first empty `block`. 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 `block`s 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].

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) just appends an operation with element e to the root. Dequeue() appends an operation to the root and computes its ordering and the enqueue operation containing the head before it calling ComputeHead() and then gets and returns the operation's element.

Append(op) adds op to the invoking process's leaf's ops and blocks, propagates it up to the root and if the op is a dequeue returns its order in residing block in the root and the block's index. As we said later Propagate assuredly aggregates new blocks to a block in the parent by calling Refresh two times. Refresh(n) creates a block, tries to CAS it into the pn's blocks and if it was successful updates super and counter in both of n's children.

We only want to know the element of enqueue operations and compute ordering for dequeue operations. That's the reason here Get() searches between enqueues only and Index() returns ordering of a dequeue among dequeues. Get(n, b ,i) decides the requested element is in which child of n and continues to search recursively. index(n, i, b) calculates the ordering of the given operation in n's parent each step and finally returns the result among total ordering.

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

▶ *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

▶ *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* $\texttt{sum}_{\texttt{enq-left}}$: # enqueue operations in the prefix for $\texttt{left.blocks[end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-left}}$: # dequeue operations in the prefix for $\texttt{left.blocks[end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: # enqueue operations in the prefix for $\texttt{right.blocks[end}_{\texttt{right}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-right}}$: # dequeue operations in the prefix for $\texttt{right.blocks[end}_{\texttt{right}}\texttt{]}$

▶ *RootBlock* extends *InternalBlock*

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

Abbreviations:

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

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, computes its order among operations, 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 the response to the 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 ≥ e
<int, int> DSearch(int e, int end)                ▷ Returns <b,i> if E_e(root) = E_i(root, b).
    start= end-1
    while root.blocks[start].sum_enq≥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)                ▷ Append is 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()
    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, hp)
    head.CAS(h, h+1)
end Advance

⇝ Precondition: blocks[start..end] contains a block with field f ≥ i
int BinarySearch(field f, int i, int start, int end)
    ▷ Does 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 ≥ i.
end BinarySearch

<Block, int, int> CreateBlock(int i)   ▷ Creates and returns the block to be inserted as ith block in blocks.
    block newBlock= new(block)
    for each dir in {left, right} do
        index_last= dir.head-1
        index_prev= blocks[i-1].end_dir
        newBlock.end_dir= index_last
        block_last= dir.blocks[index_last]
        block_prev= dir.blocks[index_prev]
                                        ▷ newBlock includes dir.blocks[index_prev+1..index_last].
        newBlock.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq - block_prev.sum_enq
        newBlock.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq - block_prev.sum_deq
    end for
    if this is root then
        newBlock.size = max(root.blocks[i-1].size + newBlock.num_enq - newBlock.num_deq, 0)
    end if
    return <b, np_left, np_right>
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(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 ≥ 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
        end if
        if dir is right then
            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* $n\texttt{.blocks[}i\texttt{].end}_{\texttt{left}} \geq n\texttt{.blocks[}i-1\texttt{].end}_{\texttt{left}}$ *and* $n\texttt{.blocks[}i\texttt{].end}_{\texttt{right}} \geq n\texttt{.blocks[}i-1\texttt{].end}_{\texttt{right}}$.

Proof. Consider the block b written into `n.blocks[`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 334 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 $n.\texttt{blocks[}i\texttt{].end}_{\texttt{left}} \leq n.\texttt{blocks[}j-1\texttt{].end}_{\texttt{left}}$, so the ranges $[n.\texttt{blocks[}i-1\texttt{].end}_{\texttt{left}}+1 \cdots n.\texttt{blocks[}i\texttt{].end}_{\texttt{left}}]$ and $[n.\texttt{blocks[}j-1\texttt{].end}_{\texttt{left}}+1 \cdots n.\texttt{blocks[}j\texttt{].end}_{\texttt{left}}]$ 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

