

1 Pseudocode

Algorithm Fields description

◇ Shared

- *Tree* `tree` : A binary tree of Nodes. `root` is a pointer to the root node.

◇ Local

- **Node* `leaf` : a pointer to the process's leaf in the tree.

◇ Structures

► Node

- **Node* `left`, `right`, `parent` : initialized when creating the tree.
- *BlockList* `blocks` implemented with an array.
- *int* `size`= 1: #blocks in `blocks`.
- *int* `numpropagated`= 0 : # groups of blocks that have been propagated from the node to its parent. Since it is incremented after propagating, it may be behind by 1.
- *int[]* `super`: `super[i]` stores an approximate index of the superblock of the blocks in `blocks` whose `group` field have value `i`.

► Leaf extends Node

- *int* `lastdone`
Stores the index of the block in the root such that the process that owns this leaf has most recently finished the. A block is finished if all of its operations are finished. `enqueue(e)` is finished if `e` is returned by some `dequeue()` and `dequeue()` is finished when it computes its response. *put the definitions before the pseudocode*

► Block

▷ 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`. *put the definitions before the pseudocode*

- *int* `group` : the value read from `numpropagated` when appending this block to the node.

► LeafBlock extends Block

- *Object* `element` : Each block in a leaf represents a single operation. For enqueue operations `element` is the input of the enqueue and for dequeue operations it is `null`.
- *Object* `response` : stores the response of the operation in the `LeafBlock`.
- *int* `sumenq`, `sumdeq` : # enqueue, dequeue operations in the prefix for the block

► InternalBlock extends Block

- *int* `endleft`, `endright` : index of the last subblock of the block in the left and right child
- *int* `sumenq-left` : # enqueue operations in the prefix for `left.blocks[endleft]`
- *int* `sumdeq-left` : # dequeue operations in the prefix for `left.blocks[endleft]`
- *int* `sumenq-right` : # enqueue operations in the prefix for `right.blocks[endright]`
- *int* `sumdeq-right` : # dequeue operations in the prefix for `right.blocks[endright]`

► RootBlock extends InternalBlock

- *int* `length` : length of the queue after performing all operations in the prefix for this block
 - *counter* `numfinished` : number of finished operations in the block
-

Variable naming:

- `bop`: index of the block containing operation `op`
- `rop`: rank of operation `op` i.e. the ordering among the operations of its type according to linearization ordering

Abbreviations:

- `blocks[b].sumx`=`blocks[b].sumx-left`+`blocks[b].sumx-right` (for `b`≥0 and `x` ∈ {enq, deq})
- `blocks[b].sum`=`blocks[b].sumenq`+`blocks[b].sumdeq` (for `b`≥0)
- `blocks[b].numx`=`blocks[b].sumx`-`blocks[b-1].sumx`
(for `b`>0 and `x` ∈ {∅, enq, deq, enq-left, enq-right, deq-left, deq-right}, `blocks[0].numx`=0)

Algorithm Queue

```
201: void ENQUEUE(Object e)  ▷ Creates a block with element e and appends
    it to the tree.
202:   block newBlock= NEW(LeafBlock)
203:   newBlock.element= e
204:   newBlock.sumenq= leaf.blocks[leaf.size].sumenq+1
205:   newBlock.sumdeq= leaf.blocks[leaf.size].sumdeq
206:   leaf.APPEND(newBlock)
207: end ENQUEUE

208: Object DEQUEUE()
209:   block newBlock= NEW(LeafBlock)          ▷ Creates a block
    with null value element, appends it to the tree, computes its order among
    operations, then computes and returns its response.
210:   newBlock.element= null
211:   newBlock.sumenq= leaf.blocks[leaf.size].sumenq
212:   newBlock.sumdeq= leaf.blocks[leaf.size].sumdeq+1
213:   leaf.APPEND(newBlock)
214:   return leaf.HELPDEQUEUE()
215: end DEQUEUE

216: <int, int> FINDRESPONSE(int b, int i)      ▷ Computes the rank and
    index of the block in the root of the enqueue that is the response of the ith
    dequeue in the root's bth block. Returns <-1,--> if the queue is empty.
217:   if root.blocks[b-1].length + root.blocks[b].numenq - i < 0 then
218:     return <-1,-->
219:   else
    ▷ We call the dequeues that
    return a value non-null dequeues. rth non-null dequeue returns the element
    of th rth enqueue. We can compute # non-null dequeues in the prefix for
    a block this way: #non-null dequeues= length - #enqueues. Note that the
    ith dequeue in the given block is not a non-null dequeue.
220:     renq= root.blocks[b-1].sumenq- root.blocks[b-1].length + i
221:     return <root.BSEARCH(sumenq, renq, root.FindMostRecentDone(),
    root.size), renq>
222:   end if
223: end FINDRESPONSE
```

