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* $\texttt{num}_{\texttt{propagated}}$= 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* $\texttt{last}_{\texttt{done}}$

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

► *InternalBlock* `extends` *Block*

- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: index of the last subblock of the block in the left and right child
- *int* $\texttt{sum}_{\texttt{enq-left}}$: # enqueue operations in the prefix for $\texttt{left.blocks[end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-left}}$: # dequeue operations in the prefix for $\texttt{left.blocks[end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: # enqueue operations in the prefix for $\texttt{right.blocks[end}_{\texttt{right}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-right}}$: # dequeue operations in the prefix for $\texttt{right.blocks[end}_{\texttt{right}}\texttt{]}$

► *RootBlock* `extends` *InternalBlock*

- *int* `length` : length of the queue after performing all operations in the prefix for this block
- *counter* $\texttt{num}_{\texttt{finished}}$: number of finished operations in the block

---

Variable naming:

- $\texttt{b}_{\texttt{op}}$: index of the block containing operation`op`
- $\texttt{r}_{\texttt{op}}$: rank of operation `op` i.e. the ordering among the operations of its type according to linearization ordering

Abbreviations:

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

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

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()
302:     if not REFRESH() then
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()
310:     s= size
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
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
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 before
     increasing size.
322:         return false
323:     end if
324: end REFRESH

     ⇝ Precondition: blocks[start..end] contains a block with field f ≥ 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 ≥ i.
326: end BSEARCH

327: <Block, int, int> CREATEBLOCK(int i)
                                                        ▷ Creates a block to be
     inserted into 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.
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-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq
     - block_prev.sum_enq
340:         newBlock.sum_deq-dir= blocks[i-1].sum_deq-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: `blocks[b].num`$_{\texttt{enq}}$≥`i`

```
401: element GetEnq(int b, int i)
402:     if this is leaf then
403:         return blocks[b].element
404:     else if i ≤ blocks[b].num_enq-left then                    ▷ i exists in the left child of this node
405:         subBlock= left.BSearch(sum_enq, i, blocks[b-1].end_left+1, blocks[b].end_left)    ▷ Search range of left child's subblocks of blocks[b].
406:         return left.Get(i-left.blocks[subBlock-1].sum_enq, subBlock)
407:     else
408:         i= i-blocks[b].num_enq-left
409:         subBlock= right.BSearch(sum_enq, i, blocks[b-1].end_right+1, blocks[b].end_right)    ▷ Search range of right child's subblocks of blocks[b].
410:         return right.Get(i-right.blocks[subBlock-1].sum_enq, subBlock)
411:     end if
412: end GetEnq
```

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

```
413: <int, int> IndexDeq(int b, int i)    ▷ Returns the rank of ith dequeue in the bth 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(sum_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_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.
602:     size+=1                                                                                          appendEnd
603:     blk.group= size                                                                                  ppendStart
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
         b_deq is the index of that block of it in the blocklist. If the response is null then r_deq is -1.   deqRest
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.   appendEnd appendStart
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

▷ : 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 $\text{end}_{\texttt{left}}$, $\text{end}_{\texttt{right}}$ 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%p^2==0 then                           ▷ 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
```

◇ *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(num_enq), 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.established 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 refrences).

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 $\texttt{ops(EST}_{\texttt{n, t}}\texttt{)}$ with $\texttt{EST}_{\texttt{n, t}}$?

TEST How to merge notions of blocks and operations? block b ∈ block c means b is subblock of c. block b ∈ 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 $ops(b)$, $ops(B)$. We also say b contains op if $op \in ops(b)$.

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

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

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

Lemma 6. *Each block is not duplicated.*

Proof. `n.CreateBlock(i)` is successful for `i=1,2,3,...` exactly one time and by definition of `CreateBlock` the range of their blocks are disjoint so in propagation procedure on block can not be propagated twice (reference to propagation). □

eProgress **Lemma 7** (headProgress)**.** `n.established` *is non-decreasing over time.*

Proof. Induced trivially from the pseudocode since `n.established` is only incremented. □

dPosition **Lemma 8** (headPosition)**.** *If the value read in Line* prevLine *333(* `h=n.established`*) is h then,* `n.blocks[i]=null` *for* `i>h` *and* `n.blocks[i]`$\neq$`null` *for* `i<h`*.*

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

shedOrder **Lemma 9** (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 grows. □

eateBlock **Lemma 10** (createBlock)**.** *If* `n.CreateBlock(i)` *is invoked at time t and returns block new, then* $\texttt{ops(EST}_{\texttt{n.left, t}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}}\texttt{)}$ - $\texttt{ops(EST}_{\texttt{n, t}}\texttt{)} \subseteq \texttt{ops(new)}$.

