Wait-free Queues with Polylogarithmic Step Complexity

Hossein Naderibeni

supervised by Eric Ruppert

September 26, 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.

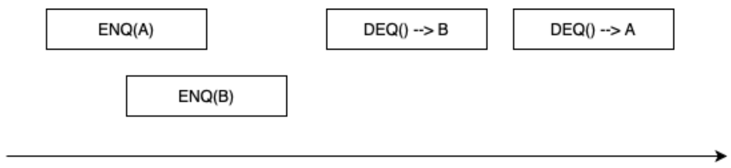


Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

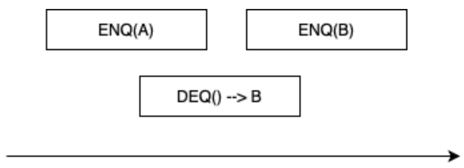


Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A` or nothing.

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 [?] 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 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.

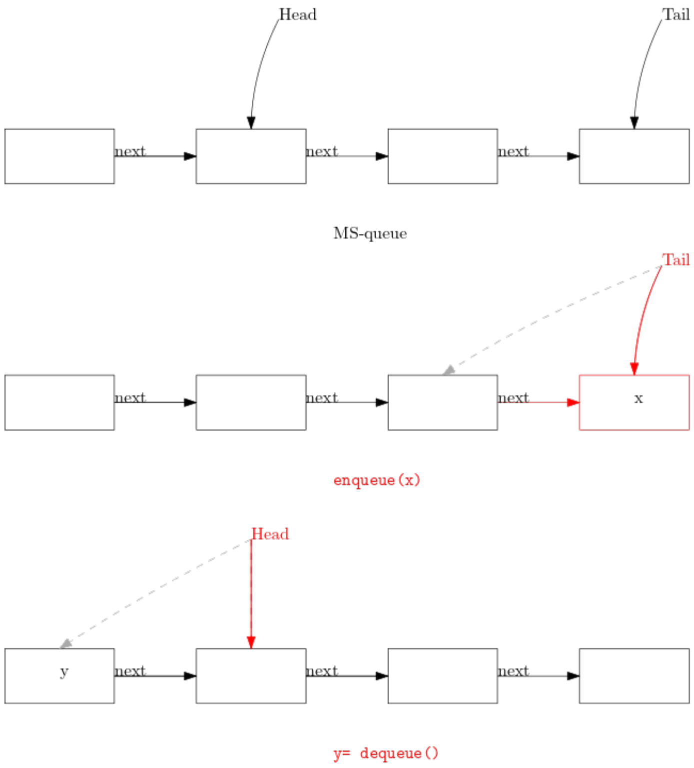


Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

Moir, Nussbaum, and Shalev [?] 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.

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 $\Omega(p)$ `CAS`es.

Hendler et al. [?] proposed a new paradigm called flat combining. Their queue is linearizable but not lock-free. Their main idea is

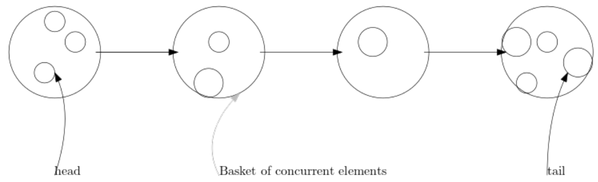


Figure 4: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do CAS. To order the operations in a basket, the mechanism in the algorithm for processes is to CAS again. The successful process will be the next one in the basket and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

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 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 $\Omega(p)$.

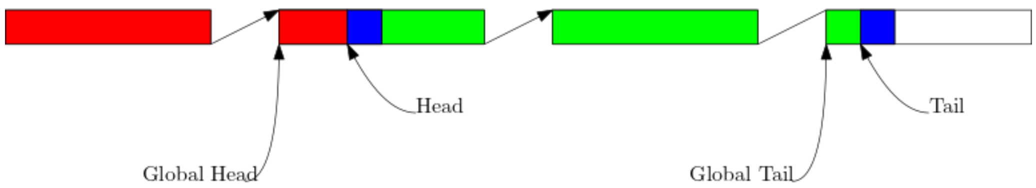


Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

Kogan and Petrank [?] 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 $O(\log^2 p)$ 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 Our Queue

Jayanti and Petrovic introduced a wait-free polylogarithmic multi-enqueuer single-dequeuer queue [?]. 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 processess 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.

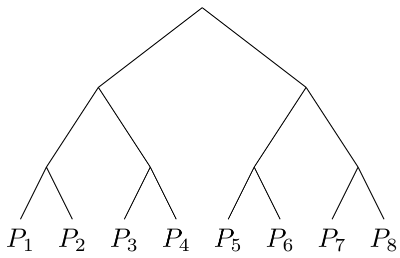
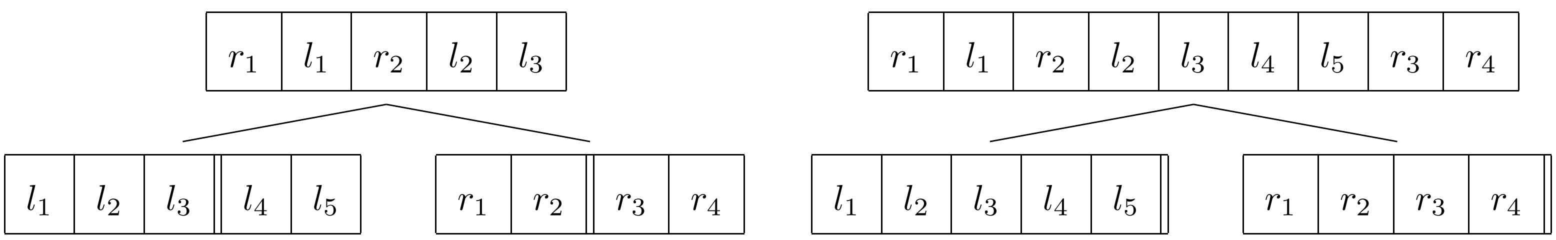


Figure 6: Each process has a leaf and in each node there is an ordering of operations stored. Each node tries to propagate its operations up to the root, which stores the total ordering of all operations.

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.



(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.

fix ||

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

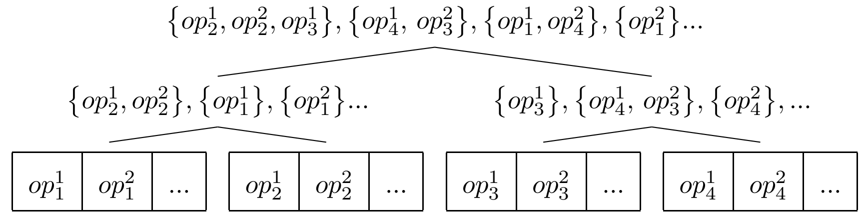


Figure 8: In each internal node, we store the set of all the operations propagated together, and one can arbitrarily linearize the sets of concurrent operations among themselves. Since we linearize operations when they are added to the root, ordering the blocks in the root is important.

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 `GetENQ(n,b,i)`. `DSearch(i)` finds the block containing the ith enqueue in the root and `GetENQ(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 **??** we know a block size is at most p. So BinarySearch takes at most (O)($\log p$), since with knowing pointers of a block and its previous block we can determine the base `(domain ?)` to search and its size is O(p).