Algorithm Node

```

301: void PROPAGATE()
302:   if not REFRESH() then
303:     REFRESH()
304:   end if
305:   if this is not root then
306:     parent.PROPAGATE()
307:   end if
308: end PROPAGATE

309: boolean REFRESH()
310:   s = size
311:   <new, npleft, npright> = CREATEBLOCK(s)
312:   if new.num==0 then return true
313:   else if blocks.tryAppend(new, s) then
314:     for each dir in {left, right} do
315:       CAS(dir.super[npdir], null, h+1)
316:       CAS(dir.numpropagated, npdir, npdir+1)
317:     end for
318:     CAS(size, s, s+1)
319:     return true
320:   else
321:     CAS(size, s, s+1)
322:     return false
323:   end if
324: end REFRESH

325: int BSEARCH(field f, int i, int start, int end)
326: end BSEARCH

327: <Block, int, int> CREATEBLOCK(int i)
328:   block newBlock = NEW(block)
329:   newBlock.group = numpropagated
330:   newBlock.order = i
331:   for each dir in {left, right} do
332:     indexlast = dir.size
333:     indexprev = blocks[i-1].enddir
334:     newBlock.enddir = indexlast
335:     blocklast = dir.blocks[indexlast]
336:     blockprev = dir.blocks[indexprev]
337:     thisdir = dir.numpropagated
338:     newBlock.sumenq-dir = blocks[i-1].sumenq-dir + blocklast.sumenq
339:     - blockprev.sumenq
340:     newBlock.sumdeq-dir = blocks[i-1].sumdeq-dir + blocklast.sumdeq
341:     - blockprev.sumdeq
342:   end for
343:   if this is root then
344:     newBlock.length = max(root.blocks[i-1].length + b.numenq -
345:                           b.numdeq, 0)
346:   end if
347:   return <b, npleft, npright>
348: end CREATEBLOCK

```

302: **firstRefresh**
 303: **secondRefresh**
 310: **readSize**
 314: **okcas**
 318: **incrementHead1**
 324: **incrementHead2**

311: \triangleright np_{left}, np_{right} are the values read from the children's num_{propagated} field.
 312: \triangleright The block contains nothing.
 337: \triangleright newBlock includes dir.blocks[index_{prev}+1..index_{last}].
 343: \triangleright Even if another process wins, help to increase the size. The winner might have fallen sleep before increasing size.

325: \rightsquigarrow Precondition: blocks[start..end] contains a block with field $f \geq i$
 326: \triangleright 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 $\geq i$.

Algorithm Node

↪ Precondition: $\text{blocks}[b].\text{num}_{\text{enq}} \geq i$

```
401: element GETENQ(int b, int i)
402:   if this is leaf then
403:     return blocks[b].element
404:   else if  $i \leq \text{blocks}[b].\text{num}_{\text{enq-left}}$  then                                ▷ i exists in the left child of this node
405:     subBlock= left.BSEARCH( $\text{sum}_{\text{enq}}$ , i,  $\text{blocks}[b-1].\text{end}_{\text{left}}+1$ ,  $\text{blocks}[b].\text{end}_{\text{left}}$ )    ▷ Search range of left child's subblocks of blocks[b].
406:     return left.GET( $i-\text{left.blocks}[\text{subBlock}-1].\text{sum}_{\text{enq}}$ , subBlock)
407:   else
408:     i=  $i-\text{blocks}[b].\text{num}_{\text{enq-left}}$ 
409:     subBlock= right.BSEARCH( $\text{sum}_{\text{enq}}$ , i,  $\text{blocks}[b-1].\text{end}_{\text{right}}+1$ ,  $\text{blocks}[b].\text{end}_{\text{right}}$ )    ▷ Search range of right child's subblocks of blocks[b].
410:     return right.GET( $i-\text{right.blocks}[\text{subBlock}-1].\text{sum}_{\text{enq}}$ , subBlock)
411:   end if
412: end GETENQ
```