Proof. From lemmas 7,8 we know that `n.blocks[i-1]` is full and assume it is invoked at t'. We want to prove every block b that is $\in \texttt{EST}_{\texttt{n.left,t}} \cup \texttt{EST}_{\texttt{n.right,t}}$ is also $\in \texttt{EST}_{\texttt{n,t}}' \cup \texttt{new}$. It is easy to see $\texttt{EST}_{\texttt{n.left,t}} \cap \texttt{EST}_{\texttt{n.right,t}}$ is $\emptyset$. If b is $\in \texttt{EST}_{\texttt{n.left,t}'}$ it is simple to see. Else we can see from the code of the `CreateBlock` that `new` contains it. Right child is the same. □

ueRefresh **Lemma 11** (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* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \in \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$.

Proof. Suppose the previous true returning `n.Refresh()` took place in time $(t_i' - t_t')$. By induction on the number of succsuful `Refresh()`'s `n` we know that $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}'}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}'}\texttt{)} \in \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}'}\texttt{)}$. Operations that are in $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \in \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$ but not in $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}'}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}'}\texttt{)} \in \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}'}\texttt{)}$ are in the block returned by `n.CreateBlock()` by lemma 10. By Lemma 10 `new` contains `n`'s childrens' established blocks after Line 310 which is appended to `n.blocks[]` by `TryAppend` in Line 313. □

seRefresh **Lemma 12** (falseRefresh)**.** *If instance* `r` *of* `n.Refresh()` *reads value* `s` *on line 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()`. □

bleRefresh **Lemma 13** (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* $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_2}\texttt{)}$.

Proof.

If Line 313 of r_1 or r_2 returns `true`, then the claim is held by Lemma 11. If not, then there is another successful instance of `n.Refresh()` r_1' by Lemma 12 for r_2. From $t_{i'_{r_2}} > t_1$ and Lemma 11, 9 we know that $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_2}\texttt{)}$. □

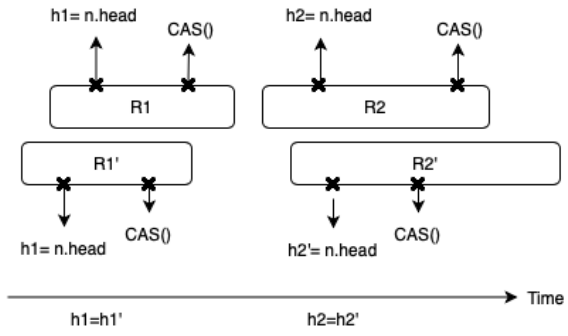

Figure 1: R_2''s `CAS` is executed after `h1=n.head`.

lyRefresh **Corollary 14** (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 13. □

`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.

append **Lemma 15** (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 16. *After* `Append(blk)` *finishes* `ops(blk)⊆ops(root.blocks[x])` *for some* `x` *and only one* `x`.

blockSize **Lemma 17** (Block Size Upper Bound). *Each block contains at most one operation from each processs* (∀ `process p:#operations of p∈ops(n.blocks[x]≤1)`.

ocksBound **Lemma 18** (Subblocks Upperbound). *Each block has at most p direct subblocks.*

ordering **Definition 19** (Ordering of operations inside the nodes). ▶ Note that from Lemma blockSize 17 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 ∈ `ops(n.blocks[i])` ordered by their process id.
- $D(n,i)$ is the sequence of dequeue operations ∈ `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 20** (Search Ranges). *Preconditions of all invocation of* `BSearch` *are satisfied.*

get **Lemma 21** (Get correctness). `n.GetENQ(b,i)` *returns the ith Enqueue in* $E(n,b)$.

help **Lemma 22** (help). *After that* `TryAppend()` *who is helping finishes, prefix for the blocks of* `root.blocks[root.FindMostRecentDone]` *are done.*

Lemma 23 (Index correctness). `n.Index(b,i)` *returns the rank in* $D(root)$ *of the ith Dequeue in* $D(n,b)$.

uperBlock **Lemma 24** (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`.

mputeHead **Lemma 25** (Computing Queue's Head). *Let S be the state of an empty queue if the operations in prefix in L of ith 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,-->`.

erCounter **Lemma 26** (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±p`.

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

Lemma 29 (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* $O(\log^2 p)$.