1 Pableudocode

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* $\text{num}_{\text{propagated}}= 0$: # 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* $\text{last}_{\text{done}}$

  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 $\text{num}_{\text{propagated}}$ 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* sum_{enq}, sum_{deq} : # enqueue, dequeue operations in the prefix for the block

▶ *InternalBlock* extends *Block*

- *int* end_{left}, $\text{end}_{\text{right}}$: index of the last subblock of the block in the left and right child

- *int* $\text{sum}_{\text{enq-left}}$: # enqueue operations in the prefix for `left.blocks[`end_{left}`]`

- *int* $\text{sum}_{\text{deq-left}}$: # dequeue operations in the prefix for `left.blocks[`end_{left}`]`

- *int* $\text{sum}_{\text{enq-right}}$: # enqueue operations in the prefix for `right.blocks[`$\text{end}_{\text{right}}$`]`

- *int* $\text{sum}_{\text{deq-right}}$: # dequeue operations in the prefix for `right.blocks[`$\text{end}_{\text{right}}$`]`

▶ *RootBlock* extends *InternalBlock*

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

- *counter* $\text{num}_{\text{finished}}$: number of finished operations in the block

---

Variable naming:

- b_{op}: index of the block containing operation`op`

- r_{op}: rank of operation `op` i.e. the ordering among the operations of its type according to linearization ordering

Abbreviations:

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

- `blocks[b].sum=blocks[b].sum`$_{\text{enq}}$`+blocks[b].sum`$_{\text{deq}}$ (for b≥0)

- `blocks[b].num`$_x$`=blocks[b].sum`$_x$`-blocks[b-1].sum`$_x$

  (for b>0 and x ∈ {∅, enq, deq, enq-left, enq-right, deq-left, deq-right}, `blocks[0].num`$_x$`=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.sum$_{enq}$= leaf.blocks[leaf.size].sum$_{enq}$+1

205:    newBlock.sum$_{deq}$= leaf.blocks[leaf.size].sum$_{deq}$

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.sum$_{enq}$= leaf.blocks[leaf.size].sum$_{enq}$

212:    newBlock.sum$_{deq}$= leaf.blocks[leaf.size].sum$_{deq}$+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].num$_{enq}$ - 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:        r$_{enq}$= root.blocks[b-1].sum$_{enq}$- root.blocks[b-1].length + i

221:        **return** <root.BSEARCH(sum$_{enq}$, r$_{enq}$, root.FindMostRecentDone(), root.size), r$_{enq}$>

222:    **end if**

223: **end** FINDRESPONSE

Algorithm *Node*

301: *void* PROPAGATE()

firstRefresh 302:   **if not** REFRESH() **then**

secondRefresh 303:   REFRESH()   ▷ Lemma Double Refresh

304:    **end if**

305:    **if this is not** root **then**

306:       parent.PROPAGATE()

307:    **end if**

308: **end** PROPAGATE

309: *boolean* REFRESH()   lastLine

readSize 310:   s= size   prevLine

311:    <new, np_{left}, np_{right}>= CREATEBLOCK(s)   ▷ np_{left}, np_{right} are the
    values read from the children's $num_{propagated}$ field.

312:    **if** new.num==0 **then return** true   ▷ The block contains nothing.

313:    **else if** blocks.tryAppend(new, s) **then**

okcas 314:   **for each** dir **in** {left, right} **do**

315:          CAS(dir.super[np_{dir}], null, h+1)   ▷ Write would work too.

316:          CAS(dir.num$_{propagated}$, np_{dir}, np_{dir}+1)

317:       **end for**

incrementHead3 318:   CAS(size, s, s+1)

319:       **return** true

320:    **else**

321:       CAS(size, s, s+1)   ▷ Even if another
    process wins, help to increase the size. The winner might have fallen sleep

incrementHead2   before increasing size.

322:       **return** false

323:    **end if**

324: **end** REFRESH

⤳ Precondition: blocks[start..end] contains a block with field f $\geq$ i

325: *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 $\geq$ i.

326: **end** BSEARCH

327: <*Block, int, int*> CREATEBLOCK(*int* i)

                              ▷ Creates a block to be inserted into as *i*th 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.

328:    block newBlock= NEW(*block*)

329:    newBlock.group= num$_{propagated}$

330:    newBlock.order= i

331:    **for each** dir **in** {left, right} **do**

332:       index$_{last}$= dir.size

333:       index$_{prev}$= blocks[i-1].end$_{dir}$

334:       newBlock.end$_{dir}$= index$_{last}$

335:       block$_{last}$= dir.blocks[index$_{last}$]

336:       block$_{prev}$= dir.blocks[index$_{prev}$]

337:             ▷ newBlock includes dir.blocks[index$_{prev}$+1..index$_{last}$].