↪ Precondition: b th block of the node has propagated up to the root and $\text{blocks}[b].\text{num}_{\text{enq}} \geq i$.

```
413: <int, int> INDEXDEQ(int b, int i)                                ▷ Returns the rank of  $i$ th dequeue in the  $b$ th block of the node, among the dequeues in the root.
414:   if this is root then
415:     return <b, i>
416:   else
417:     dir= (parent.left==n)? left: right                                ▷ check if a left or a right child
418:     superBlock= parent.BSEARCH( $\text{sum}_{\text{deq-dir}}$ , i,  $\text{super}[\text{blocks}[b].\text{group}]-p$ ,  $\text{super}[\text{blocks}[b].\text{group}]+p$ )
                                                                                               ▷ superblock's group has at most  $p$  difference with the value stored in super[].
419:     if dir is right then
420:       i+=  $\text{blocks}[\text{superBlock}].\text{sum}_{\text{deq-left}}$                                 ▷ consider the dequeues from the right child
421:     end if
422:     return this.parent.INDEXDEQ(superBlock, i)
423:   end if
424: end INDEX
```

Algorithm Root

```
501: Block FINDMOSTRECENTDONE
502:   for leaf l in leaves do
503:     max= Max(l.maxOld, max)
504:   end for
505:   return max                                                        ▷ This snapshot suffices.
506: end FINDMOSTRECENTDONE
```

appendEnd

pendStart

deqRest

Algorithm Leaf

```
601: void APPEND(block blk)                                ▷ Append is only called by the owner of the leaf.
602:     size+=1
603:     blk.group= size
604:     blocks[size]= blk
605:     parent.PROPAGATE()
606: end APPEND

607: Object HELPDEQUEUE()
608:     <bdeq, rdeq>= INDEXDEQ(leaf.size, 1)                ▷ r is the rank among the dequeues of the dequeue of the bdeqth block in the root containing.
609:     <benq, renq>= FINDRESPONSE(bdeq, rdeq)    ▷ renq is the rank of the enqueue whose element is the response to the dequeue in the block containing it and
        bdeq is the index of that block of it in the blocklist. If the response is null then rdeq is -1.
610:     if renq==-1 then
611:         output= null
612:         root.blocks[bdeq].numfinished.inc()                ▷ shared counter
613:         if root.blocks[bdeq].numfinished==root.blocks[bdeq].num then
614:             lastdone= bdeq
615:         end if
616:     else
617:         output= GETENQ(benq, renq)                        ▷ getting the reponse's element.
618:         root.blocks[benq].numfinished.inc()
619:         root.blocks[benq].numfinished.inc()
620:         if root.blocks[bdeq].numfinished==root.blocks[bdeq].num then
621:             lastdone= bdeq
622:         else if root.blocks[benq].numfinished==root.blocks[benq].num then
623:             lastdone= benq
624:         end if
625:     end if
626:     return output
627: end DEQUEUE

628: void HELP                                                ▷ Helps pending operations
629:     last= l.size-1                                        ▷ l.blocks[last] can not be null because size increases after appending, see lines 603-602.
630:     if l.blocks[last].element==null then                ▷ operation is dequeue
631:         l.blocks[last].response= l.HELPDEQUEUE()
632:     end if
633: end HELP
```

appendStartEnd
603-602

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 i th 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 `endleft`, `endright` pointers to the first block of the corresponding children.

◇ *root implementation*

```
701: boolean TRYAPPEND(block blk, int n)                                ▷ adds block b to the root.blocks[n]
702:   if root.size%p2==0 then                                          ▷ Help every often p2 operations appended to the root.
703:     for leaf l in tree.leaves do
704:       l.Help()
705:     end for
706:   end if
707:   blk.numfinished = 0
708:   return CAS(blocks[n], null, blk)
709: end TRYAPPEND
```

◇ *Array implementation*

`blocks[]`: array of blocks

```
710: boolean TRYAPPEND(block blk, int n)
711:   return CAS(blocks[n], null, blk)
712: end TRYAPPEND
```

Algorithm Yet to decide how to handle.

```
801: response FALLBACK(op i)                                          ▷ how to use as exception handling? by adding try catch in all the methods reading the root?
802:   if root.blocks.get(numenq), i is null then                      ▷ this enqueue was already finished
803:     return this.leaf.response(block.order)
804:   end if
805: end FALLBACK
```

2 Proof of Linearizability

TEST As a temporary test I have changed the name of `n.size` to `n.head` here, other options are `n.head` and `n.lastBLock` but they might be confusing since we have used them before. Fix the logical order of definitions (cyclic references).

