Wait-free Queues with Polylogarithmic Step Complexity

Hossein Naderibeni

supervised by Eric Ruppert

August 31, 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.

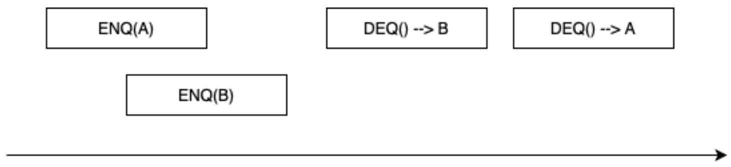


Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.

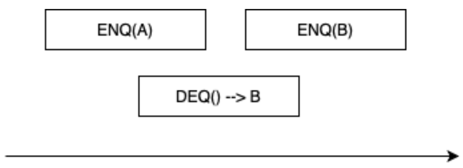


Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A` or nothing.

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.

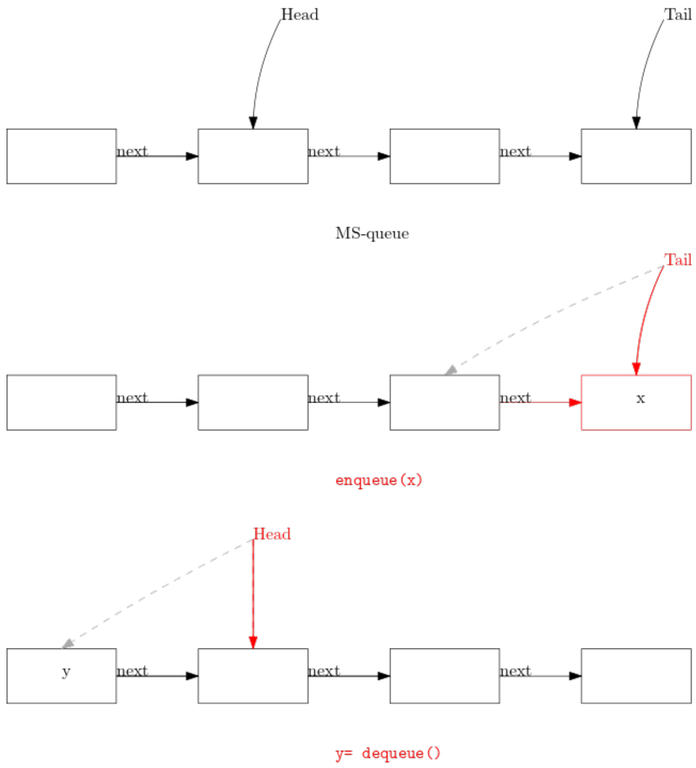


Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

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

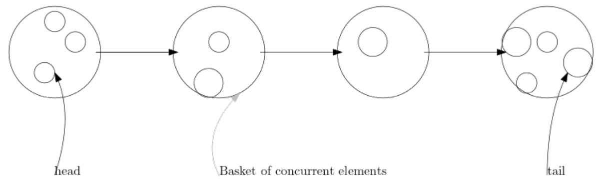


Figure 4: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do `CAS`. To order the operations in a basket, the mechanism in the algorithm for processes is to `CAS` again. The successful process will be the next one in the basket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

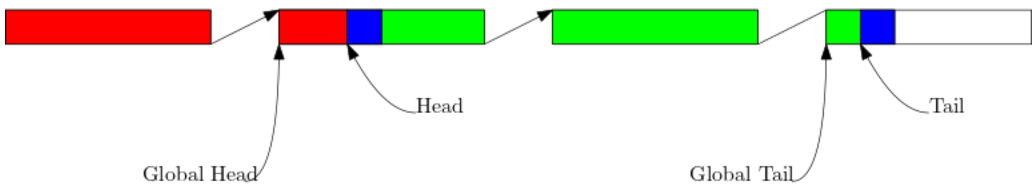


Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

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 work

Jayanti and Petrovic introduced a wait-free polylogarithmic multi-enqueuer single-dequeuer queue [**?**]. We are going to design a polylogarithmic multi-enqueuer multi-dequeuer queue using some of their ideas. But we do not use `CAS` operations with big words and do not put a limit on the number of concurrent operations. We apply two ideas from their work to create a new shared data structure which enables processes to agree on the linearization ordering of the processes. We use the shared tournament tree among p 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 the process wishes to do an operation it appends the operation to its leaf's sequence and after that, the process tries to propagate its new operation up to the tree's root. An ordering of operations propagated up to a node is stored in that node. All processes agree on the sequence stored in the root that is used as the linearization ordering.

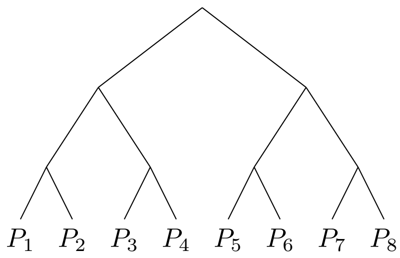
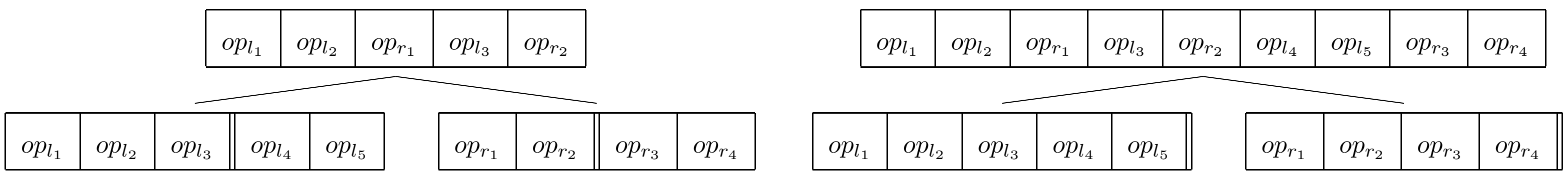


Figure 6: Each process has a leaf and in each node there is an ordering of operations stored. Each node tries to propagate its operations up to the root, which stores the totall ordering of all operations.

As we said in each node the sequence of operations is stored. We implement the sequence using an array and appending to the sequence by doing `CAS` operations on the first null element in the array. In each propagate step, our algorithm uses a subroutine Refresh(n) that aggregates new operations from node n's both children (that have not already been propagated to n) and tries to append them into n. The general 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()`es fail there is another instance of `Refresh()` in between which has succeeded to do `CAS` and has already appended the operations the first `Refresh` was trying to append. This mechanism makes us to do `TryAppend`s twice instead of `Append`ing to the array. From the next paragraph we explain our ideas which are different from Jayanti.



(a) Operations after || are new.

(b) New operations are added to the parent node.

Figure 7: Successful `Refresh`, operations in children after || are new.

Instead of storing operations explicitly in the nodes, we are going to 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. Since one operation cannot invoke two operations concurrently. Also as 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 one `Porpagate()` step in a node they are going to be propagated up to the root together. From now on instead of propagating sequence of operations in `Refresh` steps we propagate `block`s of operations. The idea is that we can describe a blocks contents only using some numbers. 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 total operations in a block and the number of operations from the left child we can order the operations in a unique way which is a complete ordering. We can instead keep track of sets of concurrent operations and create the total ordering of all operations at the root (see Figure 8). In the next paragraphs we explain the reason behind our choice of ordering.

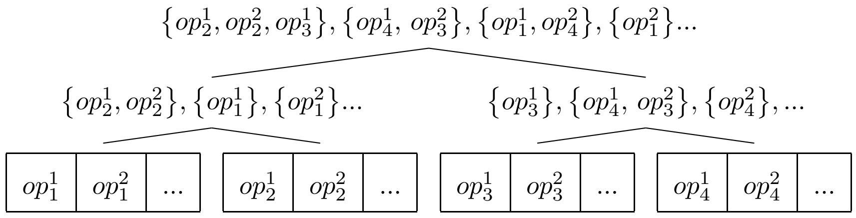


Figure 8: In each internal node, we store the set of all the operations propagated together, and one can arbitrarily linearize the sets of concurrent operations among themselves. Since we linearize operations when they are added to the root, ordering the blocks in the root is important.

