

1 Pseudocode

Algorithm Tree Fields Description

◇ Shared

- *Tree* *tree* : A binary tree of Nodes. *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* *blocks* implemented with an array.
- *int* *head*= 1: #blocks in *blocks*(-1). *blocks*[0] is a block with all fields equal to zero.
- *int* *num_{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*.

► *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 *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}* : index of the last subblock of the block in the left and right child
- *int* *sum_{enq-left}* : # enqueue operations in the prefix for *left.blocks[end_{left}]*
- *int* *sum_{deq-left}* : # dequeue operations in the prefix for *left.blocks[end_{left}]*
- *int* *sum_{enq-right}* : # enqueue operations in the prefix for *right.blocks[end_{right}]*
- *int* *sum_{deq-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-left}*+*blocks*[*b*].*sum_{x-right}* (for *b*≥0 and *x* ∈ {*enq*, *deq*})
- *blocks*[*b*].*sum*=*blocks*[*b*].*sum_{enq}*+*blocks*[*b*].*sum_{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.sumenq= leaf.blocks[leaf.head].sumenq+1
205:   newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq
206:   leaf.head+=1
207:   leaf.APPEND(newBlock)
208: end ENQUEUE

209: <int, int> FINDRESPONSE(int bd, int id) ▷ If  $E(root, b_e, i_e)$  is the response to the  $D(root, b_d, i_d)$  returns <be, ie>. Returns <-1, --> if the queue is empty.
210:   if root.blocks[bd-1].size + root.blocks[bd].numenq - (i + root.blocks[bd-1].sumdeq) < 0 then
211:     return <-1, -->
212:   else
213:     renq= i + root.blocks[bd-1].sumdeq - (root.blocks[bd-1].size - root.blocks[bd-1].sumenq + root.blocks[bd-1].sumdeq)
214:     ▷ size-enqs+deqs=null deqs
215:     return root.DSEARCH(renq, bd)
216:   end if
217: end FINDRESPONSE

218: Object DEQUEUE()
219:   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.
220:   newBlock.element= null
221:   newBlock.sumenq= leaf.blocks[leaf.head].sumenq
222:   newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq+1
223:   leaf.APPEND(newBlock)
224:   <bdeq, rdeq>= INDEXDEQ(leaf.head, 1) ▷ rdeq is the rank among the dequeues of the dequeue of the bdeqth block in the root containing.
225:   <benq, renq>= FINDRESPONSE(bdeq, rdeq) ▷  $E(root, b_{enq}, i_{enq})$  is response to the  $D(root, b_{deq}, i_{deq})$ . If the response is null then
226:     renq is -1.
227:   if renq== -1 then
228:     output= null
229:   else
230:     output= GETENQ(benq, renq) ▷ getting the reponse's element.
231:   end if
232:   return output
233: end DEQUEUE
```

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:   h = head
311:   <new, npleft, npright> = CREATEBLOCK(s)
312:   if new.num==0 then return true
313:   else if blocks.tryAppend(new, h) 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(head, h, h+1)
319:     return true
320:   else
321:     CAS(head, h, h+1)
322:   end if
323: end REFRESH

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:   for each dir in {left, right} do
331:     indexlast = dir.head
332:     indexprev = blocks[i-1].enddir
333:     newBlock.enddir = indexlast
334:     blocklast = dir.blocks[indexlast]
335:     blockprev = dir.blocks[indexprev]
336:     npdir = dir.numpropagated
337:     newBlock.sumenq-dir = blocks[i-1].sumenq-dir + blocklast.sumenq
338:     newBlock.sumdeq-dir = blocks[i-1].sumdeq-dir + blocklast.sumdeq
339:     newBlock.size = max(root.blocks[i-1].size + newBlock.numenq - newBlock.numdeq, 0)
340:   end for
341:   if this is root then
342:     newBlock.size = max(root.blocks[i-1].size + newBlock.numenq - newBlock.numdeq, 0)
343:   end if
344:   return <b, npleft, npright>
345: end CREATEBLOCK

```

firstRefresh
 secondRefresh
 readHead
 cas
 okcas
 incrementHead1
 incrementHead2
 lastLine
 prevLine
 computeLength

▷ 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.