TODO Fallback safety lemmas. Some parts are obsolete.

Questions When I write the lemmas since every claim in my mind is correct maybe I miss some fact that need proof or maybe I refer to some lemmas that are generally correct but not needed for the linearizability proof. Is lemma 7 necessary? Is lemma 13 induced trivially from lemma 8?

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

TEST How to merge notions of blocks and operations? block $b \sqsubseteq$ block c means b is subblock of c . block $b \in$ set B means b is in B . Merge these two to have shorter formulaes.

Definition 1 (Block). A block is an object storing some statistics described in Algorithm Queue. It implicitly shows a set of operations. The set of operations of block b are the operations in the leaf subblocks of b . We show the set of operations of block b , set of blocks B by $\text{ops}(b)$, $\text{ops}(B)$. We also say b contains op if $op \in \text{ops}(b)$.

Definition 2 (Order). 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 b 's index is smaller than b' 's.

Definition 3 (Subblock). Block b is a *direct subblock* of `n.blocks[i]` if it is $\in \text{n.left.blocks}[\text{n.blocks[i-1].end}_{\text{left}}+1 \dots \text{n.blocks[i].end}_{\text{left}}] \cup \text{n.right.blocks}[\text{n.blocks[i-1].end}_{\text{right}}+1 \dots \text{n.blocks[i].end}_{\text{right}}]$. Block b is a subblock of a `n.blocks[i]` if it is a direct subblock of it or subblock of a direct subblock of it. Block b is direct superblock of block c if c is direct subblock of b .

For simplicity we say block b is propagated to node n or to a set of blocks S if b is in `n.blocks` or S or is a subblock of a block in `n.blocks` or S .

Definition 4. Block b in `n.blocks` is *Established* at time t if `n.head` is greater than index of b at time t . Block b is in $\text{EST}_{n, t}$ if b is a subblock of b' in `n.blocks` such that b' is established at time t .

head **Observation 5.** Once block b is written in `n.blocks[i]` then `n.blocks[i]` never changes.

headProgress **Lemma 6** (headProgress). `n.head` is non-decreasing over time and `n.blocks[i].endleft, right` \geq `n.blocks[i-1].endleft, right`.

Proof. Induced trivially from the pseudocode since `n.head` is only incremented. `n.blocks[i].` Also `endleft, right` are greater than or equal the previous values in the `CreateBlock(i)` code (Lines lastLine, prevLine ~~177~~). □

Lemma 7. Every block has most one direct superblock.

Proof. To show this we are going to refer to the way `n.blocks[]` is partitioned while propagating up to `n.parent`. `n.CreateBlock(i)` merges the blocks in `n.left.blocks[n.blocks[i-1].endleft .. n.blocks[i].endleft]` and `n.right.blocks[n.blocks[i-1].endright .. n.blocks[i].endright]` (Lines lastLine, pr ~~177~~). From we know that `endleft, endright` are non decreasing, so the range of the subblocks of `n.blocks` are disjoint. □

headPosition **Lemma 8** (headPosition). If the value of `n.head` is h then, `n.blocks[i]=null` for $i>h$ and `n.blocks[i]≠null` for $i<h$.

Proof. The value of `n.head` is modified only in lines incrementHead1, incrementHead2 ~~177~~ and a value is written into `n.blocks[head]` is only in Line 313. It's obvious that writing into `n.blocks[head]` does not change the state of the claim of the lemma and if it was right before 312 it is going to be right after 312. We show in both cases of the successful 313 or not the claim holds after the increment lines of `n.head`. In the first case the process applied a successful `TryAppend(new, h)` in line okcas ~~314~~, which means `n.blocks[h]` is not null anymore. If the claim was held before it is going to be held after the incrementHead1 ~~318~~. Since wether incrementHead1 ~~318~~ returns true or false it means `n.head` has been incremented. The failiure case is also the same since it means `n.blocks[head]` is written by some process. □

Lemma 9 (establishedOrder). *If time $t < \text{time } t'$, then $\text{ops}(\text{EST}_n, t) \subseteq \text{ops}(\text{EST}_n, t')$.*

