1 Pseudocode

Algorithm Tree Fields Description

◇ *Shared*

- A binary tree of Nodes with one `leaf` for each process. `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*
- *int* `head= 1`: #blocks in `blocks`. `blocks[0]` is a block with all integer fields equal to zero.
- *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.

▶ *Block*

- *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. If the operation is `enqueue(x)` then `element=x`, otherwise `element=null`.
- *int* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: # enqueue, dequeue operations in the prefix for the block

▶ *InternalBlock* `extends` *Block*

- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: indices 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* `size` : size of the queue after performing all operations in the prefix for this block

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 $\texttt{x} \in \{\emptyset, \texttt{enq, deq, enq-left, enq-right, deq-left, deq-right}\}$)

Algorithm *Queue*

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

208: Object DEQUEUE()  ▷ Creates a block with null value element, appends it
     to the tree, computes its order among operations, and returns its response.
209:     block newBlock= NEW(LeafBlock)
210:     newBlock.element= null
211:     newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq
212:     newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq+1
213:     leaf.APPEND(newBlock)
214:     <b, i>= INDEXDEQ(leaf.head, 1)
215:     output= FINDRESPONSE(b, i)
216:     return output
217: end DEQUEUE

218: <int, int> FINDRESPONSE(int b, int i)
                                  ▷ Returns the the response to the D_{root,b,i}.
219:     if root.blocks[b-1].size  + root.blocks[b].num_enq - i < 0 then
220:         return null                          ▷ Check if the queue is empty.
221:     else
222:         e= i - root.blocks[b-1].size + root.blocks[b-1].sum_enq
                                                  ▷ E_{root,e} is the response.
223:         return root.GetENQ(root.DSEARCH(e, b))
224:     end if
225: end FINDRESPONSE
```

checkEmpty

computeE

findAnswer

deqRest

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, np_left, np_right>= CREATEBLOCK(h)     ▷ 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, h) then
314:         for each dir in {left, right} do
315:             CAS(dir.super[np_dir], null, h)      ▷ Write would work too.
316:             CAS(dir.num_propagated, np_dir, np_dir+1)
317:         end for
318:         CAS(head, h, h+1)
319:         return true
320:     else
321:         CAS(head, h, h+1)            ▷ Even if another process wins, help
     to increase the head. The winner might have fallen sleep before increasing
     head.
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 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:     for each dir in {left, right} do
331:         index_last= dir.head-1
332:         index_prev= blocks[i-1].end_dir
333:         newBlock.end_dir= index_last
334:         block_last= dir.blocks[index_last]
335:         block_prev= dir.blocks[index_prev]
336:                 ▷ newBlock includes dir.blocks[index_prev+1..index_last].
337:         np_dir= dir.num_propagated
338:         newBlock.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq
     - block_prev.sum_enq
339:         newBlock.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq
     - block_prev.sum_deq
340:     end for
341:     if this is root then
342:         newBlock.size = max(root.blocks[i-1].size  + newBlock.num_enq
     - newBlock.num_deq, 0)
343:     end if
344:     return <b, np_left, np_right>
345: end CREATEBLOCK
```

Algorithm Root

```
     ⇝ Precondition: root.blocks[end].sum_enq ≥ e
801: <int, int> DSEARCH(int e, int end)          ▷ Returns <b,i> if E_root,e = E_root,b,i.
802:     start= end-1
803:     while root.blocks[start].sum_enq≥e do
804:         start= max(start-(end-start), 0)
805:     end while
806:     b= root.BSearch(sum_enq, e, start, end)
807:     i= e- root.blocks[b-1].sum_enq
808:     return <b,i>
809: end DSEARCH
```

Algorithm Node

⇝ Precondition: `blocks[b].num`$_{\texttt{enq}}$`≥i≥ 1`

401: *element* GETENQ(*int* b, *int* i) ▷ Returns the element of $E_{this,b,i}$.

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

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

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

405: subBlock= left.BSEARCH(sum$_{\texttt{enq}}$, i+blocks[b-1].sum$_{\texttt{enq-left}}$, blocks[b-1].end$_{\texttt{left}}$+1, blocks[b].end$_{\texttt{left}}$)

406: **return** left.GETENQ(subBlock, i)

407: **else**

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

409: subBlock= right.BSEARCH(sum$_{\texttt{enq}}$, i+right.blocks[b-1].sum$_{\texttt{enq-right}}$, blocks[b-1].end$_{\texttt{right}}$+1, blocks[b].end$_{\texttt{right}}$)

410: **return** right.GETENQ(subBlock, i)

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 <x, y> if $D_{this,b,i} = D_{root,x,y}$.