▷ Lemma Double Refresh

▷ np_{left}, np_{right} are the values read from the children's num_{propagated} field.

▷ The block contains nothing.

▷ Write would work too.

▷ Even if another process wins the race to increase the head. The winner might have fallen sleep before increasing head.

▷ 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 $\geq i$.

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

Algorithm Root

\leadsto Precondition: $\text{root.blocks}[\text{end}].\text{sum}_{\text{enq}} \geq r_{\text{enq}}$

```
801: <int, int> DSEARCH(int i, int end)
    ▷ Searches for the  $i$ th enqueue of the given prefix sum of  $b$ th block in the  $\text{root}$ . Returns the index of the leftmost block in  $\text{root.blocks}$  whose  $\text{sum}_{\text{enq}}$  is  $\geq i$ .
802:     start= end-1
803:     while root.blocks[start].sumenq ≥ renq do
804:         start= start-(end-start)
805:     end while
806:     return root.BSearch(sumenq, renq, start, end)
807: end DSEARCH
```

Algorithm Node

\leadsto 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 ≤ blocks[b].numenq-left then                                     ▷ i exists in the left child of this node
405:         subBlock= left.BSEARCH(sumenq, i, blocks[b-1].endleft+1, blocks[b].endleft)           ▷ Search range of left child's subblocks of blocks[b].
406:         return left.GET(i-left.blocks[subBlock-1].sumenq, subBlock)
407:     else
408:         i= i-blocks[b].numenq-left
409:         subBlock= right.BSEARCH(sumenq, i, blocks[b-1].endright+1, blocks[b].endright)           ▷ Search range of right child's subblocks of blocks[b].
410:         return right.GET(i-right.blocks[subBlock-1].sumenq, subBlock)
411:     end if
412: end GETENQ
```

\leadsto 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(sumdeq-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].sumdeq-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 Leaf

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

```
602:     head+=1
603:     blk.group= head
604:     blocks[head]= blk
605:     parent.PROPAGATE()
606: 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 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.

`blocks[]`: array of blocks

701: *boolean* TRYAPPEND(*block* blk, *int* n)

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

703: **end** TRYAPPEND

2 Proof of Linearizability

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

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

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

Definition 1 (Block). A block is an object storing some statistics, as described in Algorithm Queue. It implicitly represents a set of operations. If $n.\text{blocks}[i] == b$ 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.

Definition 2 (Subblock). Block b is a *direct subblock* of $n.\text{blocks}[i]$ if it is $\in n.\text{left}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1..n.\text{blocks}[i].\text{end}_{\text{left}}] \cup n.\text{right}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}+1..n.\text{blocks}[i].\text{end}_{\text{right}}]$. Block b is a subblock of a $n.\text{blocks}[i]$ if it is a direct subblock of it or subblock of a direct subblock of it.

Definition 3 (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 4 (Operations of a block). A block lb in a leaf represents one operation which if it is $\text{enqueue}(x)$ then $lb.\text{element} = x$, otherwise $\text{element} = \text{null}$. The set of operations of block b are the operations in the subblocks of b . We show the set of operations of block b by $\text{ops}(b)$.

For simplicity we say block b is propagated to node n or to a set of blocks S if b is in $n.\text{blocks}$ or S or is a subblock of a block in $n.\text{blocks}$ or S . We also say b contains op if $op \in \text{ops}(b)$.

Definition 5. A block b in $n.\text{blocks}$ is *established* at time t if the last value written into $n.\text{head}$ before t is greater than the index of b in $n.\text{blocks}$ at time t . $\text{EST}_{n, t}$ is the set of established blocks at time t of node n .

head **Observation 6.** Once a block b is written in $n.\text{blocks}[i]$ then $n.\text{blocks}[i]$ never changes.

dProgress **Lemma 7** (headProgress). $n.\text{head}$ is non-decreasing over time and $n.\text{blocks}[i].\text{end}_{\text{left}} \geq n.\text{blocks}[i-1].\text{end}_{\text{left}}, n.\text{blocks}[i].\text{end}_{\text{right}} \geq n.\text{blocks}[i-1].\text{end}_{\text{right}}$.