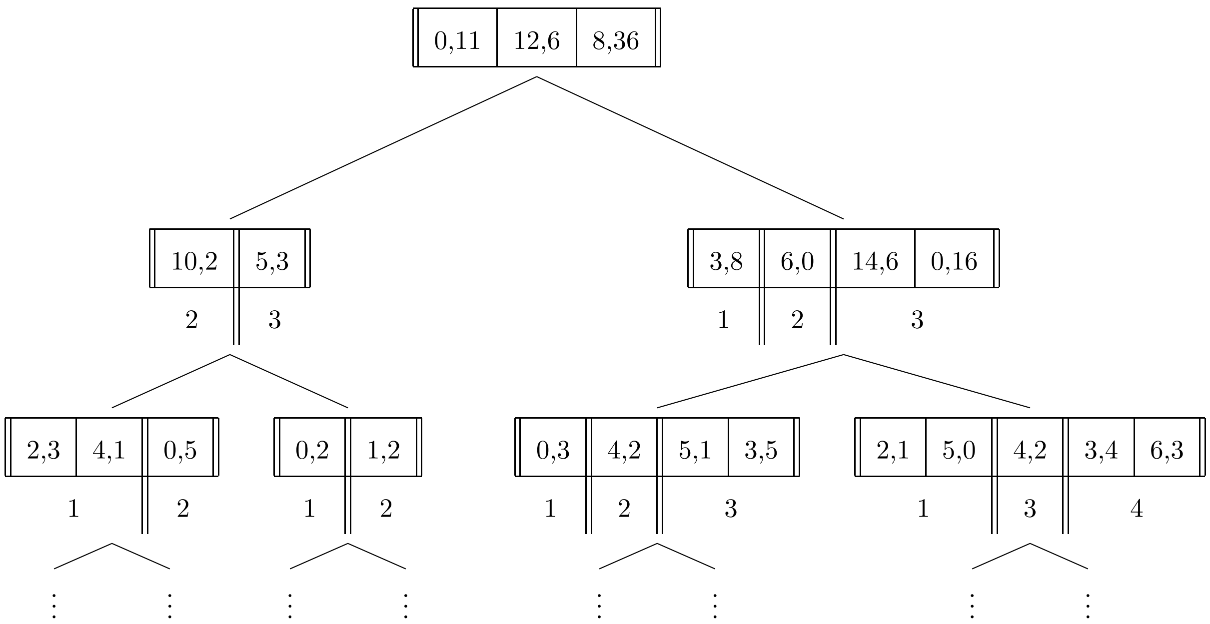


Figure 9: Showing concurrent operation sets with blocks. Each block consists of a pair(left, right) indicating the number of operations from the left and the right child, respectively. Block (12,6) in the root contains blocks (10,2) from the left child and (6,0) from the right child. Blocks between two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

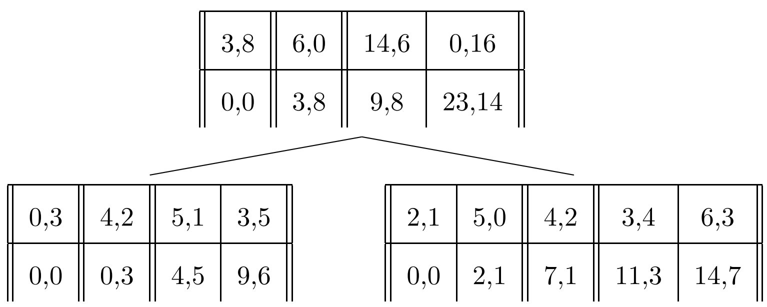


Figure 10: Using Prefix sums in blocks. When we want to find block b elements in its children, we can use binary search. The number below each block shows the count of elements in the previous blocks.

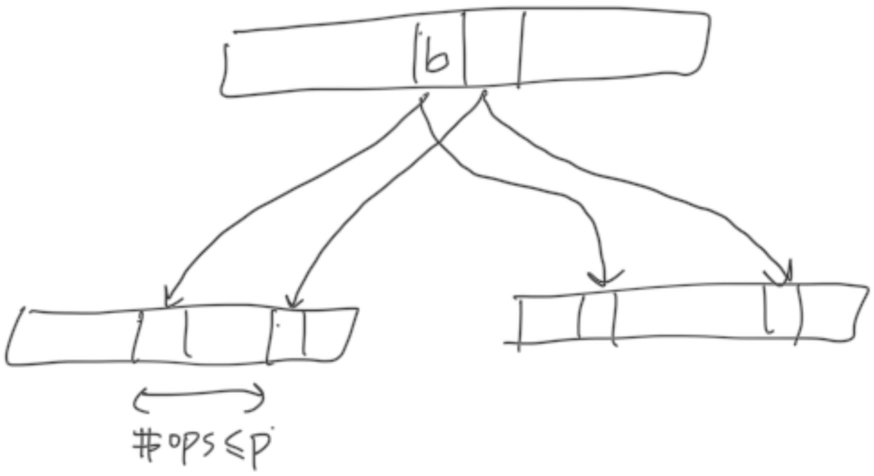


Figure 11: Block have pointers to the starting block of theirs for each child.

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.

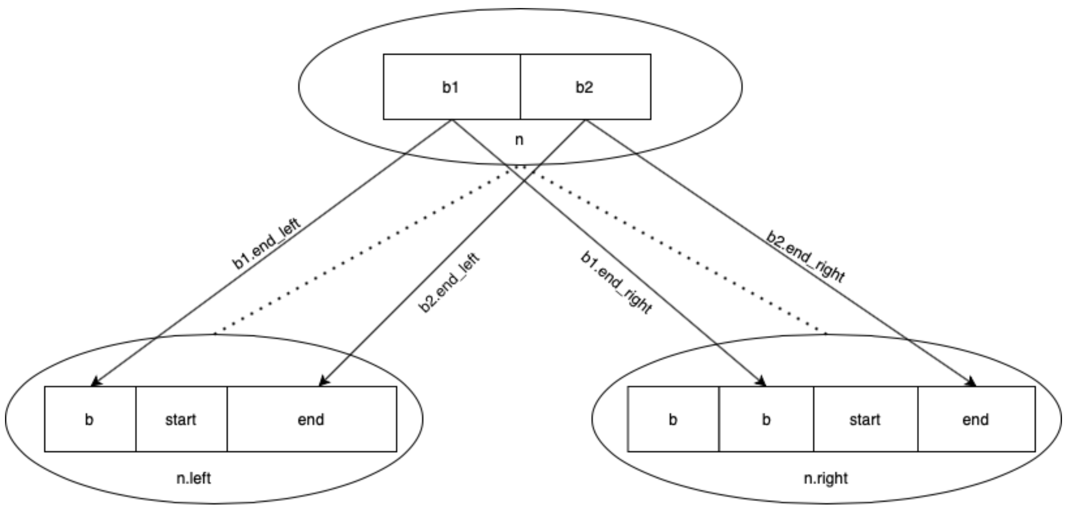


Figure 12: Snapshot of a CreateBlock()

Computing `Get(n, b, i)`

How `Refresh(n)` works.

Computing superblock

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

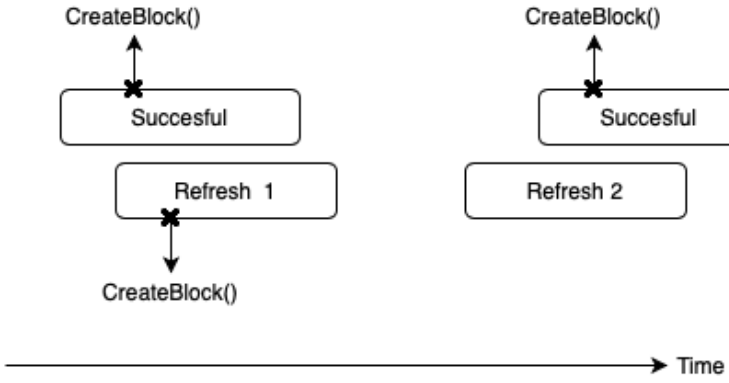


Figure 13: The second failed Refresh is assuredly concurrent to a Successful `Refresh()` with `CreateBlock` line after first failed `Refresh`'s `CreateBlock()`.