Previously we talked about storing the sequence of operations in the nodes of the tree. A process may wish to know information about the root ordering. Two functionalities are to get the ith propagated operation and compute the rank of a propagated operation in the linearization. Since our algorithm is aimed for a queue, we make some assumptions here that we one wish only to know the order of a dequeue and one only tries to get an enqueue. We will explain in detail in the next but for now let's say `enqueue`s and `dequeue`s are appended to the tree and when one wants to find the response to a dequeu, it computes the order of the dequeue in linearization and using this information it computes which eneueue 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.

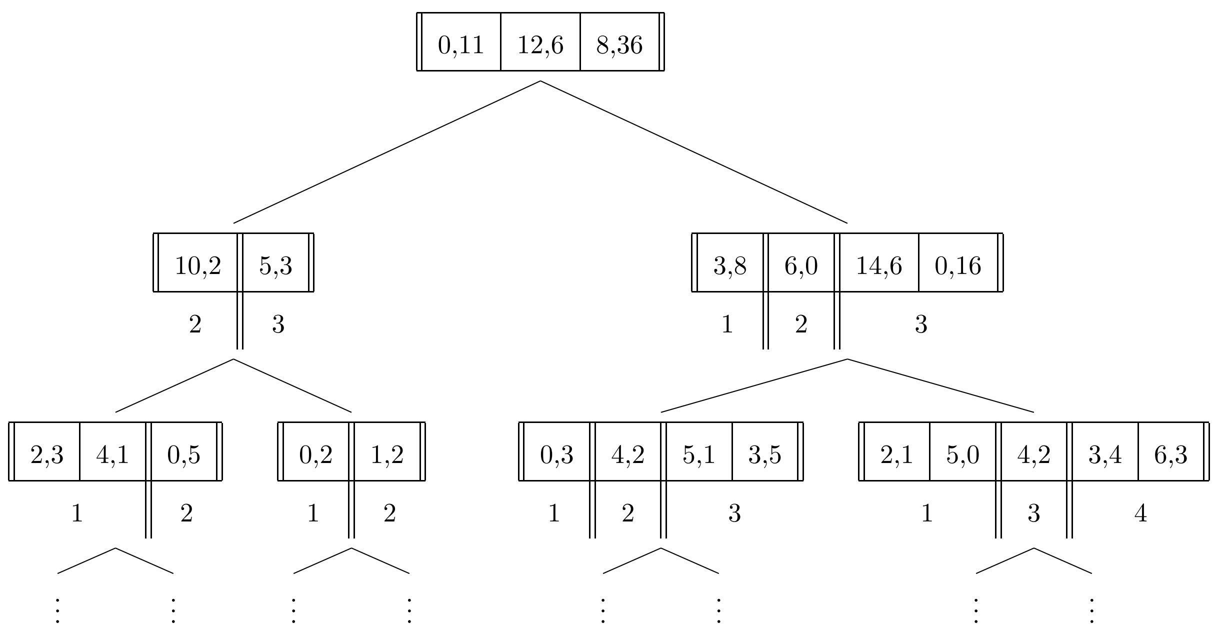


Figure 9: Showing concurrent operation sets with blocks. Each block consists of a pair(left, right) indicating the number of operations from the left and the right child, respectively. Block (12,6) in the root contains blocks (10,2) from the left child and (6,0) from the right child. Blocks between two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

The definition of linearizability allows concurrent operations to be reordered arbitrarily. Thus, a group of concurrent operations can be appended to our root sequence as one block without specifying the order among the operations.

In the original algorithm we differ between the enqueues and the dequeues in a block. Later we discuss about the reason.

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.

In a PROPAGATE() invocation path from a leaf to root, there will be REFRESH() steps with merges from $2, 4, 8, ..., p$ processes. So in a complete propagation, at most $2p$ blocks are merged into one block. `(maybe useful for analysis)`

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.

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 BSearch 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).

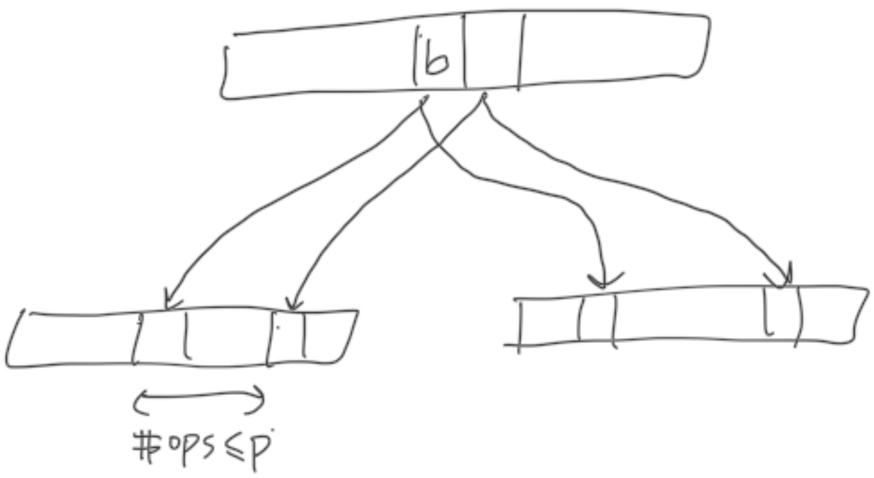


Figure 11: Block have pointers to the starting block of theirs for each child.

Starting from the root, GetIndex(i) BSearches 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 BSearch 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).

Design of a Queue Each process is assigned to a leaf in a shared tournament tree. Thus, for example, the leaf node for process p_i contains an array of elements by p_i in the order they were invoked. Each internal node of the tree contains an array of blocks of elements. Block b in node n is created in a Propagate() step and is merged block of new blocks at the time of Propagate() reading n's children blocks. Each block consists of pointers left and right, to the last block merged into itself from left and right child in that order. Moreover, two numbers, left and right, indicate the count of elements in the blocks from the left and right child consecutively. Furthermore, prefix left, and right can be computed from the prefix sum of left and right values. Elements of block b can be determined recursively (GetElements(b)). The ith element in the sequence can be determined in O($\log^2 p$) steps by recursively finding ith element in block b (GetElement(i)) After element e is propagated (appended to a block int the root), its index can be computed with GetIndex(op).

In order to compute elements of a block faster we store prefix-sum blocks(block i has tuple(right-sum=#right ops in previous block, left-sum=#left ops in previous blocks)[See Figure 10]. Here is the algorithm to get elements of a block.

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.

`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.

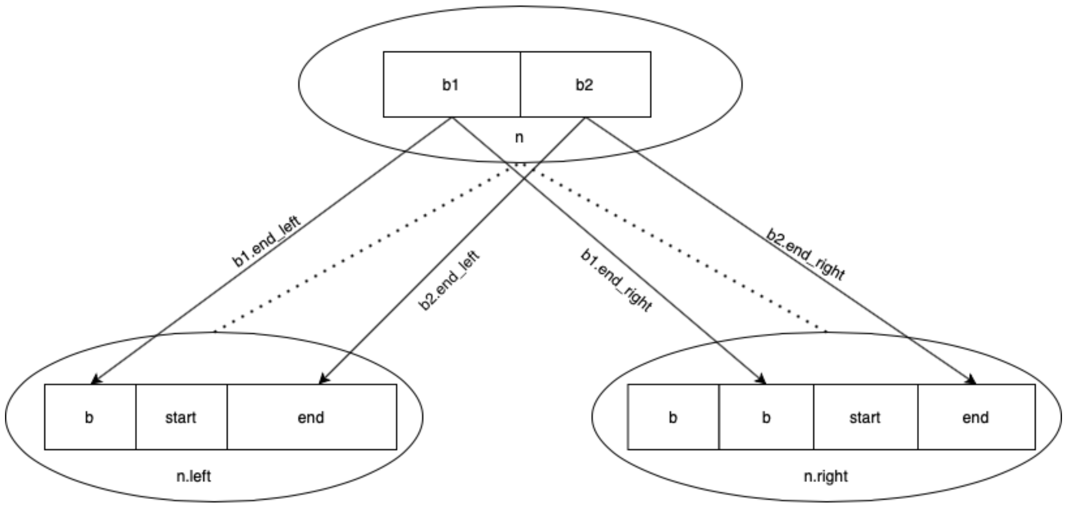


Figure 12: Snapshot of a CreateBlock()