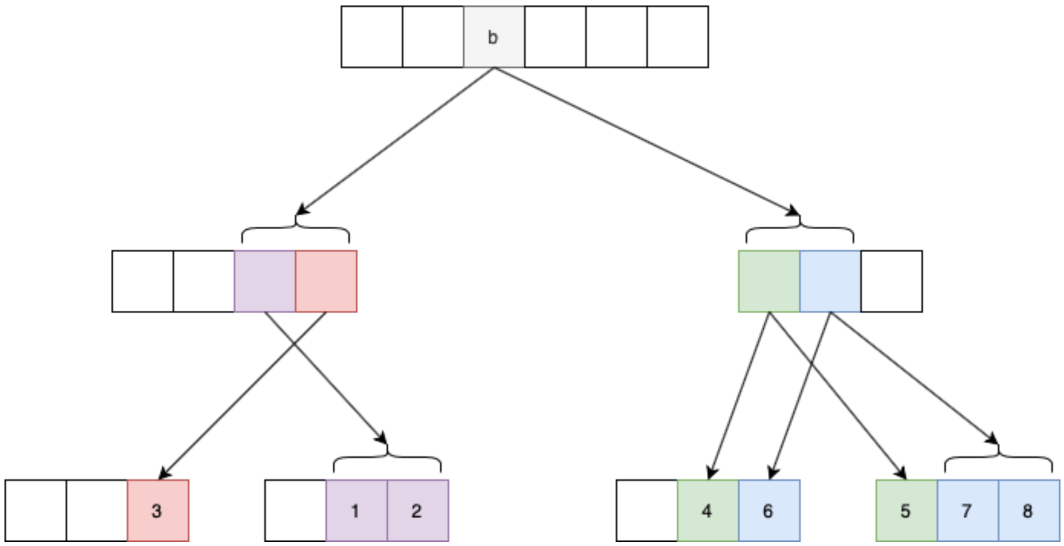


Figure 15: Order of operations in b. Operations in the leaves are ordered with numerical order shown in the drawing.

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 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

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

- $E_i(n, b)$ 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 $D(n)$.

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

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

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

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

(3) $B\texttt{.num}_{\texttt{x}} = \left|E(n, b)\right|$

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 claim for n.

$$\begin{aligned}B\texttt{.num}_{\texttt{x-left}} =& B\texttt{.sum}_{\texttt{x-left}} - B'\texttt{.sum}_{\texttt{x-left}} && \text{Definition of } \texttt{num}_{\texttt{x}}\\ =& B'\texttt{.sum}_{\texttt{x-left}} + n\texttt{.left.blocks[}B\texttt{.end}_{\texttt{left}}\texttt{].sum}_{\texttt{x}}\\ & - n\texttt{.left.blocks[}B'\texttt{.end}_{\texttt{left}}\texttt{].sum}_{\texttt{x}} - B'\texttt{.sum}_{\texttt{x-left}} && \texttt{CreateBlock}\\ =& n\texttt{.left.blocks[}B\texttt{.end}_{\texttt{left}}\texttt{].sum}_{\texttt{x}} - n\texttt{.left.blocks[}B'\texttt{.end}_{\texttt{left}}\texttt{].sum}_{\texttt{x}}\\ =& \sum_{i=B'\texttt{.end}_{\texttt{left}}+1}^{B\texttt{.end}_{\texttt{left}}} n\texttt{.left.blocks[}i\texttt{].num}_{\texttt{x}} && \text{Observation17}\end{aligned}$$

The last line holds because of the induction hypothesis. (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 $\left|E(n,b)\right| = B\texttt{.num}_{\texttt{x-left}} + B\texttt{.num}_{\texttt{x-right}} = B\texttt{.num}_{\texttt{x}}$ $\square$

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

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

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

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

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 $\texttt{end}_{\texttt{left}} = n.\texttt{left.head}_{334} - 1$. Therefore, by Definition 5, after the successful `CAS` in Line 320 we know all blocks in $n.\texttt{left.blocks[}1 \cdots n.\texttt{left.head}_{334} - 1\texttt{]}$ are subblocks of $\texttt{n.blocks[}1 \cdots n.\texttt{head}_{310}\texttt{]}$. Because of Lemma 2 we have $n.\texttt{left.head}_{309} - 1 < n.\texttt{left.head}_{334} - 1$ 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 $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq EST^{^{R_2}t}_{n}$.*

Proof. Let R_1 read i from n.`head` at Line 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 $^{R_1}t < t^{R_1}_{310} < t^{R2'}_{310}$ (see Figure 16). By Lemma 23 we have $EST^{n.\texttt{left}}_{^{R_2'}_{310}t} \cup EST^{n.\texttt{right}}_{^{R_2'}_{310}t} \subseteq ops(n.\texttt{blocks}_{t^{R_2'}_{320}})$. 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` ($t^{R_2'}_{310}$) and R_2''s successful `CAS` ($t^{R_2'}_{320}$) is before R_2's termination (t^{R_2}), so by Observation and Lemma 3 we have 13 $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{t^{R_2'}_{320}}) \subseteq EST^{^{R_2}t}_{n.\texttt{left}}$.