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 ?? 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;`

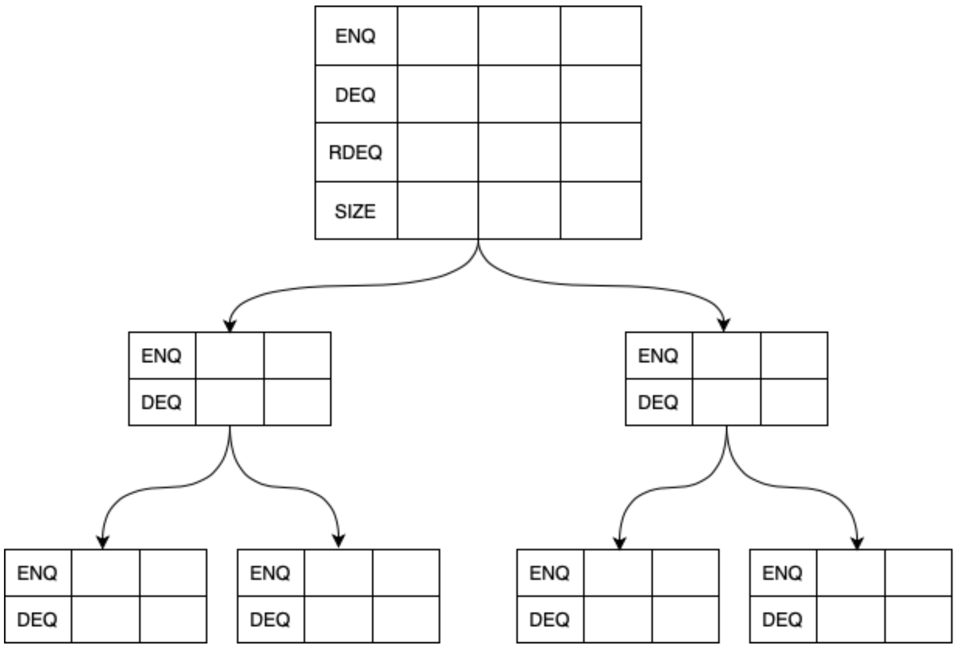


Figure 14: Fields stored in the Queue nodes.

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 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 $\mathtt{l_i}$ 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 $\mathtt{num_{enq\text{-}left}}$, $\mathtt{num_{deq\text{-}left}}$, $\mathtt{num_{enq\text{-}right}}$, $\mathtt{num_{deq\text{-}right}}$. 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 $\mathtt{sum_{non\text{-}null\ deq}}$. `size` in a `block` shows the number of elements after the `block` has finished and $\mathtt{sum_{non\text{-}null\ deq}}$ is the total number of non-null dequeues till the `block`.

`Enqueue(e)` just `append`s 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 `get`s and returns the `operation`'s element.

`Append(op)` adds `op` to the invoking process's leaf's `ops` and `blocks`, `propagate`s 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 p`n`'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 `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

► *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 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>= IndexDeq(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
        return null                     ▷ Check if the queue is empty.
    else
        e= i - root.blocks[b-1].size + root.blocks[b-1].sum_enq
                                        ▷ E_e(root) is the response.
        return root.GetENQ(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, h)
        end if
    end for
    new= CREATEBLOCK(h)
    if new.num==0 then return true
    end if
    result= blocks[h].CAS(null, new)
    h_p= parent.head
    this.ADVANCE(h, h_p)
    return  result
end REFRESH

void ADVANCE(int h, int hp)
    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 GetEnq(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.GetEnq(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.GetEnq(subBlock, i)
    end if
end GetEnq

⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num_enq ≥ i.
<int, int> IndexDeq(int b, int i)                                 ▷ Returns <x, y> if D_{this,b,i} = D_{root,x,y}.
    if this is root then
        return <b, i>
    else
        dir= (parent.left==n)? left: right                        ▷ check if this node is a left or a right child
        superBlock= parent.BinarySearch(sum_deq-dir, i+blocks[b-1].sum_deq, blocks[b].super-2, blocks[b].super+2)
                                  ▷ superblock's group has at most p difference with the value stored in super[].
        if dir is left then
            i+= blocks[b-1].sum_enq-blocks[superBlock-1].sum_enq-left    ▷ consider the enqueues in the previous blocks from the left child
        end if
        if dir is right then
            i+= blocks[b-1].sum_enq-blocks[superBlock-1].sum_enq-right   ▷ consider the enqueues in the previous blocks from the right child
            i+= blocks[superBlock].num_deq-left                          ▷ consider the dequeues from the right child
        end if
        return this.parent.IndexDeq(superBlock, i)
    end if
end IndexDeq
```

4 Proof of Correctness

To prove a shared data structure works correctly, it is sufficient to show it is linearizable. 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 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 operation of each block which we use to prove the correctness of `DSearch()`, `GetEnq()` and `IndexDeq()`. Finally (2) is followed by proving the correctness of the way we compute the response of a dequeue.

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* b to be the blocks in `n.blocks[0..`i`]`.

Next, we show that the value of `head` in a node can be increased. After 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 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 the last block written into `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`]` $\neq$ `null` for every $x > 0$. The truth of the invariant may be affected by writing into `n.blocks` or incrementing `n.head`. We show the invariant still holds after these two changes.

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`, because if so then n`.blocks[`h`].CAS` could not be successful. Writing into n`.blocks[`n`.head=` h`]` 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. If `n.head` is incremented to $h+1$ it is sufficient to show `n.blocks[`h`]`$\neq$`null`. `Advance()` is called in Lines 322 and 314. In case `advance`, Line 320 was finished before doing 327 wether 320 is successful or not n`.blocks[h]` $\neq$ `null`. In the other case also n`.blocks[h]` $\neq$ `null` because of the `if` condition in Line 313. ☐

We define the subblocks of a block recursively.

Definition 5 (Subblock)**.** A block is a *direct subblock* of ith block in node n if it is in `n.left.blocks[n.blocks[`$i-1$`].end`$_{\texttt{left}}$`+1..n.blocks[`i`].end`$_{\texttt{left}}$`]` or in `n.right.blocks[n.blocks[`$i-1$`].end`$_{\texttt{right}}$`+1..n.blocks[`i`].end`$_{\texttt{right}}$`]`. Block b is a *subblock* of block c if b is a direct subblock of c or a subblock of a direct subblock of c. We say block b is *propagated* to node n if b is in $n.blocks$ 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 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.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[$i-1$].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'.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 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 then direct subblocks of blocks of n.blocks[i] and n.blocks[j] have to overlap. Without loss of generality assume left child subblocks of n.blocks[i] overlap with the left child subblocks of n.blocks[j]. By Lemma 6 we have n.blocks[i].end$_{\texttt{left}}$ $\leq$ n.blocks[$j-1$].end$_{\texttt{left}}$, so the 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. ☐

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

Definition 8 (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 set of the operations in leaf subblocks of b. We denote the set of operations of block b by *ops*(b) and the union set of operations of set of blocks B by *ops*(B). We also say b contains *op* if op $\in$ ops(b).

Definition 9 (Superblock)**.** Block b is *superblock* of block c if c is a direct subblock of b.

Corollary 10. *Every block has at most one superblock.*

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

Operations are distinct Enqueues and Dequeues invoked by processes. 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 leads us to contradictory with the hypothesis. ☐

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

Now we want to say block 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. ☐

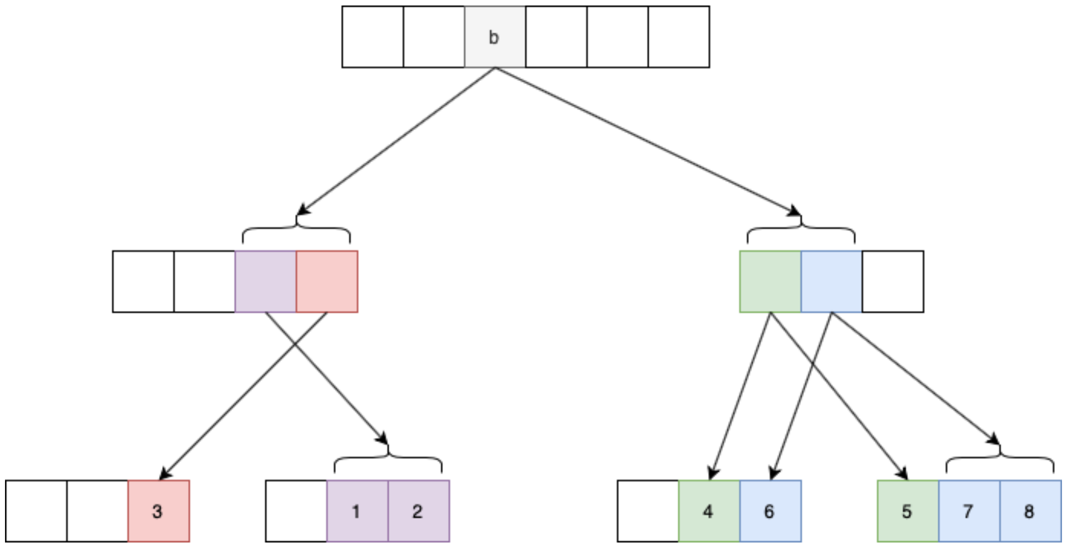


Figure 15: Order of operations in b. Operations in the leaves are ordered with numerical order shown in the drawing.

Now we define the ordering of operations stored in each node. In the non-root nodes we only need ordering of 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 25 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 $leaf\texttt{.blocks[}b\texttt{].num}_{\texttt{enq}} = 0$. If n is an internal node, then

$$\begin{aligned}E(n,b) =& E(n\texttt{.left}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{left}}+1)\cdot E(n\texttt{.left}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{left}}+2)\cdots E(n\texttt{.left}, n\texttt{.blocks[}b\texttt{].end}_{\texttt{left}})\cdot\\ & E(n\texttt{.right}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{right}}+1)\cdot E(n\texttt{.right}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{right}}+2)\cdots E(n\texttt{.right}, n\texttt{.blocks[}b\texttt{].end}_{\texttt{right}})\end{aligned}$$

- $E_i(n,b)$ 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 $leaf\texttt{.blocks[}b\texttt{].num}_{\texttt{deq}} = 0$. If n is an internal node, then

$$\begin{aligned}D(n,b) =& D(n\texttt{.left}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{left}}+1)\cdot D(n\texttt{.left}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{left}}+2)\cdots D(n\texttt{.left}, n\texttt{.blocks[}b\texttt{].end}_{\texttt{left}})\cdot\\ & D(n\texttt{.right}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{right}}+1)\cdot D(n\texttt{.right}, n\texttt{.blocks[}b-1\texttt{].end}_{\texttt{right}}+2)\cdots D(n\texttt{.right}, n\texttt{.blocks[}b\texttt{].end}_{\texttt{right}})\end{aligned}$$