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 grows. \square

Lemma 10 (trueRefresh). *Let t_i be the time `n.Refresh()` is invoked and t_t be the time it is terminated. Suppose `n.Refresh()`'s `TryAppend(new, s)` returns `true`, then $\text{ops}(\text{EST}_{n.\text{left}}, t_i) \cup \text{ops}(\text{EST}_{n.\text{right}}, t_i) \subseteq \text{ops}(\text{EST}_n, t_t)$.*

Proof. From Lemma [9](#) we know that $\text{ops}(\text{EST}_n, t_i) \subseteq \text{ops}(\text{EST}_n, t_t)$. So it remains to show $\text{ops}(\text{EST}_{n.\text{left}}, t_i) \cup \text{ops}(\text{EST}_{n.\text{right}}, t_i) - \text{ops}(\text{EST}_n, t_i) \subseteq \text{ops}(\text{EST}_n, t_t)$ which we call new operations $\subseteq \text{ops}(\text{EST}_n, t_t)$. If `TryAppend` returns true a block is appended to `n`. From the code of the `CreateBlock` this block includes the established blocks in `n`'s children at t_i . Since the head in createblock is read after t_i . So the new operations are in the block appended to `n`. \square

Lemma 11 (Double Refresh). *Consider two consecutive failed instances r_1, r_2 of `n.Refresh()` by some process. Let t_1 be the time r_1 is invoked and t_2 be the time r_2 terminated. After r_2 's `TryAppend` we have $\text{ops}(\text{EST}_{n.\text{left}}, t_1) \cup \text{ops}(\text{EST}_{n.\text{right}}, t_1) \subseteq \text{ops}(\text{EST}_n, t_2)$.*

Proof.

If Line 313 of r_1 or r_2 returns `true`, then the claim is held by Lemma [10](#). If not, then there is another successful instance of `n.Refresh()` r'_2 which has did `TryAppend` successfully into `n.blocks[i+1]`. In figure 1 we see why the block it is appending contains established block in the `n`'s children at t_i , since it create a block reading the head after t_1 . \square

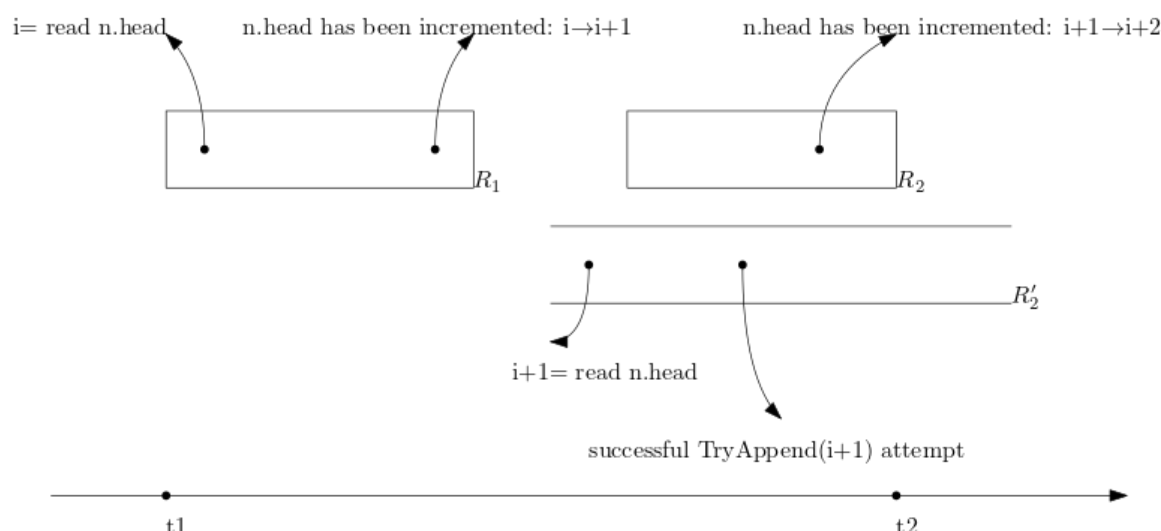


Figure 1: $t_1 < r_1$ reading head $<$ incrementing `n.head` from i to $i+1$ $<$ R'_2 reading head $<$ `TryAppend(i+1)` $<$ incrementing `n.head` from $i+1$ to $i+2$ $<$ t_2

