

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

<pre> 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, np_{left}, np_{right}> = CREATEBLOCK(h) 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[np_{dir}], null, h+1) 316: CAS(dir.num_{propagated}, np_{dir}, np_{dir}+1) 317: end for 318: CAS(size, h, h+1) 319: return true 320: else 321: CAS(size, h, h+1) 322: return false 323: end if 324: end REFRESH </pre>	<pre> 327: <Block, int, int> CREATEBLOCK(int i) 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: this_{dir} = dir.num_{propagated} 338: newBlock.sum_{enq-dir} = blocks[i-1].sum_{enq-dir} + block_{last}.sum_{enq} 339: - block_{prev}.sum_{enq} 340: newBlock.sum_{deq-dir} = blocks[i-1].sum_{deq-dir} + block_{last}.sum_{deq} 341: - block_{prev}.sum_{deq} 342: end for 343: if this is root then 344: newBlock.length = max(root.blocks[i-1].length + b.num_{enq} - 345: b.num_{deq}, 0) 346: end if 347: return <b, np_{left}, np_{right}> 348: end CREATEBLOCK </pre>
--	--

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

```

325: int BSEARCH(field f, int i, int start, int end)
326:   i of the given prefix sum field. Returns the index of the leftmost block in
327:   blocks[start..end] whose field f is  $\geq i$ .
328: end BSEARCH

```

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.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 references).

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 `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 subblock of `n.blocks[i]` if it is $\in n.left.blocks[n.blocks[i-1].end_{left}+1..n.blocks[i].end_{left}] \cup n.right.blocks[n.blocks[i-1].end_{right}+1..n.blocks[i].end_{right}]$ or is a subblock of a block in it.

For simplicity we say block b is propagated to node n or set 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 in `n.blocks` is in $EST_{n, t}$ if b is a subblock of b' in `n.blocks` such that b' is established at time t .

Lemma 5 (headProgress). `n.established` is non-decreasing over time.

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

Lemma 6 (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]≠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 incrementHead1 ~~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.established` (Line incrementHead2 ~~321~~). \square

Lemma 7 (establishedOrder). If time $t < t'$, then $ops(EST_{n, t}) \subseteq ops(EST_{n, t'})$.

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 8 (createBlock). If b is the block returned by `n.CreateBlock(h, x)` invoked at time t , then $ops(EST_{n.left, t}) \cup ops(EST_{n.right, t}) - ops(EST_{n, t}) \subseteq ops(b)$.

Proof. WLOG we prove the claim for the left child. Blocks in `n.left.blocks[n.blocks[i-1].end_{left}+1..n.blocks[i].end_{left}]` are all the new established operations at time t by definition of Subblock. Since the head is only increasing (Lemma lem::sizeProgress ~~5~~) the claim holds after t . See Figure fig::createBlock ~~??~~. The right child is the same. \square

Lemma 9 (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 $ops(EST_{n.left, t_i}) \cup ops(EST_{n.right, t_i}) \in ops(EST_{n, t_t})$.

Proof. By Lemma lem::createBlock ~~8~~ `new` contains `n`'s childrens' established blocks after Line readSize ~~310~~ which is appended to `n.blocks[]` by `TryAppend` in Line 313. \square

Lemma 10 (falseRefresh). If instance r of `n.Refresh()` reads value s on line 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()`. \square

leRefresh

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 $\text{ops}(\text{EST}_{n.\text{left}}, t_1) \cup \text{ops}(\text{EST}_{n.\text{right}}, t_1) \subseteq \text{ops}(\text{EST}_n, t_2)$.*

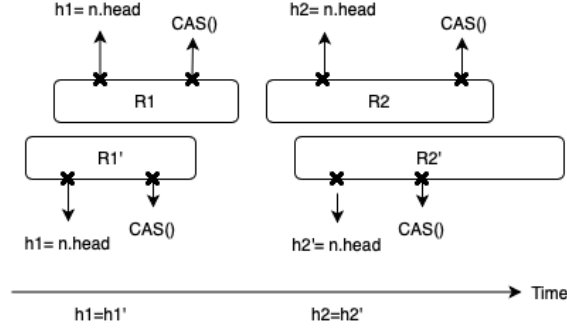


Figure 1: R_2' 's CAS is executed after $h1=n.head$.

lyRefresh

Corollary 12 (Propagate Step). *All operations in n 's children's established blocks before line ^{firstRefresh}302 are guaranteed to be in n 's blocks after line ^{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 `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 $op \in \text{ops}(n.blocks[j])$.*

Corollary 14. *After `Append(blk)` finishes $\text{ops}(blk) \subseteq \text{ops}(\text{root.blocks}[x])$ 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 $p: \# \text{operations of } p \in \text{ops}(n.blocks[x]) \leq 1$).*

ocksBound

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

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

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

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

Theorem 18. *The queue implementation is linearizable.*

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

Lemma 20 (Index correctness). `n.Index(b, i)` returns the rank in the $D(\text{root})$ of i th Dequeue in $D(n, b)$.

superBlock **Lemma 21** (Computing SuperBlock). After computing line 418 of `computeSuper` of `n.IndexDEQ(b, i)`, `superblock` contains the i th dequeue in the b th block of the node \mathbf{n} .

computeHead **Lemma 22** (Computing Queue's Head). Let S be the state of an empty queue if the operations in prefix of i th dequeue in $D(\text{root}, b)$ 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] ≠ null`, then `super[i]` in node \mathbf{n} is the index of the superblock of a block with `time=i±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.

TODO Fallback safety lemmas.