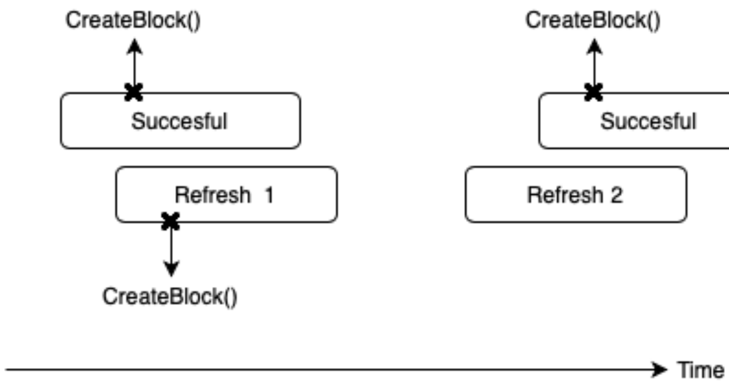


Figure 13: The second failed Refresh is assuredly concurrent to a Successful `Refresh()` with `CreateBlock` line after first failed `Refresh`'s `CreateBlock()`.

Computing `Get(n, b, i)` To find the `i`th element in block `b` of node `n`, we search among subblocks of `b` that is bounded by p. Subblocks of a block are within the start and end block of the `CreateBlock()` procedure of it.

How `Refresh(n)` works.

1. Read n's counter and head
2. Create block b
3. CAS b into n
4. If previous succeed:
   (a) Update sup of b's ending subblocks
   (b) Increment children's counters

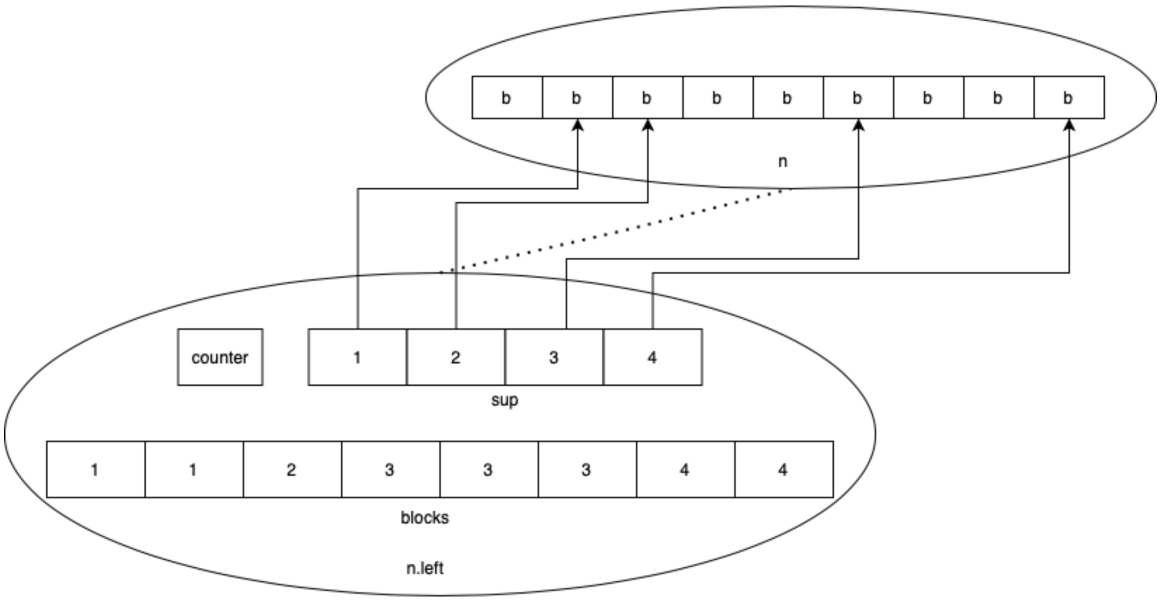


Figure 14: `Sup` and `timer` in a node, numbers on blocks are their `time` values.

Computing superblock

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.

Next, we describe how to use block tree to implement queues. The block tree, maintains the history of all operations, not only the current state of the queue. Now consider the following history of operations. What should each `Dequeue()` return? We can implement Enqueue and Dequeue using our block tree. An `Enqueue(e)` appends an operation with input argument `e` in the block tree. To do 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 **??** 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;
```

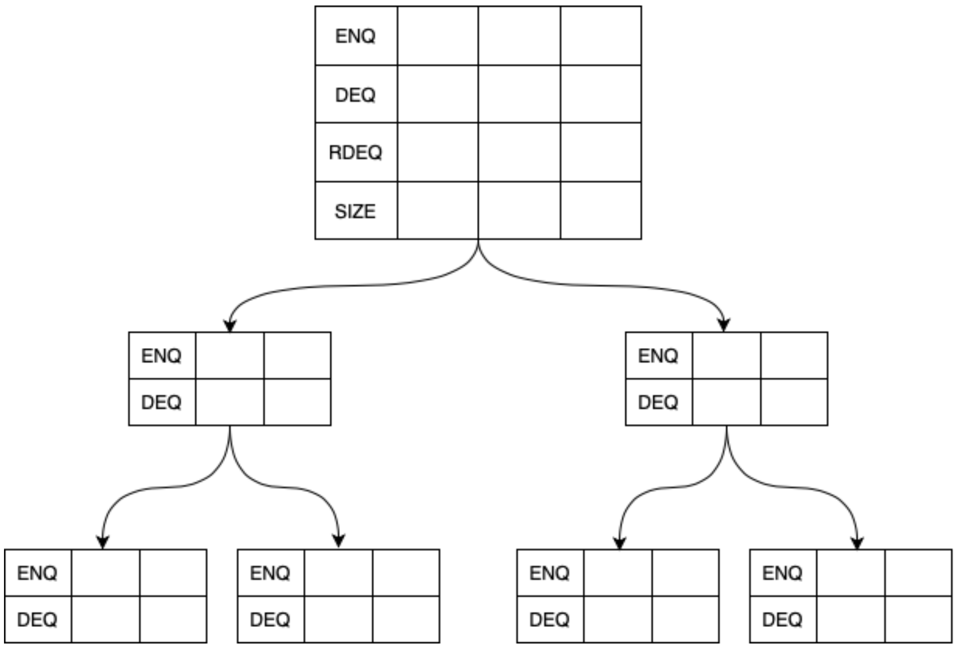


Figure 15: Fields stored in the Queue nodes.

3.1 Pseudocode description

Tree In order to reach an agreement on the order of operations among p processes, we use a Tournament Tree. Leaf `li` 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 `BSearch()`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 `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 `gets` 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.

◇ *Structures*

▶ *Node*

- **Node* left, right, parent : initialized when creating the tree.
- *BlockList*
- *int* head= 1: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.
- *int* $num_{propagated}$= 0 : # groups of blocks that have been propagated from the node to its parent.

▶ *Block*

- *int* group : the value read from $num_{propagated}$ when appending this 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* sum_{enq}, sum_{deq} : # enqueue, dequeue operations in the prefix for the block

▶ *InternalBlock* extends *Block*

- *int* end_{left}, end_{right} : indices of the last subblock of the block in the left and right child
- *int* $sum_{enq\text{-}left}$: # enqueue operations in the prefix for left.blocks[end_{left}]
- *int* $sum_{deq\text{-}left}$: # dequeue operations in the prefix for left.blocks[end_{left}]
- *int* $sum_{enq\text{-}right}$: # enqueue operations in the prefix for right.blocks[end_{right}]
- *int* $sum_{deq\text{-}right}$: # dequeue operations in the prefix for right.blocks[end_{right}]

▶ *RootBlock* extends *InternalBlock*

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

Abbreviations:

- blocks[b].sum_x=blocks[b].$sum_{x\text{-}left}$+blocks[b].$sum_{x\text{-}right}$ (for b$\geq$0 and x $\in$ {enq, deq})
- blocks[b].sum=blocks[b].sum_{enq}+blocks[b].sum_{deq} (for b$\geq$0)
- blocks[b].num_x=blocks[b].sum_x-blocks[b-1].sum_x
  (for b>0 and x $\in$ {$\emptyset$, 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 *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
    <new, np_left, np_right>= CreateBlock(h)    ▷ np_left, np_right are the
  values read from the children's num_propagated field.
    if new.num==0 then return true    ▷ The block contains nothing.
    else if blocks.tryAppend(new, h) then
        for each dir in {left, right} do
            CAS(dir.super[np_dir], null, h)    ▷ Write would work too.
            CAS(dir.num_propagated, np_dir, np_dir+1)
        end for
        CAS(head, h, h+1)
        return true
    else
        CAS(head, h, h+1)    ▷ Even if another process wins, help
  to increase the head. The winner might have fallen sleep before increasing
  head.
        return  false
    end if
end Refresh

  ⇝ Precondition: blocks[start..end] contains a block with field f ≥ i
int BSearch(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 BSearch

<Block, int, int> CreateBlock(int i)    ▷ Creates a block
  to be inserted as ith block in blocks. Returns the created block as well as
  values read from each child's num_propagated field. These values are used for
  incrementing the children's num_propagated field if the block was appended to
  blocks successfully.
    block newBlock= new(block)
    newBlock.group= num_propagated
    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].
        np_dir= dir.num_propagated
        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 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.BSearch(sum_enq, e, start, end)
    i= e- root.blocks[b-1].sum_enq
    return <b,i>
end DSearch
```