Corollary 12 (Propagate Step). *All operations in `n`'s children's established blocks before line [302](#) are guaranteed to be in `n`'s established blocks after line [303](#).*

Proof. Lines [302](#) and [303](#) satisfy the preconditions of Lemma [11](#). \square

`CreateBlock()` reads blocks in the children that do not exist in the parent and aggregates them into one block. If a `Refresh()` procedure returns true it means it has appended the block created by `CreateBlock()` into the parent node's sequence. So suppose two `Refreshes` fail. Since the first `Refresh()` was not successful, it means another CAS operation by a `Refresh`, concurrent to the first `Refresh()`, was successful before the second `Refresh()`. So it means the second failed `Refresh` is concurrent with a successful `Refresh()` that assuredly has read block before the mentioned line 35. After all it means if any of the `Refresh()` attempts were successful the claim is true, and also if both fail the mentioned claim still holds.

append **Lemma 13** (No Duplicates). *If op is appended to $n.blocks[i]$ then after that there is no $j > i$ such that $op \in ops(n.blocks[j])$.*

Corollary 14. *After $Append(blk)$ finishes $ops(blk) \subseteq ops(root.blocks[x])$ for some x and only one x .*

blockSize **Lemma 15** (Block Size Upper Bound). *Each block contains at most one operation from each process (\forall process $p: \#operations\ of\ p \in ops(n.blocks[x]) \leq 1$).*

blocksBound **Lemma 16** (Subblocks Upperbound). *Each block has at most p direct subblocks.*

ordering **Definition 17** (Ordering of operations inside the nodes). \blacktriangleright Note that from Lemma ^{blockSize}15 we know there is at most one operation from each process in a given block.

- We call operations strictly before op in the sequence of operations S , prefix of the op .
- $E(n, i)$ is the sequence of enqueue operations $\in ops(n.blocks[i])$ ordered by their process id.
- $D(n, i)$ is the sequence of dequeue operations $\in ops(n.blocks[i])$ ordered by their process id.
- Order of the enqueue operations in n : $E(n) = E(n, 1).E(n, 2).E(n, 3)...$
- Order of the dequeue operations in n : $D(n) = D(n, 1).D(n, 2).D(n, 3)...$
- Linearization: $L = E(root, 1).D(root, 1).E(root, 2).D(root, 2).E(root, 3).D(root, 3)...$

Note that in the non-root nodes we only order enqueues and dequeues among the operations of their own type. Since $GetENQ()$ only works on enqueues and $IndexDEQ()$ works on dequeues.

search **Lemma 18** (Search Ranges). *Preconditions of all invocation of $BSearch$ are satisfied.*

get **Lemma 19** (Get correctness). $n.GetENQ(b, i)$ returns the i th Enqueue in $E(n, b)$.

help **Lemma 20** (help). *After that $TryAppend()$ who is helping finishes, prefix for the blocks of $root.blocks[root.FindMostRecentDone]$ are done.*

Lemma 21 (Index correctness). $n.Index(b, i)$ returns the rank in $D(root)$ of the i th Dequeue in $D(n, b)$.

superBlock **Lemma 22** (Computing SuperBlock). *After computing line ^{computeSuper}418 of $n.IndexDEQ(b, i)$, $superblock$ contains the i th dequeue in the b th block of the node n .*

computeHead **Lemma 23** (Computing Queue's Head). *Let S be the state of an empty queue if the operations in prefix in L of i th dequeue in $D(root, b)$ are applied on it. $FindResponse()$ returns the index in $E(root, b)$ of the enqueue that is the head in S . If the queue is empty in S it returns $<-1, -->$.*

superCounter **Lemma 24** (Validity of super and counter). *If $super[i] \neq null$, then $super[i]$ in node n is the index of the superblock of a block with $time=i \pm p$.*

rootRange **Lemma 25** (Root search range). $root.size - root.FindMostRecentDone()$ is $O(p^2 + q)$, which p is $\#$ processes and q is the length of the queue.

Theorem 26 (Main). *The queue implementation is linearizable.*

Lemma 27 (Time analysis). $n.GetEnq(b, i)$, $n.Index(b, i)$ take $O(\log^2 p)$ steps. Search in the root may take $O(\log Q + p^2)$ steps. Helping is done every p^2 block appended to the root and takes $p \times \log^2 p$ steps. Amortized time consumed for helping by each process is $O(\log^2 p)$.