338:       this$_{dir}$= dir.num$_{propagated}$

339:       newBlock.sum$_{enq\text{-}dir}$= blocks[i-1].sum$_{enq\text{-}dir}$ + block$_{last}$.sum$_{enq}$
    - block$_{prev}$.sum$_{enq}$

340:       newBlock.sum$_{deq\text{-}dir}$= blocks[i-1].sum$_{deq\text{-}dir}$ + block$_{last}$.sum$_{deq}$
    - block$_{prev}$.sum$_{deq}$

341:    **end for**

342:    **if this is** root **then**

343:       newBlock.length= max(root.blocks[i-1].length + b.num$_{enq}$ -
    b.num$_{deq}$, 0)

344:    **end if**

345:    **return** <b, np_{left}, np_{right}>

346: **end** CREATEBLOCK

Algorithm Node

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

401: *element* GETENQ(*int* b, *int* i)

402:     **if** this **is** leaf **then**

403:         **return** blocks[b].element

404:     **else if** i $\leq$ blocks[b].num$_{\texttt{enq-left}}$ **then**   ▷ i exists in the left child of this node

405:         subBlock= left.BSEARCH(sum$_{\texttt{enq}}$, i, blocks[b-1].end$_{\texttt{left}}$+1, blocks[b].end$_{\texttt{left}}$)   ▷ Search range of left child's subblocks of blocks[b].

406:         **return** left.GET(i-left.blocks[subBlock-1].sum$_{\texttt{enq}}$, subBlock)

407:     **else**

408:         i= i-blocks[b].num$_{\texttt{enq-left}}$

409:         subBlock= right.BSEARCH(sum$_{\texttt{enq}}$, i, blocks[b-1].end$_{\texttt{right}}$+1, blocks[b].end$_{\texttt{right}}$)   ▷ Search range of right child's subblocks of blocks[b].

410:         **return** right.GET(i-right.blocks[subBlock-1].sum$_{\texttt{enq}}$, subBlock)

411:     **end if**

412: **end** GETENQ

⤳ Precondition: bth block of the node has propagated up to the root and blocks[b].num$_{\texttt{enq}} \geq$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

puteSuper

418:         superBlock= parent.BSEARCH(sum$_{\texttt{deq-dir}}$, i, super[blocks[b].group]-p, super[blocks[b].group]+p)

                                                       ▷ superblock's group has at most p difference with the value stored in super[].

419:         **if** dir **is** right **then**

420:             i+= blocks[superBlock].sum$_{\texttt{deq-left}}$ ▷ 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 suffies.

506: **end** FINDMOSTRECENTDONE

Algorithm Leaf

601: *void* APPEND(*block* blk)   ▷ Append is only called by the owner of the leaf.

appendEnd
602:     size+=1

pendStart
603:     blk.group= size

604:     blocks[size]= blk

605:     parent.PROPAGATE()

606: **end** APPEND

607: *Object* HELPDEQUEUE()

608:     <b_{deq}, r_{deq}>= INDEXDEQ(leaf.size, 1)   ▷ r is the rank among the dequeues of the dequeue of the b_{deq}th block in the root containing.

609:     <b_{enq}, r_{enq}>= FINDRESPONSE(b_{deq}, r_{deq})   ▷ r_{enq} is the rank of the enqueue whose element is the response to the dequeue in the block containing it and

deqRest
     b_{deq} is the index of that block of it in the blocklist. If the response is **null** then r_{deq} is -1.

610:     **if** r_{enq}==-1 **then**

611:         output= null

612:         root.blocks[b_{deq}].num$_{finished}$.inc()   ▷ shared counter

613:         **if** root.blocks[b_{deq}].num$_{finished}$==root.blocks[b_{deq}].num **then**

614:             last$_{done}$= b_{deq}

615:         **end if**

616:     **else**

617:         output= GETENQ(b_{enq}, r_{enq})   ▷ getting the reponse's **element**.

618:         root.blocks[b_{enq}].num$_{finished}$.inc()

619:         root.blocks[b_{enq}].num$_{finished}$.inc()

620:         **if** root.blocks[b_{deq}].num$_{finished}$==root.blocks[b_{deq}].num **then**

621:             last$_{done}$= b_{deq}

622:         **else if** root.blocks[b_{enq}].num$_{finished}$==root.blocks[b_{enq}].num **then**

623:             last$_{done}$= b_{enq}

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

Algorithm BlockList

$\triangleright$: 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.