Proof. The first claim follows trivially from the pseudocode since $n.\text{head}$ is only incremented. Also when $n.\text{blocks}[i]$ is created its end_{left} , $\text{end}_{\text{right}}$ are greater than or equal to the values in $n.\text{blocks}[i-1]$. Since $\text{blocks}[i-1].\text{end}_{\text{dir}} < \text{dir}.\text{head} = \text{blocks}[i].\text{end}_{\text{dir}}$ (Lines lastLine, prevLine ~~177~~). □

Lemma 8. Every block has most one direct superblock.

Proof. To show this we are going to refer to the way $n.\text{blocks}[]$ is partitioned while propagating blocks up to $n.\text{parent}$. $n.\text{CreateBlock}(i)$ merges the blocks in $n.\text{left}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}..n.\text{blocks}[i].\text{end}_{\text{left}}]$ and $n.\text{right}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}..n.\text{blocks}[i].\text{end}_{\text{right}}]$ (Lines lastLine, pr ~~177~~). Since $\text{end}_{\text{left}}, \text{end}_{\text{right}}$ are non-decreasing, so the range of the subblocks of $n.\text{blocks}[i]$ which is $(n.\text{blocks}[i-1].\text{end}_{\text{dir}}+1..n.\text{blocks}[i].\text{end}_{\text{dir}}]$ does not overlap with the range of the subblocks of $n.\text{blocks}[i-1]$. □

append **Corollary 9** (No Duplicates). If op is in $n.\text{blocks}[i]$ then there is no $j \neq i$ such that $op \in \text{ops}(n.\text{blocks}[j])$.

dPosition **Invariant 10** (headPosition). If the value of $n.\text{head}$ is h then, $n.\text{blocks}[i] = \text{null}$ for $i > h$ and $n.\text{blocks}[i] \neq \text{null}$ for $i < h$.

Proof. The invariant is true initially since 1 is assigned to $n.\text{head}$ and $n.\text{blocks}[x]$ is null for every x . The truth of the invariant may be affected by writing into $n.\text{blocks}$ or incrementing $n.\text{head}$.

Some value is written into $n.\text{blocks}[\text{head}]$ only in Line 313. It is obvious that writing into $n.\text{blocks}[\text{head}]$ preserves the invariant. The value of $n.\text{head}$ is modified only in lines incrementHead ~~318~~, ~~321~~. Depending on whether the $\text{TryAppend}()$ in Line cas ~~313~~ succeeded or not we show that

the claim holds after the increment lines of `n.head` in either case. If `head` is incremented to h it is sufficient to show `n.blocks[h] ≠ null` to prove the invariant still holds. In the first case the process applied a successful `TryAppend(new, h)` in line ^{lokcas}314, which means `n.blocks[h]` is not null anymore. Note that whether ^{incrementHead1}318 returns true or false after Line `n.head` we know has been incremented from Line ^{readHead}310. The failure case is also the same since it means some value is written into `n.blocks[head]` by some process. \square

Explain More

Lemma 11 (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 can only grows. \square

`CreateBlock()` aggregates the blocks in the children that are not already established in the parent 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 another 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 12 (trueRefresh). *Let t_i be the time a `n.Refresh()` operation is invoked and t_t be the time it terminates. 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 ^{lem::establishedOrder}11 we know that $\text{ops}(\text{EST}_n, t_i) \subseteq \text{ops}(\text{EST}_n, t_t)$. So it remains to show the operations of $\text{ops}(\text{EST}_{n.\text{left}}, t_i) \cup \text{ops}(\text{EST}_{n.\text{right}}, t_i) - \text{ops}(\text{EST}_n, t_i)$, which we call new operations, are all in $\text{ops}(\text{EST}_n, t_t)$. 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