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

Definition 16 (Linearization)**.** $L = E(root,1).D(root,1).E(root,2).D(root,2).E(root,3).D(root,3)$...

Definition 17. Let t^{op} be the time op is invoked and ${}^{op}t$ be the time op terminates. Let t_l be the time immediately before executing Line l and ${}_lt$ be the time immediately after executing Line l. Let ${}^{op}_{l}t$ is the time immediately before running Line l of operation op and t_l^{op} is the immediate time after running Line l of operation op. 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 18 (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 established in the parent after the `Refresh`'s succesful `CAS` at Line 320.

Lemma 19. *If R is a successful instance of n.*`Refresh()`*, then we have* $EST^{t^R}_{n.\texttt{left}} \cup EST^{t^R}_{n.\texttt{right}} \subseteq ops(n.blocks_{320})$.

Proof. We show $EST^{t^R}_{n.\texttt{left}} = ops(n.\texttt{left.blocks[0..n.left.head}_{309} - 1\texttt{]}) \subseteq ops(n.blocks_{320}) = ops(n.\texttt{blocks[0..}n.head_{320}\texttt{]})$. Line 320 stores a block `new` in n that has $\texttt{end}_{\texttt{left}} = n.left.head_{334} - 1$. Therefore by Definition 5, after the successful `CAS` in Line 320 we know all blocks in $\texttt{n.left.blocks[1..}n.left.head_{334} - 1\texttt{]}$ are subblocks of $\texttt{n.blocks[1..}n.head_{310}\texttt{]}$. Because of Lemma 2 we have $n.left.head_{309} - 1 < n.left.head_{334} - 1$ and $n.head_{310} < n.head_{320}$. From Observation 13 the claim follows. The proof for the right child is the same. ☐

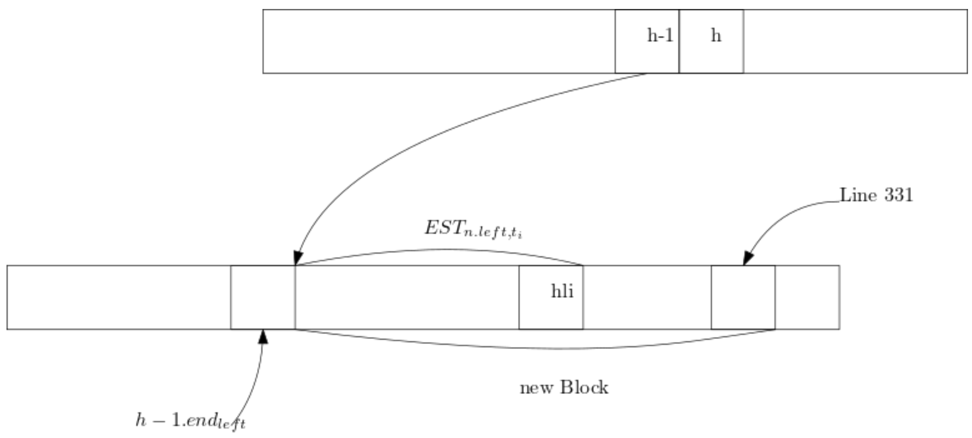


Figure 16: New established operations of the left child are in the new block. (TO UPDATE)

Corollary 20. *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 19, so it is sufficient to show when `R` terminates the established blocks in n are a superset of n`.blocks` after Line 320. Because of Lemma 3 we are sure that n`.head` is incremented after line 327. 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 established blocks in the children of n before the first `Refresh()` are guaranteed to be in n after the second `Refresh()`.

Lemma 21. *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 $n.\texttt{head}$ at Line 310. Note that by Lemma 3 R_1 and R_2 both cannot read the same value i. By Observation 2 R_2 reads larger value of $n.\texttt{head}$ 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 `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 $n.\texttt{head}$ (Line 310) and R_1 reads the value i from $n.\texttt{head}$. By Observation 2 we have ${}^{R_1}t < t^{R_1}_{310} < t^{R2'}_{310}$ (see Figure 17). By Lemma 20 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` ($t^{R2'}_{320}$) is before R_2's termination (t^{R_2}), so by Observation 13 $ops(EST_{n.\texttt{left},t^{R_1}}) \cup ops(EST_{n.\texttt{right},t^{R_1}}) \subseteq ops(n.\texttt{blocks}_{t^{R_2}})$

If R_2 reads some value greater than $i+1$ in Line 310 it means $n.\texttt{head}$ has been incremented more than two times since ${}^{R_1}_{310}t$. By Lemma 4, when $n.\texttt{head}$ is incremented from $i+1$ to $i+2$, $n.\texttt{blocks[i+1]}$ 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 310 and has put its block in $n.\texttt{blocks}[i+1]$ before R_2's read of $n.\texttt{head}$ 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 Lemma 19 on R_3 the claim holds. □

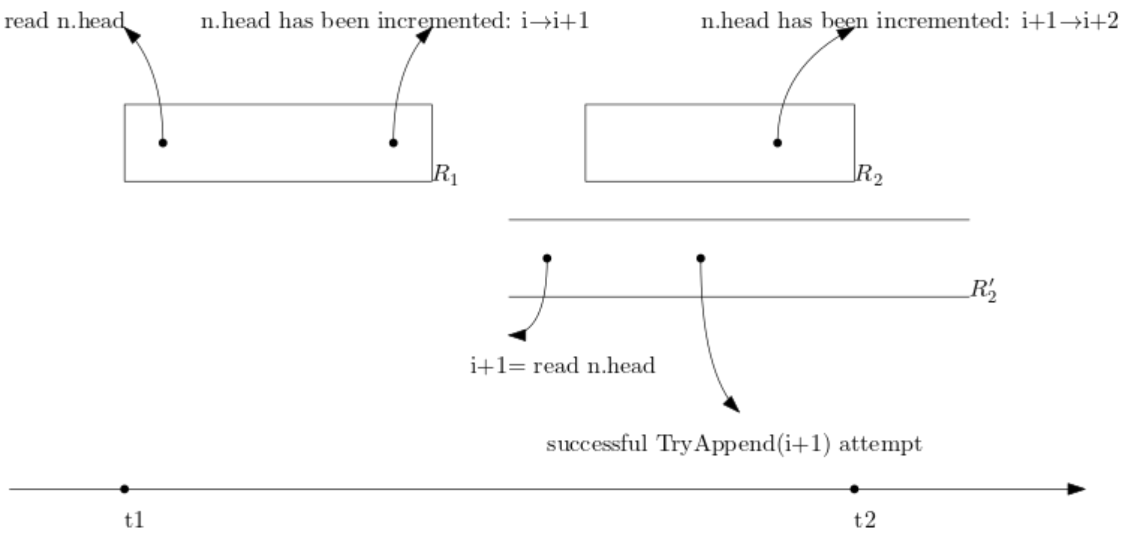


Figure 17: ${}_{R_1}t < t^{R_1}_{310} <$ incrementing `n.head` from i to $i+1 < t^{R_2'}_{310} < t^{R_2'}_{320} <$ incrementing `n.head` from $i+1$ to $i+2 < t_{R_2}$

Corollary 22. $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ {}_{302}\texttt{t}}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ {}_{302}\texttt{t}}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ \texttt{t}_{303}})$

Proof. If the first `Refresh()` in line 302 returns `true` then by Lemma 20 the claim holds. If the first `Refresh()` failed and the second `Refresh()` succeeded the claim still holds by Lemma 20. Otherwise both failed and the claim is satisfied by Lemma 21. □

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

Corollary 23. *For $A =$ `Append(`b`)` we have $ops(b) \subseteq ops(EST_{\texttt{n},t^A})$ where $n \in \{$nodes in the path from the leaf to the 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 by Corollary 22 and then calls n.`parent.Propagate()` utill n is `root`. □

Corollary 24. *After* `Append(`b`)` *finishes, b is in the nodes the path from the leaf to the root for exactly one time.*

Proof. By the previous corollary and Lemma 24 there is exactly one block in each node containing b. $\square$

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

Lemma 25. *Each block contains at most one operation of each process.*

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process p in block b in node n. Without loss of generality op_1 is invoked earlier than op_2. Process p cannot invoke more than one operations concurrently, so op_1 has to be finished before op_2 begins. By Corollary 24 before op_2 calls `Append()`, op_1 exists in every node of the tree on the path from p's leaf to the root. 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. $\square$

Lemma 26. *Each block has at most p direct subblocks.*

Proof. The claim follows directly from Lemma 25 and the observation that each block appended to an initial 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. $\square$

`DSearch(`e`, `end`)` returns `<`b`, `i`>` so that ith `Enqueue` in bth block of root is eth `Enqueue` in entire sequence stored in the root.

Lemma 27 (`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 `sum`$_{\texttt{enq-left}}$, `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`$_{\texttt{enq-right}}$, `sum`$_{\texttt{enq}}$ values of `root.block[`$0 \cdots end$`]` 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`$_{\texttt{enq}}$ $< e$. 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 Binary Search finds b in the range mentioned. Finally i, is computed from the definition of `sum`$_{\texttt{enq}}$. $\square$