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.BSearch(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.BSearch(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.BSearch(sum_deq-dir, i+blocks[b-1].sum_deq, super[blocks[b].group]-p, super[blocks[b].group]+p)
                    ▷ 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
```

Algorithm Leaf

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

Algorithm BlockList

▷ : Supports two operations blocks.tryAppend(Block b), blocks[i]. Initially empty, when blocks.tryAppend(b, n) returns true b is appended to blocks[n] and blocks[i] returns ith block in the blocks. If some instance of blocks.tryAppend(b, n) returns false there is a concurrent instance of blocks.tryAppend(b′, n) which has returned true.blocks[0] contains an empty block with all fields equal to 0 and end_left, end_right pointers to the first block of the corresponding children.

block[] blocks: array of blocks

int[] super: super[i] stores an approximate index of the superblock of the blocks in blocks whose group field have value i.

```
boolean TryAppend(block blk, int n)
    return CAS(blocks[n], null, blk)
end TryAppend
```

3.3 Proof of Correctness

TEST Fix the logical order of definitions (cyclic refrences).

TEST Is it better to show ops($EST_{n,\ t}$) with $EST_{n,\ t}$?

Question A good notation for *the index of the* b?

Question How to remove the notion of time? To say pre(n,i) contains n.blocks[0..i] instead of EST(n,t) which head=i at time t. Is it good? Furthermore, can we remove the notion of established blocks?

Definition 1 (Block)**.** A block is an object storing some statistics, as described in Algorithm Queue. A block in a node's blocklist implicitly represents a set of operations. If `n.blocks[i]==b` we call `i` the *index* of block `b`. Block `b` is before block `b'` in node `n` if and only if the index of the `b` is smaller than the index of the `b'`'s. For a `block` in a `BlockList` we define *the prefix for the block* to be the blocks in the `BlockList` up to and including the `block`.

Lemma 2 (head Increment)**.** *Let* `R` *be an instance of* `Refresh` *on node* `n` *that reaches Line 313. After* `R` *terminates* `n.head` *is greater than h, the value read in line 310 of* `R`*.*

Proof. If Line 318 or 321 are successful then the claim holds, otherwise another process has incremented the head from `h` to `h+1`. ☐

Invariant 3 (headPosition)**.** If the value of `n.head` is `h` then, `n.blocks[i]=null` for `i>h` and `n.blocks[i]`$\neq$`null` for `i<h`.

Proof. The invariant is true initially since 1 is assigned to `n.head` and `n.blocks[x]` is null for every `x`. 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, some value is appended to `n.blocks[]` by writing into `n.blocks[head]` only in Line 313. Writing into `n.blocks[head]` preserves the invariant, since the claim does not talk about `n.blocks[head]`. The value of `n.head` is modified only in lines 318 and 321. Depending on whether the `TryAppend()` in Line 313 succeeded or not, we show that the claim holds after the increment of `n.head` in either case. If `n.head` is incremented to h it is sufficient to show `n.blocks[h]`$\neq$`null` to prove the invariant still holds. In the first case the process applied a successful `TryAppend(new,h)` in line 314, which means `n.blocks[h]` is not null anymore. Note that whether 318 or 318 return true or false, after they finish we know that `n.head` has been incremented from the value read in Line 310 (Lemma 2). The failure case is also the same since it means some non-null value has been written into `n.blocks[head]` by some process. ☐

Explain More

Lemma 4 (headProgress)**.** `n.head` *is non-decreasing over time. If* `n.blocks[i]`$\neq$`null` *and* i.0 *then* `n.blocks[i].`$\texttt{end}_{\texttt{left}} \geq$ `n.blocks[i-1].`$\texttt{end}_{\texttt{left}}$ *and* `n.blocks[i].`$\texttt{end}_{\texttt{right}} \geq$ `n.blocks[i-1].`$\texttt{end}_{\texttt{right}}$.

Proof. The first claim follows trivially from the pseudocode since `n.head` is only incremented in the pseudocode in lines 318 and 321 of `Refresh()` .

Consider the block `b` written into `n.blocks[i]` by `TryAppend()` at Line 313. It is created by the `CreateBlock(i)` called at Line 311. 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 3. Thus the `CreateBlock(i-1)` that creates `b'` terminates before `CreateBlock(i)` that creates `b` is invoked. The value written into `b.`$\texttt{end}_{\texttt{left}}$ at Line 333 of `CreateBlock(i)` was read from `n.left.head-1` at Line 331 of `CreateBlock(i)`. Similarly, the value in `n.blocks[i-1].`$\texttt{end}_{\texttt{left}}$ was read from `n.left.head-1` during the call to `CreateBlock(i-1)`. Since `n.left.head` is non-decreasing `b'.`$\texttt{end}_{\texttt{left}} \leq$ `b.`$\texttt{end}_{\texttt{left}}$. The proof for $\texttt{end}_{\texttt{right}}$ is similar. ☐

Definition 5 (Subblock)**.** Block `b` is a *direct subblock* of `n.blocks[i]` if it is in `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].`$\texttt{end}_{\texttt{right}}$`]` . Block `b` is a subblock of `n.blocks[i]` if `b` is a direct subblock of `n.blocks[i]` or a subblock of a direct subblock of `n.blocks[i]`.

Corollary 6 (No Duplicates)**.** *If* `op` *is in* `n.blocks[i]` *then there is no* `j`$\neq$`i` *such that* `op`$\in$`ops(n.blocks[j])`.

Proof. Operation `op` is invoked only one time in an execution because every operations invoked is distinct. Since there is node `n` which `op` is in two different blocks of `n`, there is node $\texttt{n}'$ that is the lowest height node in the tree that contains `op` in two of its blocks `b1,b2`. By Definition 5, `b1` and `b2` have distinct subblocks(not only direct subblocks) and since `op` is in only one leaf block, then it cannot be in both `b1` and `b2`. □