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 $t^{R_1} <^{R_3}_{310} t <^{R_3}_{320} t < t^{R_2}_{310} <^{R}_{2} t$. From Observation 13 on the operations before and after R_3's `CAS` and Lemmas 22, 3 on R_3 the claim holds. □

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

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`.

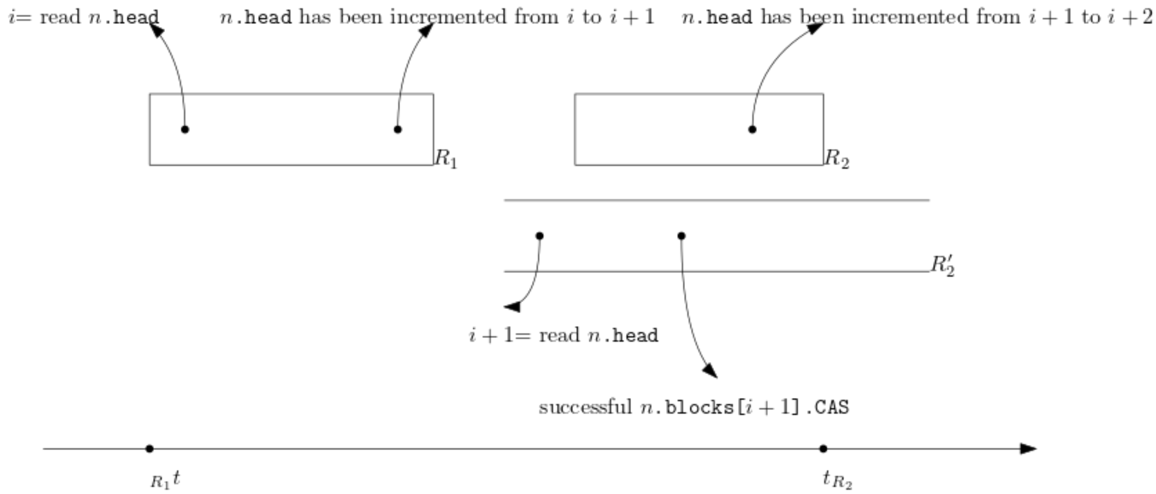


Figure 16: ${}_{R_1}t < t_{310}^{R_1} <$ incrementing n.`head` from i to $i+1 < t_{310}^{R_2'} < t_{320}^{R_2'} <$ incrementing n.`head` from $i+1$ to $i+2 < t_{R_2}$

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`. $\square$

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. $\square$

4.4 Correctness of GetEnqueue

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

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

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process p in block b in node n. Without loss of generality op_1 is invoked earlier than op_2. Process p cannot invoke more than one operation concurrently, so op_1 has to be finished before op_2 begins. By Corollary 27 before op_2 calls `Append`, op_1 exists in every node of the tree on the path from p's leaf to the root. Since b contains op_2, it must be created after op_2 is invoked. 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`>` so 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. `DSearch` performs a doubling search from `root.blocks[end]` to `root.blocks[0]` to find $E_e(root)$. From Lines 340, 341 we know the `sum`$_{\texttt{enq-left}}$ and `sum`$_{\texttt{enq-right}}$ fields of `blocks` in each node are sorted in non-decreasing order. Since `sum`$_{\texttt{enq}}$ $=$ `sum`$_{\texttt{enq-left}}$ $+$ `sum`$_{\texttt{enq-right}}$, the `sum`$_{\texttt{enq}}$ values of `root.blocks[`$0 \cdot\cdot end$`]` is 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 $E_i(root, b)$. The doubling search on Lines 802–805 doubles its 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`$_{\texttt{enq}}$. In Line 806, the binary search finds the b required in the range mentioned. Finally i, is computed from the definition of `sum`$_{\texttt{enq}}$ and lemma 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 $E_i(n.child, b')$ then n`.GetEnqueue(`b`, `i`)` returns $E_i(n, b)$. From Definition 15 of $E(n, b)$, operations from the left subblocks come before the operations from the right subblocks in a block (see Figure 17). 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 `i`th enqueue in bth block of internal node n is in the left child or right child subblocks of n`.blocks[b]`. By Definitions 10 and 5 to find an operation in the subblocks of n`.blocks[`i`]` we need to search in the range