Lemma 28 (Get correctness). *If* n.blocks[b].$\texttt{num}_{\texttt{enq}} \geq i$ *then* n.GetENQ(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, n is a leaf. Leaf blocks each contain exactly one operation, so only n.GetENQ(b,1) can be called when n is a leaf. Line 403 of n.GetENQ(b, 1) returns the element of the enqueue operation stored in the bth block of leaf n .

For the induction step we prove if n.child.GetENQ(sb, i) returns $E_i(n.child, sb)$ then n.GetENQ(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 18). $\texttt{num}_{\texttt{enq-left}}$ field in n.blocks[b] stores the number of Enqueue() operations from the blocks's subblocks in the left child of n. So the ith enqueue operation is propagated from the right child if and only if i is greater than b.$\texttt{num}_{\texttt{enq-left}}$. Line 404 decides whether the ith enqueue in bth block of internal node n is in the left child or right child subblocks of n.blocks[b]. By definition 8 to find an operation in subblocks of n.blocks[i] we need to search in the range n.left.blocks[n.blocks[i-1].$\texttt{end}_{\texttt{left}}$+1..$n$.blocks[$i$].$\texttt{end}_{\texttt{left}}$] $\cup$ n.right.blocks[n.blocks[i-1].$\texttt{end}_{\texttt{right}}$+1..$n$.blocks[$i$].$\texttt{end}_{\texttt{right}}$].

There are $eb = n.blocks[b-1].sum_{enq-left}$ enqueues in the blocks before the left subblocks of n.blocks[b], so $E_i(n, b)$ is $E_{i+eb}(n.left)$ which is $E_{i'}(n.left, b')$ for some b' and i'. We can compute b' and then search for $i+eb$th enqueue in n.left, where i' is i+eb-n.left.blocks[$b'-$1].$\texttt{sum}_{\texttt{enq}}$. The parameters in Line 405 are for searching $E_{i+eb}(n.left)$ in n.left.block in the expected range of blocks, 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 left Enqueues are before the right Enqueues, there are n.blocks[b].$\texttt{num}_{\texttt{enq-left}}$ enqueues ahead from the left child. So we need to search for i-n.blocks[b].$\texttt{num}_{\texttt{enq-left}}$+ n.blocks[b-1].$\texttt{sum}_{\texttt{enq-right}}$ (Line 409). Other parameters for the left child are chosen similar to the right child.

So, in both cases the direct subblock containing $E_i(n, b)$ is computed in Lines 405 and 409. Finally, n.child.GetENQ(subblock, i) is invoked on the subblock containing $E_i(n, b)$ and it returns $E_i(n, b)$ by the hypothesis of the induction. $\square$

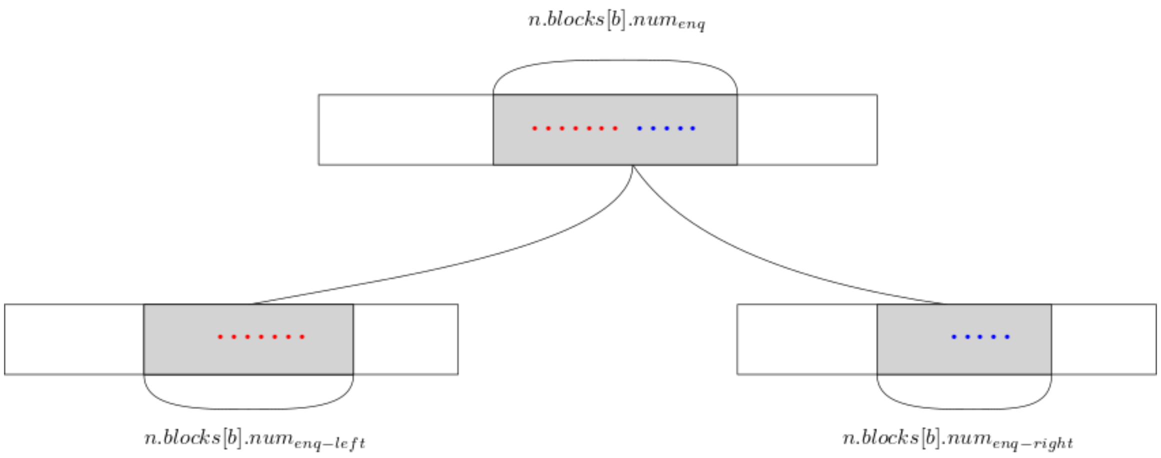


Figure 18: The number and ordering of the enqueue operations propagated from the left and the right child to n.blocks[b]. Enqueue operations from the left subblocks (colored red), are ordered before the enqueue operations from the right child (colored blue).

Definition 29. A Refresh is *successful* if it performs a successful `CAS` on Line 320. If `Refresh` instance R_1 does its `CAS` at Line 320 sooner than `Refresh` instance R_2 we say R_1 *happened* before R_2.

Let `i` be the value R_n, a successful instance of `Refresh()` on the node `n`, reads from `n.head`. R_n does a successful `CAS(null, new)` into `n.blocks[h]`, where `b` is the `new`. Without loss of generality for the rest of this section assume `n` is the left child of `n.parent`. From now on we say `p` as an abbreviation for `n.parent`. Let R_p be the first successful `p.Refresh()` that reads some value greater than `i` for `left.head` and is contains `b` in its created block `s` in Line 317. From Lemma 24 we know there could be only one `p.Refresh()` propagating `b`. R_p does a successful `CAS(null, new)` into `p.blocks[j]`, where `s` is the `new`.

Although other fields of `b` are set while creating it, because the index of the superblock of `b` is not known until it is propagated, R_n cannot set the `super` field of a `b` while creating it. One approach is to set the `super` field of `b` after it is propagated by R_b but this would not be efficient because there might be p subblocks in `s`. However, 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 aproximate 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.Refresh`es could happen that contain blocks from the right child of `p` and `j` could be arbitrarily (right word?) greater than `b.super`. We set `b.super` to `p.head` after appending `b` to `n.blocks` (Line 326). Maybe R_n goes to sleep at some time after installing `b` and before setting `b.super`. In this case the next `Refresh`es 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` by the same process or another process's `n.Refresh()` 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`.

Observation 30. *After* `n.blocks[i].CAS(null, b)` *succeeds,* `n.head` *cannot increase from* `i` *to* `i+1` *unless* `b.super` *is set.*

Proof. From the Observation 2 we know the only change to `n.head` is on Line 327 which is incrementing. 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 31. *If* `n.blocks[i].super` *is* `null`*, then* `n.head` $<$ `i` *and* `n.blocks[i+1]` *is* `null`*.*

Proof. If `b.super` is `null` then `n.head` cannot advance so the next `n.Refresh()` will fail. By the previous corollary, `b.super` has to be set before the next successful Refresh() on n after R_n. ☐

Now let us talk about how the `p.Refresh`es that took place after the putting `b` into `n`, will help to set `b.super` and propagate `b`.

Lemma 32. *If* `b` $\in$ `n.parent.blocks[i]` *then* `b.super` $\leq$ `i`*.*

Proof. For R_p to contain block `b`, it has to read `n.head` greater than `h` (see Line 334). For `n.head` to be greater than `h` it means `n.head` is incremented in Line 327 which means `b.super` was already set in Line 326 (see Observation 30). So if R_p propagates `b` it means `b.super` was already set. Let `j` be the value written in `b.super`. `j` has been read in Line 310 or Line 321 which both are before calling `Advance` that sets `b.super`. From Observation 2 we know `p.Head` is non-decreasing so `j` $\leq$ `i`. The reader may wonder when the case `j` $=$ `i` happens, it happens when `p.blocks[j]=null` while `j` is read and R_p puts its created block into `n.blocks[j]`. ☐