Definition 7 (Superblock)**.** Block `b` is *direct superblock* of block `c` if c is a direct subblock of b. Block `b` is *superblock* of block `c` if `c` is a subblock of `b`.

Definition 8 (Operations of a block)**.** A leaf block `b` in a leaf represents `enqueue(x)` if $\texttt{b.element=x}\neq\texttt{null}$. Else if `b.element=null` b represents a `dequeue()`. The set of operations of block `b` are the operations in the subblocks of `b`. We denote the set of operations of block `b` by `ops(b)`.

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`. We also say `b` contains `op` if $\texttt{op}\in\texttt{ops(b)}$.

Definition 9. A block `b` in `n.blocks` is *established* at time t if `n.head`> index of `b` at time t. $EST_{n,\ t}$ is the set of established blocks of node `n` at time t.

Observation 10. *Once a block* `b` *is written in* `n.blocks[i]` *then* `n.blocks[i]` *never changes.*

Lemma 11. *Every block has at most one direct superblock.*

Proof. To show this we are going to refer to the way `n.blocks[]` is partitioned while propagating blocks up to `n.parent`. `n.CreateBlock(i)` merges the blocks in $\texttt{n.left.blocks[n.blocks[i-1].end}_{\texttt{left}}\texttt{..n.blocks[i].end}_{\texttt{left}}\texttt{]}$ and $\texttt{n.right.blocks[n.blocks[i-1].end}_{\texttt{right}}\texttt{..n.blocks[i]}$ (Lines 331, 332). Since $\texttt{end}_{\texttt{left}}\texttt{,end}_{\texttt{right}}$ are non-decreasing ($\texttt{n.blocks[i].end}_{\texttt{left|right}}\texttt{>n.blocks[i-1].end}_{\texttt{left|right}}$), so the range of the subblocks of `n.blocks[i]` which is $\texttt{(n.blocks[i-1].end}_{\texttt{dir}}\texttt{+1..n.blocks[i].end}_{\texttt{dir}}\texttt{)}$ does not overlap with the range of the subblocks of `n.blocks[i-1]`. □

Lemma 12 (establishedOrder)**.** *If time* $t <$ *time* t', *then* $\texttt{ops(EST}_{\texttt{n, t}}\texttt{)}\subseteq\texttt{ops(EST}_{\texttt{n, t}'}\texttt{)}$.

Proof. Blocks are only appended (not modified) with CAS to `n.blocks[n.head]` and `n.head` is non-decreasing, so the set of operations in established blocks of a node can only grow. □

useless?

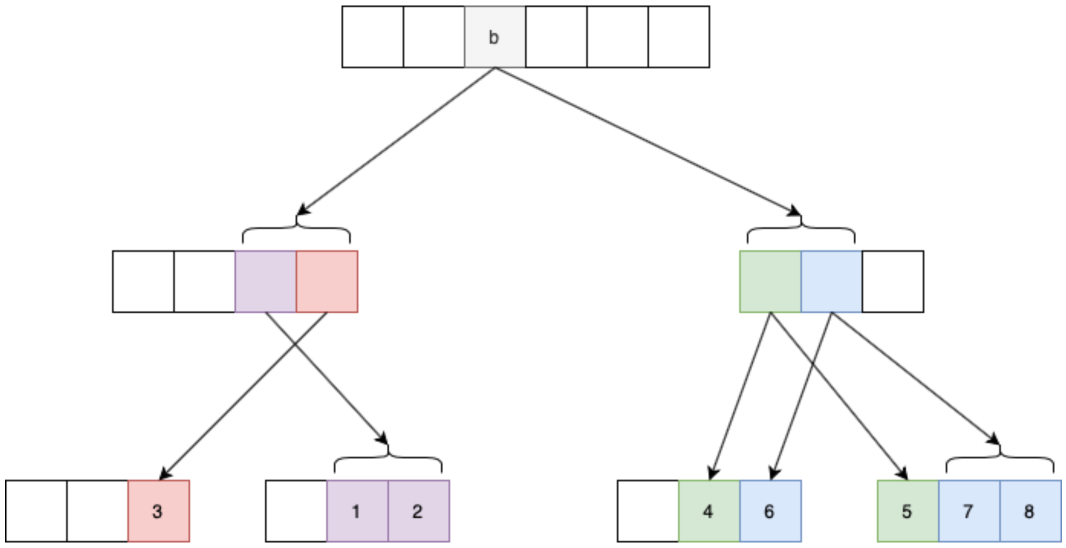


Figure 16: Order of elements in b: elements in leaves are ordered with numerical order in the drawing.

► Processes are numbered from 1 to p and leaves of the tree are assigned from left to right. We will show in Lemma 23 that there is at most one operation from each process in a given block.

Definition 13 (Ordering of operations inside the nodes).

- The prefix of an operation *op* in the sequence of operations S is the sequence of operations strictly before *op*.
- $E(n,b)$ is the sequence of enqueue operations in `ops(n.blocks[b])` defined recursively as follows. $E(leaf,b)$ is the single enqueue operation in `ops(leaf.blocks[b])` or an empty sequence if `leaf.blocks[b].num`$_{\texttt{enq}}$`=0`. If `n` is an internal node, then

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

- $E_i(n,b)$ 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.blocks[b])` defined recursively as follows. $D(leaf,b)$ is the single dequeue operation in `ops(leaf.blocks[b])` or an empty sequence if `leaf.blocks[b].num`$_{\texttt{deq}}$`=0`. If `n` is an internal node, then

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

- $D_i(n,b)$ 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 14 (Linearization). $L = E(root,1).D(root,1).E(root,2).D(root,2).E(root,3).D(root,3)$...

► In the non-root nodes, we only need ordering of enqueues and dequeues among the operations of their own type. Since `GetENQ()` only searches among enqueues and `IndexDEQ()` works with dequeues.

Lemma 15 (trueRefresh)**.** *Let t_i be the time an instance* `R` *of* `n.Refresh()` *is invoked and t_t be the time it terminates. If the* `TryAppend(new, s)` *of* `R` *returns* `true`*, then* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$.

Proof. Since `TryAppend` returns true a block `new` is written into `n.blocks[h]` in Line 313.

We show $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)}\subseteq\texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$. Let `h` be the value `n.Refresh()` reads from `n.head` at line 310, $\texttt{h}_{\texttt{left,i}}$ be the value of `n.left.head` at $\texttt{t}_\texttt{i}$ and $\texttt{h}_{\texttt{left,read}}$ be the value read from `n.left.head-1` at line 331. $\texttt{end}_{\texttt{left}}$ field of the block returned by `CreateBlock(i)` is $\texttt{h}_{\texttt{left,read}}$. By lines 332 and 331 the `new` block in `n.blocks[h]` contains $\texttt{n.left.blocks[n.blocks[h-1].end}_{\texttt{left}}\texttt{+1..h}_{\texttt{left,read}}\texttt{]}$. Since `left.head` is read after $\texttt{t}_\texttt{i}$ then $\texttt{h}_{\texttt{left,read}}\texttt{>}\texttt{h}_{\texttt{left,i}}$ which means $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)}\subseteq\texttt{ops(n.left.blocks [0..h}_{\texttt{left,read}}\texttt{])}$. After the successful `TryAppend` in line 313 we know all blocks in $\texttt{n.left.blocks[0..h}_{\texttt{left,read}}\texttt{-1}$ are subblocks of `n.blocks[0..h]` by the definition of subblock. At $\texttt{t}_\texttt{t}$ we have `n.head>h` by Lemma 4. So `n.blocks[1..h]` are in $\texttt{EST}_{\texttt{n},\texttt{t}_\texttt{t}}$ by definition of `EST`. Note that after line 321 we are sure that the `head` is incremented by Lemma2) which means `n.head=h+1` at t_t so the new block is established at t_t and the new block contains the new operations which is what we wanted to show. The proof for $\texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)}\subseteq\texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$is the same.

□

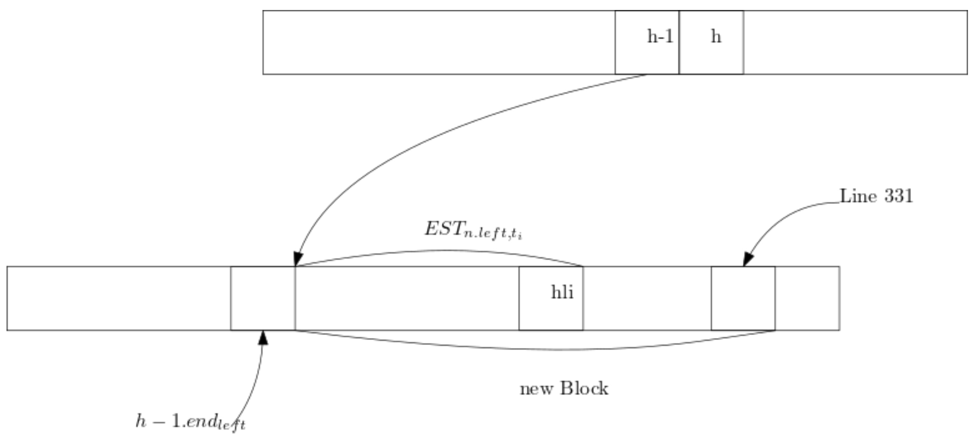


Figure 17: New established operations of the left child are in the new block.