Mention this new block is establish in t_t . last sentece is not complete

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. 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 ^{cas}313 of R_1 or R_2 returns true, then the claim is held by Lemma ^{lem::trueRefresh}12. If not, then there is another successful instance R'_2 of `n.Refresh()` that has done `TryAppend()` successfully into `n.blocks[i+1]`. If R_2 reads some value greater than $i+1$ in Line ^{readHead}310 it means a successful instance of `Refresh()` started after Line ^{readHead}310 of R_1 and finished its Line ^{incrementHead2}318 or ^{incrementHead2}321 before ^{readHead}310 of R_2 , from Lemma ^{lem::trueRefresh}12 by the end of this instance $\text{ops}(\text{EST}_{n.\text{left}}, t_1) \cup \text{ops}(\text{EST}_{n.\text{right}}, t_1)$ has been propagated. In Figure 1 we see why the block R'_2 is appending contains established block in the `n`'s children at t_1 , since it create a block reading the head after t_1 . \square

last sentence need more detail and should be earlier. define i and tell why R_2 prime exists

this chain with more depth should be in the proof

Corollary 14 (Propagate Step). *All operations in `n`'s children's established blocks before line ^{firstRefresh}302 are guaranteed to be in `n`'s established blocks after line ^{secondRefresh}303.*

Proof. Lines ^{firstRefresh}302 and ^{secondRefresh}303 satisfy the preconditions of Lemma ^{doubleRefresh}13. \square

Corollary 15. *After `Append(blk)` finishes $\text{ops}(\text{blk}) \subseteq \text{ops}(\text{root.blocks}[x])$ for some x and only one x .*

Proof. Follows from Lemma ^{doubleRefresh}13, 14. \square

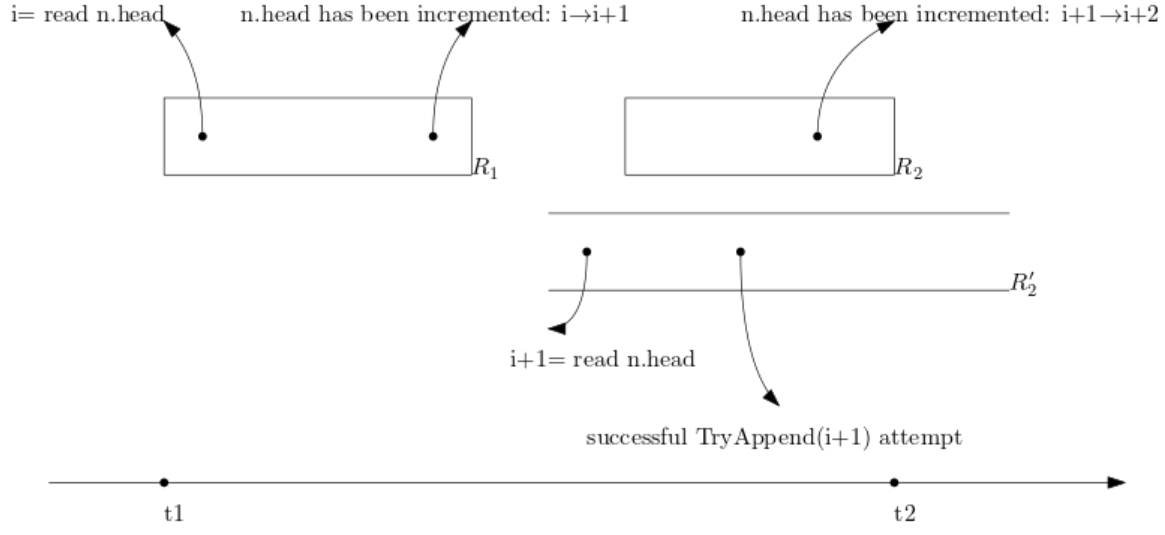


Figure 1: $t1 < r_1$ reading head < incrementing `n.head` from i to $i + 1 < R'_2$ reading head < `TryAppend(i+1)` < incrementing `n.head` from $i + 1$ to $i + 2 < t2$

blockSize **Lemma 16** (Block Size Upper Bound). *Each block contains at most one operation from each process.*

Proof. By proof of contradiction, assume there are more than one operation from process p in block b in node n . A process cannot invoke more than one operations concurrently. From p 's operations in b , let op_1 be the first operation invoked and op_2 be the second one. Note that it is terminated before op_2 started. So before appending op_2 to the tree op_1 exists in every node from the path of p 's leaf to the root. So there is some block b' before b in n containing op_1 . op_1 existing in b and b' contradicts with append. \square