414: **if** this **is** root **then**

415: **return** <b, i>

416: **else**

417: dir= (parent.left==n)? left: right ▷ check if this node is a left or a right child

418: superBlock= parent.BSEARCH(sum$_{\texttt{deq-dir}}$, i+blocks[b-1].sum$_{\texttt{deq}}$, 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].num$_{\texttt{deq-left}}$ ▷ consider the dequeues from the right child

421: **end if**

422: **return** this.parent.INDEXDEQ(superBlock, i)

423: **end if**

424: **end** INDEXDEQ

Algorithm Leaf

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

602: blk.group= head

603: blocks[head]= blk

604: head+=1

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 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$_{\texttt{left}}$, end$_{\texttt{right}}$ pointers to the first block of the corresponding children.

block[] blocks: array of blocks

int[] super: super[i] stores an approximate index of the superblock of the blocks in blocks whose group field have value i.

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 refrences).

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

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

Question How to remove the notion of time? To say pre(n,i) contains n.blocks[0..i] instead of EST(n,t) which head=i at time t. Is it good? Furthermore, can we remove the notion of established blocks?

Definition 1 (Block)**.** A block is an object storing some statistics, as described in Algorithm Queue. A block in a node's blocklist implicitly represents a set of operations. 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 the index of the `b` is smaller than the index of the `b'`s. 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`.

::headInc **Lemma 2** (head Increment)**.** *Let* `R` *be an instance of* `Refresh` *on node* `n` *that reaches Line* cas *313. After* `R` *terminates* `n.head` *is greater than h, the value read in line* readHead *310 of* `R`*.*

Proof. If Line incrementHead1 318 or incrementHead2 321 are successful then the claim holds, otherwise another process has incremented the head from `h` to `h+1`. □

dPosition **Invariant 3** (headPosition)**.** If the value of `n.head` is `h` then, `n.blocks[i]=null` for `i>h` and $\texttt{n.blocks[i]}\neq\texttt{null}$ for `i<h`.

Proof. The invariant is true initially since 1 is assigned to `n.head` and `n.blocks[x]` is null for every `x`. The truth of the invariant may be affected by writing into `n.blocks` or incrementing `n.head`. We show the invariant still holds after these two changes.

In the algorithm, some value is appended to `n.blocks[]` by writing into `n.blocks[head]` only in Line 313. Writing into `n.blocks[head]` preserves the invariant, since the claim does not talk about `n.blocks[head]`. The value of `n.head` is modified only in lines incrementHead1 318 and incrementHead2 321. Depending on whether the `TryAppend()` in Line cas 313 succeeded or not, we show that the claim holds after the increment of `n.head` in either case. If `n.head` is incremented to h it is sufficient to show $\texttt{n.blocks[h]}\neq\texttt{null}$ to prove the invariant still holds. In the first case the process applied a successful `TryAppend(new,h)` in line okcas 314, which means `n.blocks[h]` is not null anymore. Note that whether incrementHead1 318 or incrementHead1 318 return true or false, after they finish we know that `n.head` has been incremented from the value read in Line readHead 310 (Lemma lem::headInc 2). The failure case is also the same since it means some non-null value has been written into `n.blocks[head]` by some process. □

Explain More

dProgress **Lemma 4** (headProgress)**.** `n.head` *is non-decreasing over time. If* $\texttt{n.blocks[i]}\neq\texttt{null}$ *and* `i.0` *then* $\texttt{n.blocks[i].end}_{\texttt{left}} \geq \texttt{n.blocks[i-1].end}_{\texttt{left}}$ *and* $\texttt{n.blocks[i].end}_{\texttt{right}} \geq \texttt{n.blocks[i-1].end}_{\texttt{right}}$*.*

Proof. The first claim follows trivially from the pseudocode since `n.head` is only incremented in the pseudocode in lines incrementHead1 318 and incrementHead2 321 of `Refresh()` .