Lemma 16 (Stronger True Refresh)**.** *Let t_i be the time an instance of* `n.Refresh()` *read the head (Line 310) and t_t be the time its* `TryAppend(new, s)` *terminates with and returns* `true` *(Line 313). We have* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(n.blocks)}$.

Definition 17. An instance of `Refresh()` is successful iff its `TryAppend(new, s)` terminates with and returns `true`.

Definition 18. Let ${}_{R_1}t$ be the time R_1 is invoked and t_{R_2} be the time R_2 terminates. ${}_{line}t$ is the immediate time before running Line *line*. t_{line} is the immediate time after running Line *line*. ${}_{line}t^{op}$ is the immediate time before running Line *line* of operation *op*. t^{op}_{line} is the immediate time after running Line *line* of operation *op*.

Lemma 19 (Double Refresh)**.** *Consider two consecutive instances R_1, R_2 of* `Refresh()` *on internal node* `n` *by a process p. If R_1 and R_2 both fail and return false, then we have* $\texttt{ops(EST}_{\texttt{n.left, }{}_{R1}\texttt{t}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, }{}_{R1}\texttt{t}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_{R2}}\texttt{)}$.

Proof.

If R_2 reads some value greater than $i+1$ in Line 310 it means a successful instance of `Refresh()` performed its Line 310 after $t^{R_1}_{310}$ and finished its Line 318 or 321 before $t^{R_2}_{310}$, from Lemma 16 by the end of this instance $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)}$ has been propagated.

Let R_1 read i and R_2 read $i+1$ from Line 310. As R_2's `TryAppend()` returns `false`, there is another successful instance R'_2 of `n.Refresh()` that has done `TryAppend()` successfully into `n.blocks[i+1]` before R_2 tries to append. Since R'_2 creates the block after reading the value $i+1$ from `n.head` (Line 310) and R_1 reads the value i from `n.head` and the `head`'s value is increasing by Lemma 4 then $t_{R2'\,310} > t_{R1\,310} >_{R_1} t$ (see Figure 18). By Lemma 16 after R'_2's `CAS` ($t^{R'_2}_{313}$) we have $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq$ `ops(n.blocks)`. Also by Lemma 2 on R_2 the value of `n.head` head is more than $i+1$ after R'_2 terminates, so the block appended by R'_2 to `n` is established by then (`n.head` $\geq i+2 > i+1$). To summarize, ${}_{R_1}t$ is before R'_2's read of `n.head` ($t^{R'_2}_{310}$) and R'_2's successful `CAS` is before R_2's termination. So, by Lemma 16, $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_2}\texttt{)}$. ☐

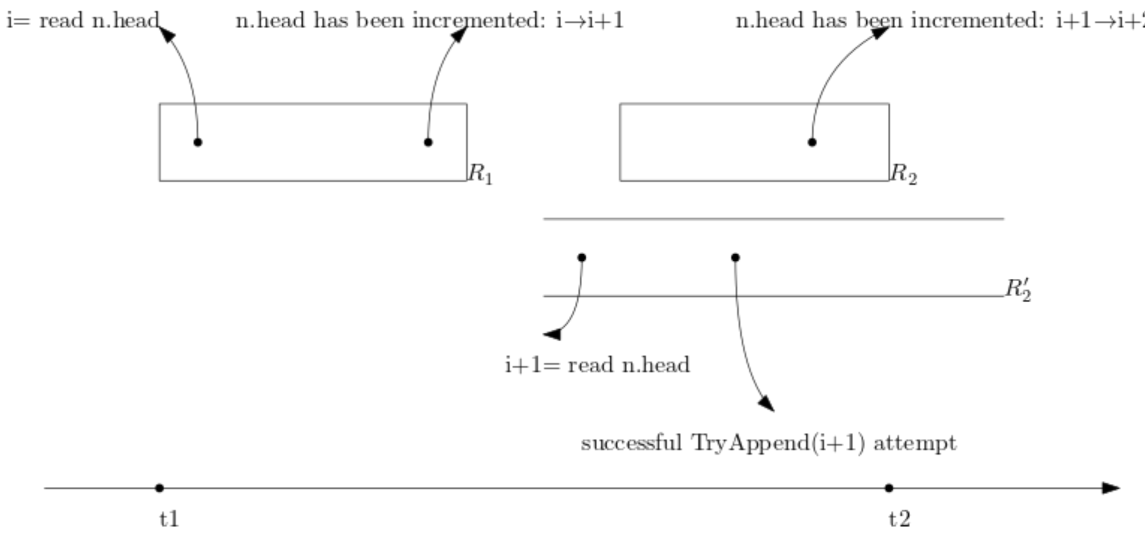


Figure 18: ${}_{R_1}t < t^{R_1}_{310} <$ incrementing `n.head` from i to $i+1 < t^{R'_2}_{310} < t^{R'_2}_{313} <$ incrementing `n.head` from $i+1$ to $i+2 < t_{R_2}$

Corollary 20. $\texttt{ops(EST}_{\texttt{n.left, }{}_{302}\texttt{t}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, }{}_{302}\texttt{t}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_{303}}\texttt{)}$

Proof. If the first `Refresh()` in line 302 returns `true` then by Lemma 15 the claim holds. Also if first `Refresh()` failed and the second `Refresh()` succeeded the claim still holds by Lemma 15. Finally, if both failed the claim is satisfied by Lemma 19. ☐

Corollary 21 (Propagate Step)**.** *All operations in* `n`*'s children's established blocks before running line 302 of a* `Propagate` *routine are guaranteed to be in* `n`*'s established blocks after line 303.*

Proof. If 302 or 303 succeed, the claim is true by Lemma 15. Otherwise Lines 302 and 303 satisfy the preconditions of Lemma 19. ☐

Corollary 22. *After* `Append(blk)` *finishes* `ops(blk)`$\subseteq$`ops(root.blocks[x])` *for exactly one* `x`*.*

Proof. After `Append(blk)`'s termination, `blk` is in `root.blocks` since `blk` is established in the leaf it has been added to. By applying Lemma 21 inductively it is propagated up to the root. Finally Lemma 6 shows only one block in the root contains `blk`. ☐

Lemma 23 (Block Size Upper Bound)**.** *Each block contains at most one operation of each processs.*

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. A process cannot invoke more than one operations concurrently, so op_1 has to be finished before op_2. By Corollary 22, before appending op_2 to the tree op_1 exists in every node on the path from p's leaf to the root, because op_1's `Append` is finished before op_2's `Append` starts. So, there is some block b' before b in n containing op_1. Existence of op_1 in b and b' contradicts Lemma 6. ☐

Lemma 24 (Subblocks Upperbound)**.** *Each block has at most p direct subblocks.*

Proof. The claim follows directly from Lemma 23 and the observation that each block appended to the tree contains at least one operation, due to the test on Line 312. We can also see the blocks in the leaves have exactly one operation in the `Enqueue()` and `Dequeue()` routines. ☐

Lemma 25 (Get correctness). *If* `n.blocks[b].num`$_{\texttt{enq}}$ $\geq$ `i` *then* `n.GetENQ(b,i)` *returns the* `element` *enqueued by* $E_i(n, b)$.

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 by the hypothesis, only `n.GetENQ(b,1)` can be called and only when `n.blocks[b]` contains an enqueue. At Line 403, `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 `n.GetENQ(b, i)` returns $E_i(n, b)$, assuming `n.child.GetENQ(subblock, i)` returns $E_i(n.child, b)$. We argue that Line 404 correctly decides whether the `i`th enqueue in `b`th block of internal node `n` is in the left child or right child subblocks of `n.blocks[b]`. From Definition 13 of $E(n, b)$ we know enqueue operations in a block are ordered from left to right and since the leaves of the tree are ordered by process id from left to right, thus operations from the left subblocks come before operations from the right subblocks in a block (See Figure 19). Furthermore the `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 `i` is greater than `b.num`$_{\texttt{enq-left}}$. Otherwise we should search for the ith enqueue in the left child. By definition 5 and 8 we need to search in subblocks of `n.blocks[b]` from the range `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` $\cup$ `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`.