Lemma 33. *If* R=`n.Refresh()` *puts* `b` *in* `n.blocks[h]` *at Line 320, then the block created by one of the next two successful* `p.Refresh`*es according to the Definition 29 contains* `b` *and* `b.super` *is set before Line 317 of the the second successful* `p.Refresh()`*.*

Proof. It is sufficient to prove one of the two successful `p.Refresh()`es propagate `b`. If the first successful `p.Refresh()` propagated `b` then the claim is true, so in the remaining part we assume the first `p.Refresh()` did not propagate `b` and prove the second `p.Refresh()` propagates `b`.

`b.super` is set by some instance of `Refresh()` on `n` or `p` showed by `R’` and `n.head` is incremented by some `Refresh()` called `R"`. We want to know how great $j - \texttt{b.super}$ can be. `p.head` is `hp` when `R’` reads it. From Lemma 6 `p.head` could only increase from `hp` to `hp+1` if `p[hp]` $\neq$ `null`. In other words there should be a successful `p.Refresh()` for `p.head` to increase. We claim there cannot be another successful `p.Refresh()` after `R’` reads `p.head` and before $\mathtt{R_p}$ performs Line 334.

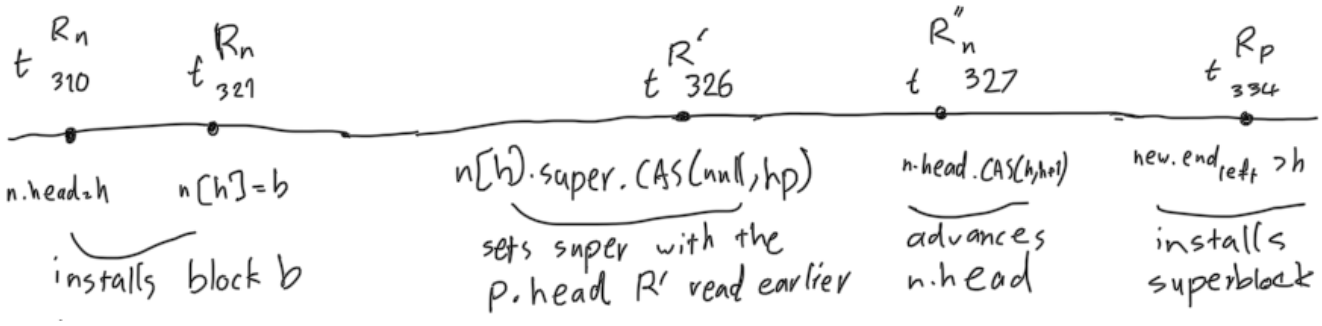


Figure 19: Time relations between $\mathtt{R_n}$,$\mathtt{R_p}$,`R’`,`R"`

Assume the first successful `p.Refresh()` after t_{320}^{Rn} did not set `b.super`. It might happen maybe because the value read for h_{left} in Line 312 is less than `i` or maybe $\mathtt{i} = \mathtt{h_{left}}$ and `left.blocks[`$\mathtt{h_{left}}$`]` $=$ `null`, which means `n.head` is advanced but `b` is still not installed in `n.blocks[i]` which means $\mathtt{R_n}$ has not reached to the Line 320.

Let the first successful `p.Refresh()` be `Rp1` and the second next successful `p.Refresh()` be `Rp2`. If `Rp1` reads `x` in Line 310, then `Rp2` has to read `x+1` in Line 310 (iduced from 6, 2). See the timeline in Figure 20 for two consecutive successful `Refresh()` instances `Rp1, Rp2` on `p`.

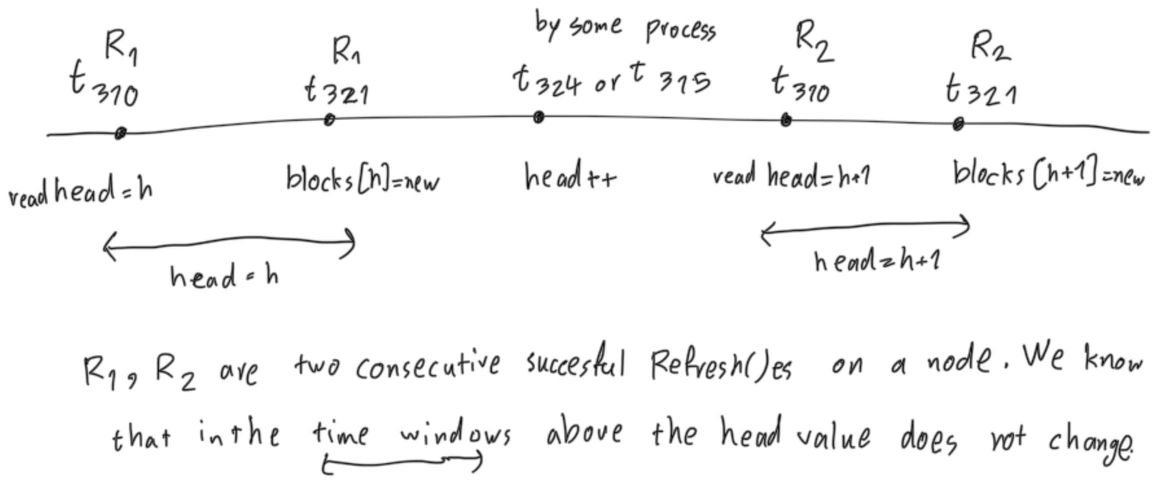


Figure 20: Rp2 reads p.head after t_{321}^{Rp1}, which is after t_{321}^{Rn}. Rp2 has to help increment n.head and set b.super.

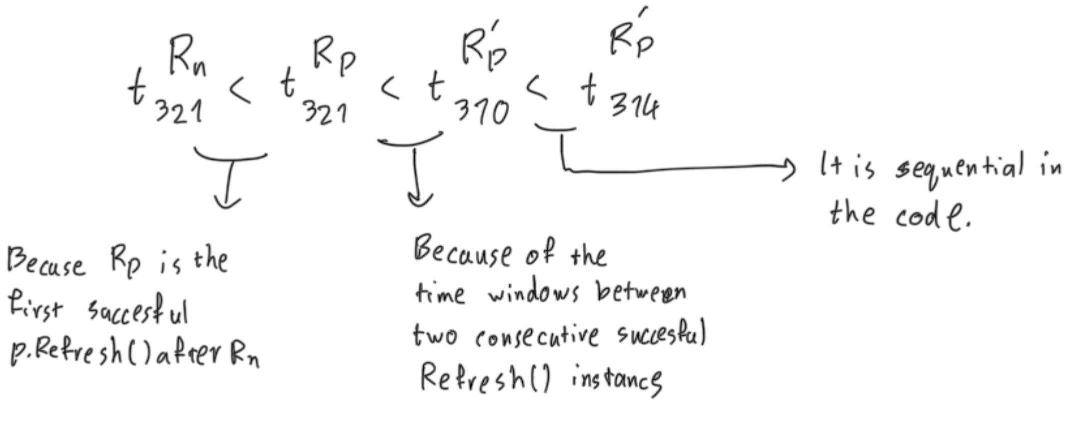


Figure 21: The second Refresh on `p` contains `b` and reads `n.head` $>$ `i`.

So `b.super` has set by some process before the second next successful `p.Refresh()` on Line 326. Since `i` is read in the Line 310 then the `CreateBlock()` in Line 317 is going to read some value fo `left.head` greater than `h` and propagates `b` to `p`. So if `b` was not propagated already we are sure the second next successful `p.Refresh()` propagates `b`. □

Corollary 34. *If* `b` *has propagated to* `f`*, then* `b.super` *has at most 1 difference with the index of the superblock of* `b` *in* `p`*.*

Lemma 35 (Computing SuperBlock)**.** *For the* `superblock` *value computed in line 418 of* `n.IndexDEQ(b,i)` *we have* `n.parent.blocks[superblock]` *contains* $D_{n,b,i}$.

Proof. First we show the value read for `super[b.group]` in line 418 is not null. Values $\texttt{np}_{\texttt{dir}}$ read in lines **??**, `super` are set before incrementing in lines **??**,**??**. So before incrementing $\texttt{num}_{\texttt{propagated}}$, $\texttt{super[num}_{\texttt{propagated}}\texttt{]}$ is set so it cannot be null while reading. Then by Lemma **??**if we search in the range p, we can find the superblock.

□

Lemma 36 (Index correctness)**.** *If* $\texttt{n.blocks[b].num}_{\texttt{deq}} \geq \texttt{i}$ *then* `n.IndexDEQ(b,i)` *returns the rank in* $D(root)$ *of* $D_{n,b,i}$.

Proof. We will prove this by induction on the distance of `n` from the `root`. We can see the base case where `n` is root is trivial (Line 415). In the non-root nodes `n.IndexDEQ(b,i)` computes the superblock of the ith Dequeue in the bth block of `n` in `n.parent` by Lemma 35 (Line 418). After that the order in $D(n.parent, superblock)$ is computed. Note that by Lemma 25 in each block there is at most one operation from each process and operations of one type are ordered based on the order in the subblocks (See Figure 22). Finally `index()` is called on `n.parent` recursively and it returns the correct response from induction hypothesis. If the operation was propagated from the right child the number of dequeues from the left child are added to it (Line **??**), because the left child operations come before the right child operations (Definition 15). □

Make sure to show preconditions of all invocation of `BinarySearch` *are satisfied.*

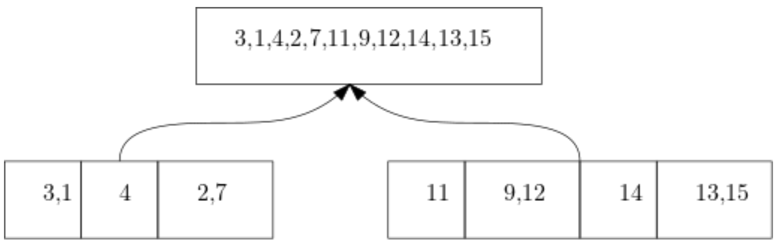


Figure 22: Relation of ordering of operations of a block from its subblocks

Definition 37. Assume the operations in L are applied on an empty queue. If element of `enqueue e` is the response to `dequeue d` then we say `R(d)=e`. If `d` 's response id `null` (queue is empty) then `R(d)=null`.

Definition 38. In an execution on a queue, the dequeue operations that return some value are called *non-null dequeues.*

Observation 39. *In a sequential execution on a queue, kth non-null dequeue returns the* `element` *of kth enqueue.*

Lemma 40. `root.blocks[b].size` *is the size of the queue if the operations in the prefix for the bth block in the root are applied with the order of L.*

Proof. need to say? :: If the size of a queue is greater than 0 then a `Dequeue()` would decrease the size of the queue, otherwise the size of the queue remains 0. By definition 15 enqueue operations come before dequeue operations in a block in L.

We prove the claim by induction on `b`. Base case `b=0` is trivial since the queue is initialy empty and `root.blocks[0].size=0`. For `b=i` we are going to use the hypothesis for `b=i-1`. If there are more than `root.blocks[i-1].size+ root.blocks[i].sum`$_{\texttt{enq}}$ dequeue operations in `root.blocks[i]` then the queue would become empty after `root.blocks[i]`. Otherwise we can compute the size of the queue after bth block using with this equality `root.blocks[b].size= root.blocks[b-1].size+ root.blocks[b].sum`$_{\texttt{enq}}$`-` `root.blocks[b].sum`$_{\texttt{deq}}$ (Line 344). See Table 4 for an example of running some blocks of operations on an empty queue. □