$\diamondsuit$ `root implementation`

701: *boolean* TRYAPPEND(*block* blk, *int* n) $\triangleright$ adds block b to the root.blocks[n]

702:     **if** `root.size`%p^2`==0` **then** $\triangleright$ Help every often p^2 operations appended to the root.

703:         **for** `leaf l in tree leaves` **do**

704:             `l.Help()`

705:         **end for**

706:     **end if**

707:     `blk.num`$_{finished}$`= 0`

708:     **return** `CAS(blocks[n], null, blk)`

709: **end** TRYAPPEND

$\diamondsuit$ `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) $\triangleright$ *how to use as exception handling? by adding try catch in all the methods reading the root?*

802:     **if** `root.blocks.get(`num_{enq}`)`, i is null **then** $\triangleright$ this enqueue was already finished

803:         **return** `this.leaf.response(block.order)`

804:     **end if**

805: **end** FALLBACK

2 Proof of Linearizability

Definition 1 (Block)**.** A block is an object that stores 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 $ops(b)$, $ops(B)$. We also say b contains op if $op \in ops(b)$.

Definition 2 (Order)**.** If $\texttt{n.blocks[i]==b}$ we call $\texttt{i}$ the *index* of block $\texttt{b}$. Block $\texttt{b}$ is before block $\texttt{b}'$ in node $\texttt{n}$ if and only if $\texttt{b}$'s index is smaller than $\texttt{b}'$'s.

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

For simplicity we say block $\texttt{b}$ is propagated to node $\texttt{n}$ or to a set of blocks $\texttt{S}$ if $\texttt{b}$ is in $\texttt{n.blocks}$ or $\texttt{S}$ or is a subblock of a block in $\texttt{n.blocks}$ or $\texttt{S}$.

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

Lemma 5. *Each block is not suplicated.*

Lemma 6 (headProgress)**.** $\texttt{n.established}$ *is non-decreasing over time.*

Proof. Induced trivially from the pseudocode since $\texttt{n.established}$ is only incremented. $\square$

Lemma 7 (headPosition)**.** *If the value read in Line* 333*(*$\overset{\text{prevLine}}{\texttt{h=n.established}}$*) is* h *then,* $\texttt{n.blocks[i]=null}$ *for* $\texttt{i>h}$ *and* $\texttt{n.blocks[i]}\neq\texttt{null}$ *for* $\texttt{i<h}$.

Proof. At the end of every $\texttt{n.Refresh()}$ with a block with size greater than 0 returned by $\texttt{CreateBlock()}$ $\texttt{n.head}$ is incremented (Lines $\overset{\text{incrementHead2}}{318}$, 321). If a process went to sleep before incrementing the $\texttt{established}$ (line $\overset{\text{incrementHead1}}{318}$), nothing can be appended to $\texttt{n.blocks}$ until another process increments $\texttt{n.establsihed}$ (Line $\overset{\text{incrementHead2}}{321}$). $\square$

Lemma 8 (establishedOrder)**.** *If time* $t < $ *time* t'*, then* $\texttt{ops(EST}_{\texttt{n, t}})\subseteq \texttt{ops(EST}_{\texttt{n, t}'})$.

Proof. Blocks are only appended(not modified) with CAS to $\texttt{n.blocks[n.head]}$ and $\texttt{n.head}$ is non-decreasing, so the set of operations in established blocks of a node grows. $\square$

Lemma 9 (createBlock)**.** *If* $\texttt{n.CreateBlock(i)}$ *is invoked at time* t *and returns block* b*, then* $\texttt{ops(EST}_{\texttt{n.left, t}}) \cup \texttt{ops(EST}_{\texttt{n.right, t}}) - \texttt{ops(EST}_{\texttt{n, t}}) \subseteq \texttt{ops(b)}$.

Proof. WLOG we prove the claim for the left child. Blocks in $\texttt{n.left.blocks[n.blocks[i-1].end}_{\texttt{left}}\texttt{+1..n.blocks[i].end}_{\texttt{left}}\texttt{]}$ are all the established operations at time t by definition of subblock. Since the head is only increasing (Lemma $\overset{\text{lem::sizeProgress}}{6}$) the claim holds after t. See Figure $\overset{\text{fig::createBlock}}{\textbf{??}}$. The right child is the same. $\square$

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* $\mathrm{ops}(\mathrm{EST}_{\mathtt{n.left},\ t_i}) \cup \mathrm{ops}(\mathrm{EST}_{\mathtt{n.right},\ t_i}) \in \mathrm{ops}(\mathrm{EST}_{\mathtt{n},\ t_t})$.