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

Proof. It follows directly from Lemma blockSize and the observation that each block contains at least one operation, induced from Line addOP. \square

ordering **Definition 18** (Ordering of operations inside the nodes). \blacktriangleright Note that processes are numbered from 1 to p , left to right in the leaves of the tree and from Lemma blockSize 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 .
- $E(n, b)$ is the sequence of enqueue operations $\in \text{ops}(\mathbf{n.blocks}[b])$ ordered by their process id.
- $E(n, b, i)$ is the i th enqueue in $E(n, b)$.
- $D(n, b)$ is the sequence of dequeue operations $\in \text{ops}(\mathbf{n.blocks}[b])$ ordered by their process id.
- $D(n, b, i)$ is the i th dequeue in $D(n, b)$.
- Order of the enqueue operations in n : $E(n) = E(n, 1).E(n, 2).E(n, 3)...$
- $E(n, i)$ is the i th enqueue in $E(n)$.
- Order of the dequeue operations in n : $D(n) = D(n, 1).D(n, 2).D(n, 3)...$
- $D(n, i)$ is the i th dequeue in $D(n)$.
- Linearization: $L = E(\text{root}, 1).D(\text{root}, 1).E(\text{root}, 2).D(\text{root}, 2).E(\text{root}, 3).D(\text{root}, 3)...$

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

get **Lemma 19** (Get correctness). *If $\mathbf{n.blocks}[b].\text{num}_{\text{enq}} \geq i$ then $\mathbf{n.GetENQ}(b, i)$ returns $E(n, b, i)$.*

Proof. We are going to prove this lemma by induction on the height of the tree. The base case for the leaves of the tree is pretty straight forward. Since leaf blocks contain exactly one operation then only `GetENQ(b,1)` can be called on leaves. `leaf.GetENQ(b,1)` returns the operation stored in the b th block of leaf l . For non leaf nodes in Line 404 it is decided that the i th enqueue in block b of internal node n resides in the left child or the right child of n . By definition of $E(n,b)$ operations from the left child come before the operations of the right child. Having `sumenq`, the prefix sum of the number of enqueues we can compute the direct subblock containing the enqueue we are finding for with binary search. Then `n.child.GetENQ(block containing, order in the block)` is invoked which returns the correct operation by the hypothesis of the induction. \square

I'm not sure it is going to be long and boring to talk about the parameters, since the reader can find out them.

dsearch

Lemma 20 (DSearch correctness). *If `root.blocks[end].numenq ≥ i` and $E(\text{root}, i)$ is the response to some `Dequeue()` in `root.blocks[end]` then `DSearch(i, end)` returns b such that `root.blocks[b]` contains $E(\text{root}, b, i)$ in $\Theta(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$ steps.*

Proof. First we show `end - b ≤ root.blocks[b].size + root.blocks[end].size`. We know each block size is greater than 0. So every block in `root.blocks[b..end]` contains at least one `Enqueue()` or one `Dequeue()`. There cannot be more than `root.blocks[b].size` `Dequeue()`s in `root.blocks[b+1..end-1]`, since the queue would become empty after b th block end before `end` and $E(n, i)$ could not be the response to some `DEQ` in `end`. And since the length of the queue would become `root.blocks[end].size` in the end so there cannot be more than `root.blocks[end].size` `Dequeus` in `root.blocks[b..end]`. Cause if there was more then the end's length would become more than `root.blocks[end].size`.

Now that we know there are at most `root.blocks[b].size + root.blocks[end].size` distance between `end` and b then with doubling search in $\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size})$ steps we reach a block c that the `c.sumenq` is less than i and the distance between c and `end` is not more than $2 \times \text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}$. So the binary search takes $\Theta(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$ steps. \square

uperBlock

Lemma 21 (Computing SuperBlock). *After computing line 418 of `n.computeSuperIndexDEQ(b,i)`, `n.parent.blocks[superblock]` contains $D(n, b, i)$.*

Proof. 1. Value read for `super[b.group]` in line 418 is not null.