If the ith `enqueue` of `n.blocks[b]` is in the left child it would be ith enqueue in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` by Definition 5. Also, we know there are $eb = n.blocks[b-1].sum_{enq-left}$ enqueues in the blocks before this range, 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[`$b'-1$`].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 `BSearch` returns the index of the subblock containing $E_i(n, b)$.

Otherwise the enqueue we are looking for is in the right child. Then, there are `n.blocks[b].num`$_{\texttt{enq-left}}$ enqueues ahead of it in `n.blocks[b]` but not in `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`. So we need to search for `i-n.blocks[b].num`$_{\texttt{enq-left}}$`+` `n.blocks[b-1].sum`$_{\texttt{enq-right}}$ (Line 409). Other parameters for the left child are chosen similarly to the way they were chosen for 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. □

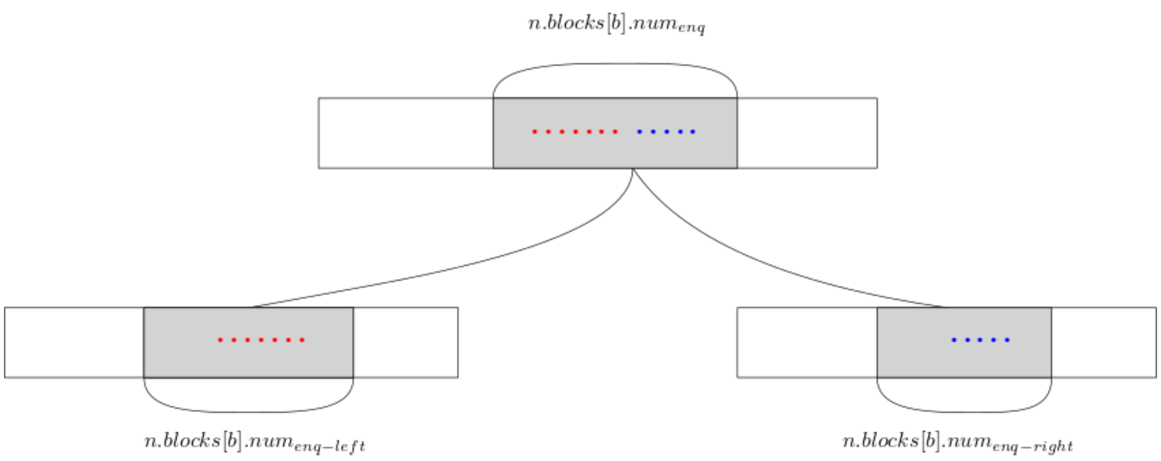


Figure 19: The number and ordering of the enqueue operations propagated from the left and the right child to `n.blocks[b]`. Enqueue operations from the left subblocks (colored red), are ordered before the enqueue operations from the right child (colored blue).

Lemma 26 (DSearch correctness)**.** *If* `root.blocks[end].sum`$_{\texttt{enq}}$ $\geq$ `e`*,* `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 Lemma we know `sum`$_{\texttt{enq}}$ fields of nfroot.blocks[] are sorted in a non-decreasing order. Since `root.blocks[0].sum`$_{\texttt{enq}}$`=0` and there is a `block` in the root with `sum`$_{\texttt{enq}}$ value greater than `e`, so there is a `b` that `root.blocks[b].sum`$_{\texttt{enq}}$ $\geq e$ but `root.blocks[b-1].sum`$_{\texttt{enq}}$ $< e$. This block contains $E_i(root, b)$ and the search on Line 802-806 will eventually reach the `b`.

□

Lemma 27 (DSearch Analysis)**.** *Assume* `root.blocks[end].sum`$_{\texttt{enq}}$ $\geq$ `e` *and* $E_e(root)$*'s* `element` *is the response to some* `Dequeue()` *operation in* `root.blocks[end]`*, then* `DSearch(e, end)` *takes* Θ(log`root.blocks[b].size + root.blocks[end].size`) *steps.*

Proof. First we show `end` $-$ `b` $\leq 2 \times \big($`root.blocks[b].size` $+$ `root.blocks[end].size` $+ 1\big)$. From line 312, we know that `num` field of the every block in the tree is greater than 0. So, each block in `root.blocks[b..end]` contains at least one `Enqueue` or at least one `Dequeue`. Suppose there were more than `root.blocks[b].size` `Dequeue`s in `root.blocks[b+1..end-1]`. Then the element in the queue which is the response to the `Dequeue()` would become dequeued at some point after `blocks[b]`'s last operations and before `root.blocks[end]`'s first operation. Which means the response to to a `Dequeue` in `root.blocks[end]` could not be in $E(n, b)$. Furthermore since the size of the queue would become `root.blocks[end].size` after the operations of `root.blocks[end]`, there cannot be more than `root.blocks[b].size + root.blocks[end-1].size` `Enqueue`s in `root.blocks[b+1..end-1]`. Because there can be at most `root.blocks[b].size` `Dequeue`s and the final size of the queue is `root.blocks[end-1].size` . Overall there can be at most 2×`root.blocks[b].size + root.blocks[end].size` operations in `root.blocks[b+1..end-1]` and since each block size is ≥ 1 thus there are at most 2×`root.blocks[b].size + root.blocks[end].size` blocks in between `root.blocks[b]` and `root.blocks[end]`. So `end-b` $\leq 2 \times$ `root.blocks[b].size` $+$`root.blocks[end].size` $+ 1$. See Figure 20.

Now that we know there are at most `root.blocks[b].size +root.blocks[end].size` blocks in between `root.blocks[b]` and `root.blocks[end]` then with doubling search in $\Theta\big($log(`root.blocks[b].size +root.blocks[end].size`)$\big)$ steps we reach `start=c` that the `root.blocks[c].sum`$_{\texttt{enq}}$ is less than `e` and `end-c` is not more than $2 \times 2 \times \big($`root.blocks[b].size +root.blocks[end].size` . Beause otherwise, then `(end-c)/2` satisfied the `root.blocks[(end-c)/2].sum`$_{\texttt{enq}}$ $<$ `e`. In line 804 the differnece between `end` and `start` is doubled. See Figure 20.

After computing `b`, the value `i` is computed via the definition of `sum`$_{\texttt{enq}}$ in constant time (Line 807). So the routine non constant part is the binary search which takes Θ(log`root.blocks[b].size +root.blocks[end].size`) steps from the first paragraph.

□

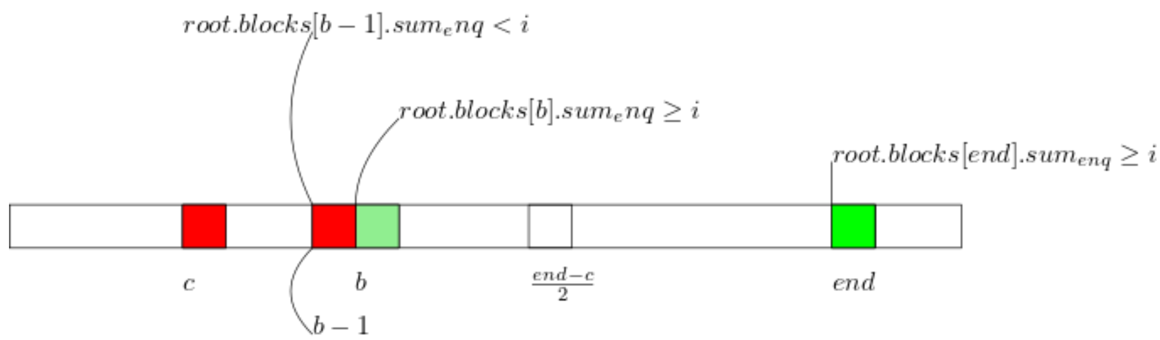


Figure 20: Distance relations between b, c, end

Lemma 28. *Let n.propagates be the number of groups of blocks that have been propagated from node* `n` *to its parent (successful* `n.parent.Refresh()`*). We have* $\texttt{num}_{\texttt{propagated}} \leq$ *n.propagates* $\leq \texttt{num}_{\texttt{propagated}}$`+p`*.* `p` *is the number of processes.*

Proof. $\texttt{num}_{\texttt{propagated}}$ is incremented after propagating (Line 316). Since maybe some process falls sleep before incrementing $\texttt{num}_{\texttt{propagated}}$ it may be behind by `p`. ☐

Lemma 29. `super[]` *preserves order from child to parent; i.e. if in node* `n` *block* `b` *is before* `c` *then* `b.group` $\leq$ `c.group`

Proof. Line 329. Since $\texttt{num}_{\texttt{propagated}}$ is increasing. ☐

Lemma 30. *Let* `b, c` *be in node* `n`*, if* `b.group` $\leq$ `c.group` *then* `super[b.group]` $\leq$ `super[c.group]`

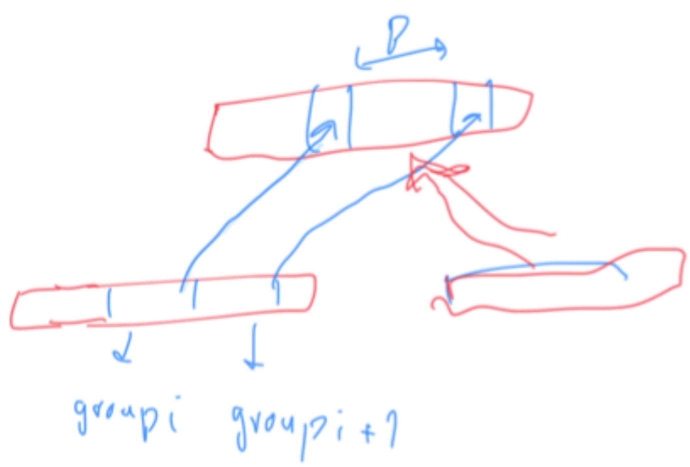