Proof. By Lemma $\overset{\text{lem::createBlock}}{9}$ `new` contains `n`'s childrens' established blocks after Line $\overset{\text{readSize}}{310}$ which is appended to `n.blocks[]` by `TryAppend` in Line 313. □

Lemma 11 (falseRefresh). *If instance* `r` *of* `n.Refresh()` *reads value* `s` *on line* $\overset{\text{readSize}}{310}$ *and then returns* `false`, *then there is another instance* r' *of* `n.Refresh()` *that has performed a successful* `TryAppend(new, s)`. *A* `TryAppend()` *is successful if its* `CAS` *is successful.*

Proof. If there is no other concurrent successful `n.Refresh()` then `n.Refresh()` would succeed in Line 313. So there is another `n.Refresh()`, that has appended its block in `n.blocks[h]` after Line 310 of the first `n.Refresh()`. □

Lemma 12 (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* $\mathrm{ops}(\mathrm{EST}_{\mathtt{n.left},\ t_1}) \cup \mathrm{ops}(\mathrm{EST}_{\mathtt{n.right},\ t_1}) \subseteq \mathrm{ops}(\mathrm{EST}_{\mathtt{n},\ t_2})$.*
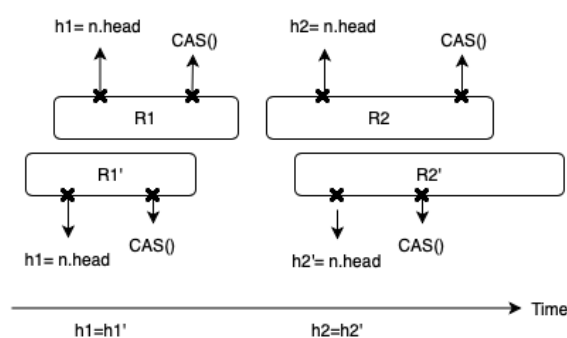

Figure 1: R_2''s `CAS` is executed after `h1=n.head`.

Corollary 13 (Propagate Step). *All operations in* `n`*'s children's established blocks before line* $\overset{\text{firstRefresh}}{302}$ *are guaranteed to be in* `n`*'s established blocks after line* $\overset{\text{secondRefresh}}{303.}$

`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 `Refresh`es 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.

Lemma 14 (No Duplicates). *If* `op` *is appended to* `n.blocks[i]` *then after that there is no* `j>i` *such that* `op`∈`ops(n.blocks[j])`.

Corollary 15. *After* `Append(blk)` *finishes* `ops(blk)`⊆`ops(root.blocks[x])` *for some* `x` *and only one* `x`.

Lemma 16 (Block Size Upper Bound). *Each block contains at most one operation from each processs (*∀ `process p:#operations of p`∈`ops(n.blocks[x]`≤`1)`.

Lemma 17 (Subblocks Upperbound). *Each block has at most p direct subblocks.*

| ordering | **Definition 18** (Ordering of operations inside the nodes). ▶ Note that from Lemma $\overset{\texttt{blockSize}}{16 \text{ we know}}$ 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.*

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

Lemma 20 (Index correctness). `n.Index(b,i)` *returns the rank in* $D(root)$ *of the ith Dequeue in* $D(n, b)$.

| uperBlock | **Lemma 21** (Computing SuperBlock). *After computing line* $\overset{\texttt{computeSuper}}{418 \text{ of}}$ `n.IndexDEQ(b,i)`, `superblock` *contains the ith dequeue in the bth block of the node* `n`.

| mputeHead | **Lemma 22** (Computing Queue's Head). *Let S be the state of an empty queue if the operations in prefix of ith dequeue in $D(root, b)$ are applied on it.* `FindResponse()` *returns the index of the enqueue that is the head in S. If the queue is empty in S it returns* `<-1,-->`.

| head | **Lemma 23** (Validity of `head`). *If block b is written in* `n.blocks[i]` *then* `n.blocks[i]` *is going to remain b (not overwritten in future).*

| erCounter | **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`.

| search | **Lemma 25** (Search Ranges). *Preconditions of all invocation of* `BSearch` *are satisfied.*

| help | **Lemma 26** (help). *After that* `TryAppend()` *who is helping finishes, prefix for the blocks of* `root.blocks[root.FindMostRecentDone` *are done.*

| rootRange | **Lemma 27** (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 28. *The queue implementation is linearizable.*