► Values `cdir` read in lines 23, `super` are set before incrementing in lines 26,27.

2. `super[]` preserves order from child to parent; if in a child block b is before c then `b.group ≤ c.group` and `super[b.group] ≤ super[c.group]`

► Follows from the order of lines 60, 48, 49.

3. `super[i+1] - super[i] ≤ p`

► 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[super[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 .

4. Superblock of b is within range $\pm 2p$ of the `super[b.time]`.

► `super[i]` is the index of the superblock of a block containing block b , followed by Lemma 25. ^{superCounter} It is trivial to see that `n.super` and `n.b.counter` are increasing. `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:

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

\square

We call the dequeues that return some value *non-null dequeues*. r th non-null dequeue returns the element of the r th enqueue. We can compute # non-null dequeues in the prefix for a block this way: #non-null dequeues = size - #enqueues. Note that the i th dequeue in the given block is not a non-null dequeue.

Lemma 22 (Index correctness). $n.\text{IndexDEQ}(b, i)$ returns the rank in $D(\text{root})$ of $D(n, b, i)$.

Proof. We can see the base case $\text{root}.\text{IndexDEQ}(b, i)$ is trivial. $n.\text{IndexDEQ}(b, i)$ computes the superblock of the i th Dequeue in the b th block of n in $n.\text{parent}(\text{Lemma } \frac{\text{superBlock}}{21})$. Then the order in $D(n.\text{parent}, \text{superblock})$ is computed and $\text{index}()$ is called on $n.\text{parent}$ recursively. It is easy to see why the second is correct. Correctness of computing superblock comes from Lemma $\frac{\text{superBlock}}{21.\text{size}}$ \square

Do I need to talk about the computation of the order in the parent which is based on the definition of ordering of dequeues in a block?

search **Lemma 23** (Search Ranges). *Preconditions of all invocation of BSearch are satisfied.*

Proof. Line 83: $\text{Get}(i)$ is called if the result of a dequeue is not null. The search is among all blocks in the root.

Line 88: This search tries to find the i th enqueue, knowing that it is in the left child. Search is done over the left subblocks. The start and end of the range are followed by definition. Line 92 is the same.

Line 101: Here, the goal is to find the superblock. We know the distance between answer and the $\text{super}[i]$ is at most p , since at most p processes could die. \square

computeHead **Lemma 24** (Computing Queue's Head block). *Let S be the state of an empty queue if the operations in prefix in L of i th dequeue in $D(\text{root}, b)$ are applied on it. $\text{FindResponse}()$ returns (b, i) which $E(\text{root}, b, i)$ is the head of the queue in S . If the queue is empty in S then it returns $\langle -1, - \rangle$.*

Proof. The size of the queue if the operations in the prefix for the b th block in the root are applied with the order of L is stored in the $\text{root}.\text{blocks}[b].\text{size}$. It is computed while creating the block in Line $\frac{\text{computeLength}}{342}$. If the size of a queue is greater than 0 then a $\text{Dequeue}()$ would decrease the size of the queue, otherwise the size of the queue remains 0. Having size of the queue after the previous block and number of enqueues and dequeues in the block, Line $\frac{\text{computeLength}}{342}$ computes whether the queue becomes empty or the size of it. \square

HOW? How to prove mathematically that $\text{ax}(\text{root}.\text{blocks}[i-1].\text{size} + b.\text{num}_{\text{enq}} - b.\text{num}_{\text{deq}}, 0)$ is the size of the queue after the block. I can only explain it here.

erCounter **Lemma 25** (Validity of super and counter). *If $\text{super}[i] \neq \text{null}$, then $\text{super}[i]$ in node n is the index of the superblock of a block with $\text{time} = i \pm p$.*

rootRange **Lemma 26** (Root search range). $\text{root}.\text{size} - \text{root}.\text{FindMostRecentDone}()$ is $O(p^2 + q)$, which p is # processes and q is the size of the queue.

Theorem 27 (Main). *The queue implementation is linearizable.*

Lemma 28 (Time analysis). $n.\text{GetEnq}(b, i)$, $n.\text{Index}(b, i)$ 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 $O(\log^2 p)$.