Proof. Line 315. ☐

Lemma 31. *The number of the* `blocks` *with* `group=i` *in a node is* $\leq p$*.*

Proof. For the sake of simplicity we assumed all the blocks are propagated from the left child. ☐

Lemma 32. `super[i+1]-super[i]`$\leq p$

Proof. In a Refresh with successful CAS in line 46, `super` and `counter` are set for each child in lines 48,49. Assume the current value of the counter in node `n` is `i+1` and still `super[i+1]` is not set. If an instance of successful `Refresh(n)` finishes `super[i+1]` is set a new value and a block is added after `n.parent[sup[i]]`. There could be at most p successful unfinished concurrent instances of `Refresh()` that have not reached line 49. So the distance between `super[i+1]` and `super[i]` is less than p. ☐



Lemma 33 (super property)**.** *If* `super[i]` $\neq$ `null` *in node* `n`*, then* `super[i]` *is the index of the superblock of a block with* `time=i` *in* `n.parent.blocks`*.*

Lemma 34. *Superblock of* `b` *is within range* $\pm 2p$ *of the* `super[b.group]`*.*

Proof. `super[i]` is the index of the superblock of a block containing block b, followed by Lemma 33. `super(b)` is the real superblock of b. `super(t]` is the index of the superblock of the last block with time `t`. If `b.time` is `t` we have:

$$super[t] - p \leq super[t-1] \leq super(t-1] \leq super(b) \leq super(t+1) \leq super(t+1] \leq super[t] + p$$

□

Lemma 35. *Search in each level of* `IndexDeq()` *takes* $O(\log p)$ *steps.*

Proof. Show preconditions are satisfied and the range is p. □

Lemma 36 (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 337, `super` are set before incrementing in lines 315,316. 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 34if we search in the range p, we can find the superblock.

□

Lemma 37 (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 36 (Line 418). After that the order in $D(n.parent, superblock)$ is computed. Note that by Lemma 23 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 21). 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 13). □

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

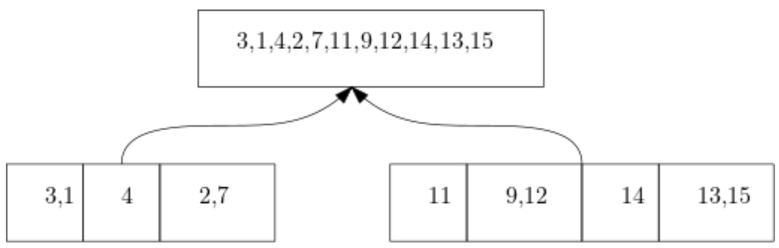


Figure 21: Relation of ordering of operations of a block from its subblocks

Definition 38. 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 39. In an execution on a queue, the dequeue operations that return some value are called *non-null dequeues.*

Observation 40. *In a sequential execution on a queue, kth non-null dequeue returns the* `element` *of kth enqueue.*

Lemma 41. `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 13 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].`$\texttt{sum}_{\texttt{enq}}$ 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].`$\texttt{sum}_{\texttt{enq}}$`-` `root.blocks[b].`$\texttt{sum}_{\texttt{deq}}$ (Line 342). See Table 4 for an example of running some blocks of operations on an empty queue. ☐

Lemma 42 (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.`$\texttt{sum}_{\texttt{enq}}$`-b.size`

Proof. There are `b.`$\texttt{sum}_{\texttt{enq}}$ 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.`$\texttt{sum}_{\texttt{enq}}$`-b.size` . The correctness of the `block.size` field is shown in Lemma 41. ☐

Lemma 43. `R(`$\texttt{D}_{\texttt{root,b,i}}$`)` *is* `null` *iff* `root.blocks[b-1].size + root.blocks[b].`$\texttt{num}_{\texttt{enq}}$`- i <0`.

Lemma 44 (Computing Response)**.** `FindResponse(b,i)` *returns* `R(`$\texttt{D}_{\texttt{root,b,i}}$`).element`.

Proof. First note that by Definition 13 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 13 and sum_{enq}. $D_{root,b,i}$ returns `null` if `root.blocks[b-1].size +` `root.blocks[b].`$\texttt{num}_{\texttt{enq}}$`- i <0` by Lemma 43 (Line 220). Otherwise if it is d'th non-null dequeue in L it returns d'th enqueue by Observation 40 . By Lemma 42 there are `root.blocks[b-1].`$\texttt{sum}_{\texttt{enq}}$ `- 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 22.

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). ☐

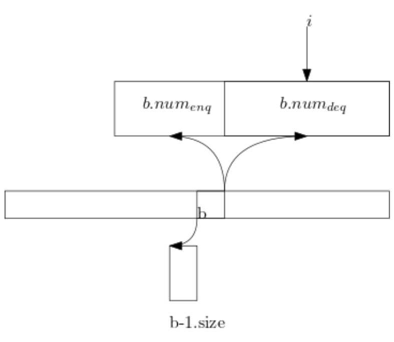


Figure 22: The position of $E_i(root, b)$.

	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 45 (Main)**.** *The queue implementation is linearizable.*

Proof. We choose L in Definition 13 to be linearization ordering of operations and prove if we linearize operations as L the queue works consistently. □

Lemma 46 (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 $\texttt{op}_1$`.Append()` is terminated before $\texttt{op}_2$`.Append()` starts. From Lemma 6 $\texttt{op}_1$ is in `root.blocks` before $\texttt{op}_2$ propagates so $\texttt{op}_1$ is linearized before $\texttt{op}_2$ by Definition 13.

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 47 (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 6. The time op_1 ends it is in root, before op_2, by Definition 13 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 44 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 25 we know that the returning response is the computed index element.

□

Lemma 48 (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 34) 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 26, which is O(log size of the queue when `enqueue` is invoked) + log size of the queue when `dequeue` is invoked). Each search in `GetEnq()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma 24), 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 49 (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 313 and one in Lines 318 or 321. □

3.4 Garbage Collection or Getting rid of the infinite Arrays

4 Using Queues to Implement Vectors

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

5 Conclusion

possible directions for work

Maybe Stacks

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