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

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

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 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. ☐

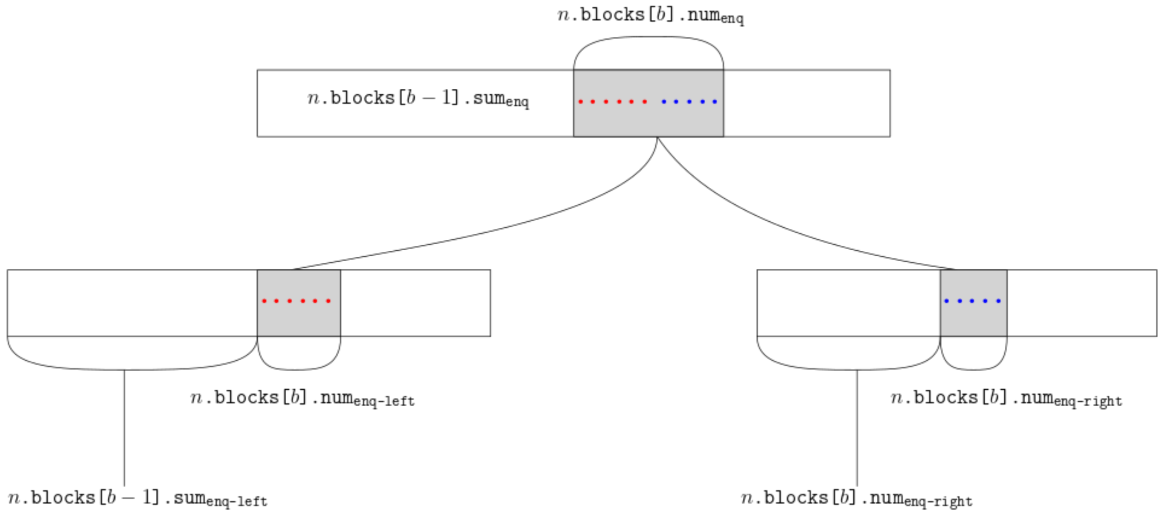


Figure 17: The number and ordering of the enqueue operations propagated from the left and the right child to `n.blocks[b]`. `n.blocks[b]` and its subblocks are shown by gray color. Enqueue operations from the left subblocks (colored red), are ordered before the enqueue operations from the right child (colored blue).

4.5 Correctness of IndexDequeue

Next few results show that the `super` field of a block is accurate within one of actual index of the block's superblock in the parent node. Next 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* $i+1$ *untill* B`.super` *is set.*

Proof. From Observation 2 we know the n`.head` ichanges only by 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 the putting B into 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* n`.head` *greater than* b *in Line 334.*

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

Proof. By 35 if R_p propagates B it has to read a greater value than b for n`.head`, which means n`.head` was incremented in Line 327. By Observation 33 B`.super` was already set in Line 326, so if R_p propagates B it means B`.super` was already set. The value written in B`.super` was read in Line 310 or Line 325, which are both before calling the `Advance` that sets B`.super`. From Observation 2 we know n`.parent.head` is non-decreasing so B`.super` $\leq sb$. 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.blocks[`b`.super]` afterwards. □

Lemma 37. *Let* R_n *be the* `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 after Line 314 of the the second successful* n`.parent.Refresh`*.*

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 neither 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 334 is $\leq b$. From Observation 2 the value R_{p2} reads in Line 312 is also $\leq b$.