Lemma 41 (Duality of #non-null dequeues and `block.size`)**.** *If the operations are applied with the order of L, the number of non-null dequeues in the prefix for a block* `b` *is* `b.sum`$_{\texttt{enq}}$`-b.size`

Proof. There are `b.sum`$_{\texttt{enq}}$ enqueue operations in the prefix for `b`, then the size of the queue after the prefix for `b` is `#enqs - #non-null dequeues` in the prefix for `b`, by Observation 35. So `#non-null dequeues` is `b.sum`$_{\texttt{enq}}$`-b.size` . The correctness of the `block.size` field is shown in Lemma 40. □

Lemma 42. `R(D`$_{\texttt{root,b,i}}$`)` *is* `null` *iff* `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0`.

Lemma 43 (Computing Response)**.** `FindResponse(b,i)` *returns* `R(D`$_{\texttt{root,b,i}}$`).element`.

Proof. First note that by Definition 15 the linearization ordering of operations will not change as new operations come so instead of talking about the linearization of operations before the $E_i(root, b)$ we talk about what if the whole operation in the linearization are applied on a queue.

$D_{root,b,i}$ is $D_{root,root.blocks[b-1].sum_{deq}+i}$ from the definition 15 and sum_{enq}. $D_{root,b,i}$ returns `null` if `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0` by Lemma 42 (Line 220). Otherwise if it is d'th non-null dequeue in L it returns d'th enqueue by Observation 39 . By Lemma 41 there are `root.blocks[b-1].sum`$_{\texttt{enq}}$ `- root.blocks[b-1].size` non-null dequeue operations before prefix for `root.blocks[b-1]`. Note that the dequeues in `root.blocks[b]` before the ith dequeue are non-null dequeues. So the response is $E_{i-root.blocks[b-1].size+root.blocks[b-1].sum_{\text{deq}}}(root)$ (Line 222). See figure 23.

After computing `e` we can find `b,i` such that $E_i(root, b) = E_e(root)$ using `DSearch` and then find its `element` using `GetEnq` (Line 223). □

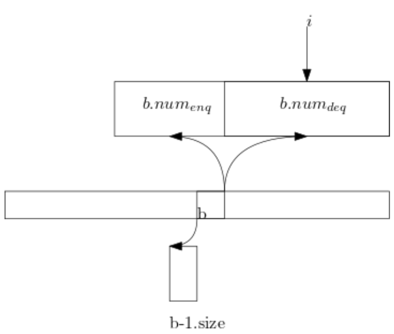


Figure 23: The position of $E_i(root, b)$.

	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 4: An example of root blocks fields. Blocks are from left to right and operations in the blocks are also from the left to right.

Theorem 44 (Main)**.** *The queue implementation is linearizable.*

Proof. We choose L in Definition 15 to be linearization ordering of operations and prove if we linearize operations as L the queue works consistently. □

Lemma 45 (satisfiability)**.** *L can be a linearization ordering.*

Proof. To show this we need to say if in an execution, $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts then $\texttt{op}_1$ is linearized before $\texttt{op}_2$. If $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts it means $\texttt{op}_1\texttt{.Append()}$ is terminated before $\texttt{op}_2\texttt{.Append()}$ starts. From Lemma **??** $\texttt{op}_1$ is in `root.blocks` before $\texttt{op}_2$ propagates so $\texttt{op}_1$ is linearized before $\texttt{op}_2$ by Definition 15.

Once some operations are aggregated in one block they will be propagated together up to the root and we can linearize them in any order among themselves. Furthermore in L we arbitrary choose the order to be by process id, since it makes computations in the blocks faster . □

Lemma 46 (correctness)**.** *If operations are applied as L on a sequential queue, the sequence of the responses would be the same as our algorithm.*

Proof. Old parts to review We show that the ordering L stored in the root, satisfies the properties of a linearizable ordering.

1. If op_1 ends before op_2 begins in E, then op_1 comes before op_2 in T.

   ► This is followed by Lemma **??**. The time op_1 ends it is in root, before op_2, by Definition 15 op_1 is before op_2.

2. Responses to operations in E are same as they would be if done sequentially in order of L.

   ► Enqueue operations do not have any response so it does no matter how they are ordered. It remains to prove Dequeue d returns the correct response according to the linearization order. By Lemma 43 it is deduced that the head of the queue at time of the linearization of d is computed properly. If the Queue is not empty by Lemma 28 we know that the returning response is the computed index element.

□

Lemma 47 (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 `IndexDeq()` is called. `IndexDeq` does a search with range p in each level (Lemma **??**) 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 27, which is $O(\log \text{size of the queue when } \texttt{enqueue} \text{ is invoked}) + \log \text{size of the queue when } \texttt{dequeue} \text{ is invoked})$. Each search in `GetEnq()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma 26), so `GetEnq()` 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 48 (CASes invoked)**.** *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()` has 2 `CAS`es, one in Line 320 and one in Lines **??** or **??**. ☐

Lemma 49 (`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* $O\big(\log(\texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size})\big)$ *steps.*

Proof. First we show $end - b - 1 \leq 2 \times \texttt{root.blocks[}b-1\texttt{].size} + \texttt{root.blocks[}end\texttt{].size}$. Suppose there were more than `root.blocks[b].size` `Dequeue`s in $\texttt{root.blocks[}b+1\cdots end-1\texttt{]}$. 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 `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 $\texttt{root.blocks[}b\texttt{].size}$ `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 $\texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size}$ `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 `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 `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 24. 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, `i` is computed via the definition of $\texttt{sum}_{\texttt{enq}}$ in constant time (Line 807). So the claim is proved. ☐

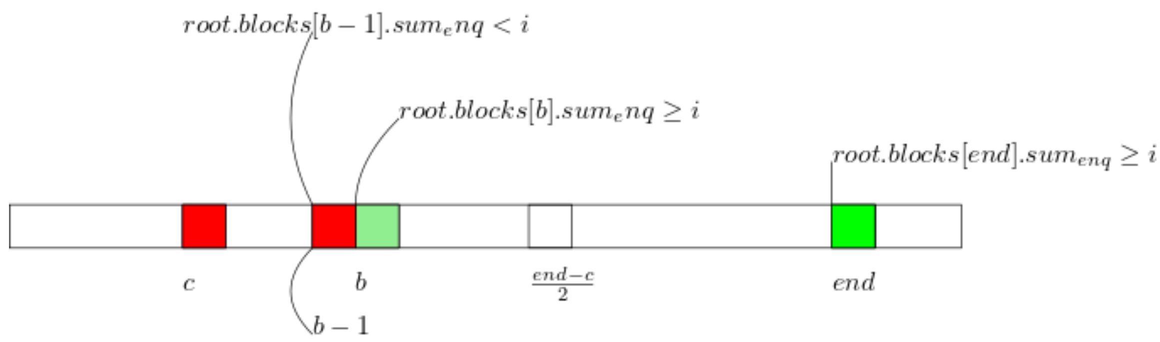


Figure 24: Distance relations between `start`, b, end. `start` is shown with c.

4.1 Garbage Collection or Getting rid of the infinite Arrays

5 Using Queues to Implement Vectors

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

6 Conclusion

possible directions for work

Maybe Stacks

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