R_n puts B into n.blocks[b] so R_n reads value b for 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 320 and before R_{p2}'s 310. We assumed $t_{320}^{R_n} < t_{320}^{R_{p1}} < t_{320}^{R_{p2}}$ by Definition 32. Finally, since 312 is after 310 and R_{p2}'s 310 is after R'_p's 327, which is after R_n's n.blocks.CAS. So R_{p2} reads a value $\geq b$ for n.head by Lemma 2.

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

So R_{p2} reads n.head $= b$. After Line 314 of R_{p2}, n.head is incremented from b. So the value R_{p2} reads in Line 334 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} in Line 326 invoked from 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. After that B is installed n.parent.head is read and B.super field is set to the value read from the parent's head (see Lines 325 and 326 ofAdvance). From previous Lemma we know that B is propagated by second next successful Refresh's CAS on n.parent.blocks. To summarize we have $n.\mathtt{parent.head}_{{}^{R_{p2}}_{320}t} = n.\mathtt{parent.head}_{{}^{R_{p1}}_{320}t} + 1$ and $n.\mathtt{parent.head}_{{}^{R_{p1}}_{320}t} \leq n.\mathtt{parent.head}_{{}^{R_n}_{320}t}$, so the value read for n.parent.head 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* $1 \leq i \leq n.\mathtt{blocks}[b].\mathtt{num_{deq}}$ *then* n.IndexDequeue(b,i) *returns* $< x, y >$ *such that* $D_i(n, b) = D_y(\mathtt{root}, x)$.

Proof. We will prove this by induction on the distance of n from the `root`. The base case where n is `root` is trivial (Line 415). For the non-root nodes n`.IndexDequeue(`b`, `i`)` computes sb, 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 419–425. 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 18). Else 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 422) (see Figure 19). Finally `IndexDequeue` is called on n`.parent` recursively and it returns the response from induction hypothesis. ☐

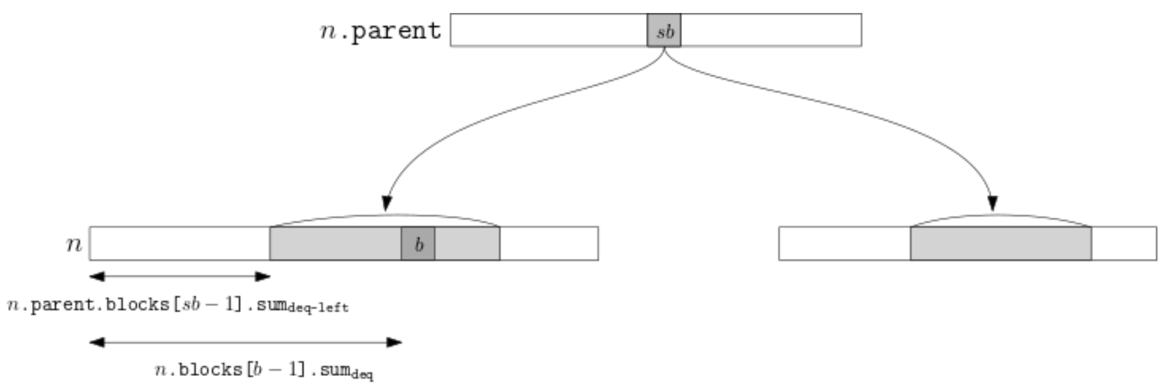


Figure 18: The number of `dequeue` operations before $E_i(n,b)$ shown in the case where n is a left child.

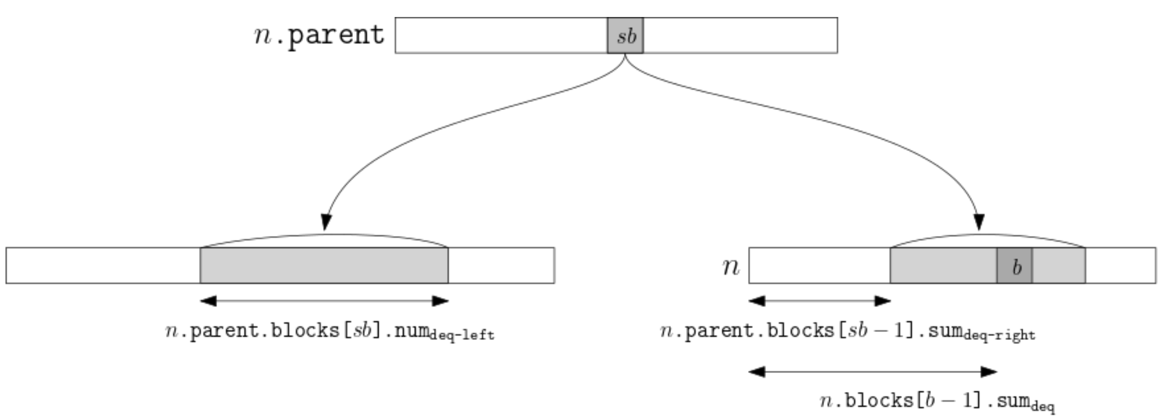


Figure 19: The number of `dequeue` operations before $E_i(n,b)$ shown in the case where n is a right child.

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 one 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 `dequeue`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 if there are more than `root.blocks[`$i-1$`].size` $+$ `root.blocks[`i`].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`].num`$_{\texttt{enq}}$ $-$ `root.blocks[`b`].num`$_{\texttt{deq}}$. In both cases, this is same as the assignment on Line 344. □

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

Proof. There are `root.blocks[`b`].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* ***dequeues*** in `root.blocks[`$0\cdots b$`]`. By the correctness of the `size` field from Lemma 43 and `sum`$_{\texttt{enq}}$ field from Lemma 18, the number of *non-null* ***dequeues*** is `root.blocks[`b`].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`].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`].num`$_{\texttt{enq}}$$-i$ < 0.

Proof. From Corollary 45. ☐

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`].num`$_{\texttt{enq}}$$-i < 0$ and $Resp(D_i(root, b)) =$ `null` in this case by Lemma 46. Otherwise, if $D_i(root, b)$ is the eth non-null dequeue in L it should return the eth enqueued value. By Lemma 44 there are `root.blocks[`$b-1$`].sum`$_{\texttt{enq}}$ $-$ `root.blocks[`$b-1$`].size` non-null `dequeue` operations in `root.blocks[`$0\cdots b-1$`]`. The `dequeues` in `root.blocks[`b`]` before $D_i(root, b)$ are non-null dequeues. So $D_i(root, b)$ is the eth non-null `dequeue` where $e = i +$ `root.blocks[`$b-1$`].sum`$_{\texttt{deq}}$ $-$ `root.blocks[`$b-1$`].size` (Line 222). See Figure 20.

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). ☐

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.*

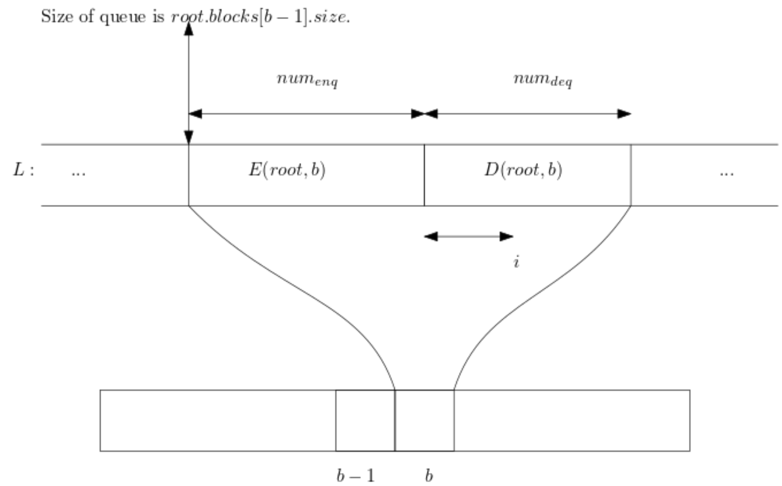


Figure 20: The position of $D_i(root, b)$.

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

Remark In fact our algorithm is strongly linearizable defined in [5]. By Definition 15 the linearization ordering of operations will not change as blocks containing new operations are appended to the root.

5 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 302, 303). 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 $O(log^2 p)$ in the tree. In the `FindResponse()` routine `DSearch()` in the root takes $\Theta(\log($`root.blocks[b].size +root.blocks[end].size`) by Lemma 30, 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 29), 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 320 and two from `Advance` in Line 327. □

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

$\texttt{root.blocks[}end\texttt{]}$'s first $\texttt{Dequeue()}$. Furthermore in the execution of queue operations in the linearization ordering, the size of the queue becomes $\texttt{root.blocks[}end\texttt{].size}$ after the operations of $\texttt{root.blocks[}end\texttt{]}$. There can be at most $\texttt{root.blocks[}b\texttt{].size}$ $\texttt{Dequeue}$s in $\texttt{root.blocks[}b+1\cdots end-1\texttt{]}$; otherwise all elements enqueued by $\texttt{root.blocks[}b\texttt{]}$ would be dequeued before $\texttt{root.blocks[}end\texttt{]}$. The final size of the queue after $\texttt{root.blocks[}1\cdots end\texttt{]}$ is $\texttt{root.blocks[}end\texttt{].size}$. 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 $\texttt{root.blocks[}b+1\cdots end-1\texttt{]}$ is less than $\texttt{root.blocks[}b\texttt{].size}$, therefore there cannot be more than $\texttt{root.blocks[}b\texttt{].size}+\texttt{root.blocks[}end\texttt{].size}$ $\texttt{Enqueue}$s in $\texttt{root.blocks[}b+1\cdots end-1\texttt{]}$. Overall there can be at most $2\times\texttt{root.blocks[}b\texttt{].size}+\texttt{root.blocks[}end\texttt{].size}$ operations in $\texttt{root.blocks[}b+1\cdots end\texttt{]}$ and since from line 318 we know that $\texttt{num}$ field of the every block in the tree is greater than 0, each block has at least one operation, there are at most $2\times\texttt{root.blocks[}b\texttt{].size}+\texttt{root.blocks[}end\texttt{].size}$ blocks in between $\texttt{root.blocks[}b\texttt{]}$ and $\texttt{root.blocks[}end\texttt{]}$. So $end-b-1 \leq 2\times\texttt{root.blocks[}b\texttt{].size}+\texttt{root.blocks[}end\texttt{].size}$.

So the doubling search reaches $\texttt{start}$ such that the $\texttt{root.blocks[start].sum}_{\texttt{enq}}$ is less than e in $O\big(\log(\texttt{root.blocks[}b\texttt{].size}+\texttt{root.blocks[}end\texttt{].size})\big)$ steps. See Figure 21. After Line 805, the binary search that finds b also takes $O\big(\log(\texttt{root.blocks[}b\texttt{].size}+\texttt{root.blocks[}end\texttt{].size})\big)$. Next, $\texttt{i}$ is computed via the definition of $\texttt{sum}_{\texttt{enq}}$ in constant time (Line 807). So the claim is proved. $\square$

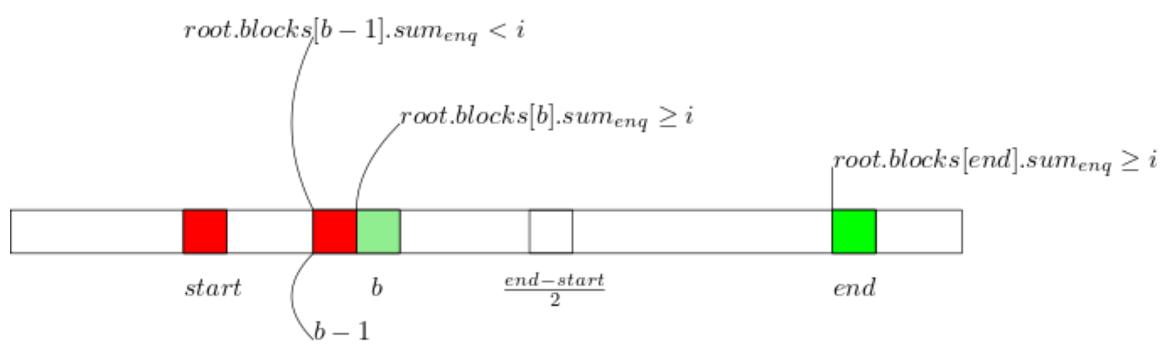


Figure 21: Distance relations between $\texttt{start}, b, end$.

5.1 Garbage Collection or Getting rid of the infinite Arrays

6 Using Queues to Implement Vectors

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

7 Conclusion

possible directions for work

Maybe Stacks

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

References

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.