Consider the block `b` written into `n.blocks[i]` by `TryAppend()` at Line cas 313. It is created by the `CreateBlock(i)` called at Line invokeCreateBlock 311. Prior to this call to `CreateBlock(i)`, `n.head=i` at Line readHead 310, so `n.blocks[i-1]` is already a non-null value `b'` by Invariant lem::headPosition 3. Thus the `CreateBlock(i-1)` that creates `b'` terminates before `CreateBlock(i)` that creates `b` is invoked. The value written into $\texttt{b.end}_{\texttt{left}}$ at Line endDefLine 333 of `CreateBlock(i)` was read from `n.left.head-1` at Line lastLine 331 of `CreateBlock(i)`. Similarly, the value in $\texttt{n.blocks[i-1].end}_{\texttt{left}}$ was read from `n.left.head-1` during the call to `CreateBlock(i-1)`. Since `n.left.head` is non-decreasing $\texttt{b'.end}_{\texttt{left}} \leq \texttt{b.end}_{\texttt{left}}$. The proof for $\texttt{end}_{\texttt{right}}$ is similar. □

:subblock **Definition 5** (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 `n.blocks[i]` if `b` is a direct subblock of `n.blocks[i]` or a subblock of a direct subblock of `n.blocks[i]`.

Definition 6 (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`.

def::ops **Definition 7** (Operations of a block)**.** A leaf block `b` in a leaf represents `enqueue(x)` if `b.element=x`$\neq$`null`. Else if `b.element=null` b represents a `dequeue()`. The set of operations of block `b` are the operations in the subblocks of `b`. We denote the set of operations of block `b` by `ops(b)`.

We say block `b` is *propagated to node* `n` if `b` is in `n.blocks` or is a subblock of a block in `n.blocks`. We also say `b` contains `op` if `op`$\in$`ops(b)`.

Definition 8. A block `b` in `n.blocks` is *established* at time t if `n.head`$>$ index of `b` at time t. $EST_{n,\ t}$ is the set of established blocks of node `n` at time t.

head **Observation 9.** *Once a block* `b` *is written in* `n.blocks[i]` *then* `n.blocks[i]` *never changes.*

Lemma 10. *Every block has at most one direct superblock.*

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

lastLine, pr

append **Corollary 11** (No Duplicates)**.** *If* `op` *is in* `n.blocks[i]` *then there is no* `j`$\neq$`i` *such that* `op`$\in$`ops(n.blocks[j])`.

shedOrder **Lemma 12** (establishedOrder)**.** *If time* $t <$ *time* t'*, then* `ops(EST`$_{\texttt{n},\ \texttt{t}}$`)`$\subseteq$ `ops(EST`$_{\texttt{n},\ \texttt{t}'}$`)`.

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 grow. □

useless?

ordering **Definition 13** (Ordering of operations inside the nodes)**.** ▶ Note that processes are numbered from 1 to p, left to right in the leaves of the tree and from Lemma 21 we know there is at most one operation from each process in a given block. blockSize

- Prefix of the *op* in the sequence of operations S, is the operations strictly before *op*.
- $E(n, b)$ is the sequence of enqueue operations in `ops(n.blocks[b])` ordered by their process id.
- $E_{n,b,i}$ is the ith enqueue in $E(n, b)$.
- $D(n, b)$ is the sequence of dequeue operations in `ops(n.blocks[b])` ordered by their process id.
- $D_{n,b,i}$ is the ith enqueue in $D(n, b)$.
- Order of the enqueue operations in the node n: $E(n) = E(n, 1).E(n, 2).E(n, 3)...$
- $E_{n,i}$ is the ith enqueue in $E(n)$.
- Order of the dequeue operations in the node n: $D(n) = D(n, 1).D(n, 2).D(n, 3)...$
- $D_{n,i}$ is the ith dequeue in $D(n)$.
- 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 need ordering of enqueues and dequeues among the operations of their own type. Since `GetENQ()` *only searches among enqueues and* `IndexDEQ()` *works on dequeues.*

ueRefresh **Lemma 14** (trueRefresh)**.** *Let t_i be the time an instance* R *of* n.Refresh() *is invoked and t_t be the time it terminates. If the* TryAppend(new, s) *of* R *returns* true*, then* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$.

Proof. Since TryAppend returns true a block new is written into n.blocks[h] in Line cas 313.

We show $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$. Let h be the value n.Refresh() reads from n.head at line readHead 310, $\texttt{h}_{\texttt{left,i}}$ be the value of n.left.head at $\texttt{t}_\texttt{i}$ and $\texttt{h}_{\texttt{left,read}}$ be the value read from n.left.head-1 at line lastLine 331. $\texttt{end}_{\texttt{left}}$ field of the block returned by CreateBlock(i) is $\texttt{h}_{\texttt{left,read}}$. By lines prevLine 332 and lastLine 331 the new block in n.blocks[h] contains $\texttt{n.left.blocks[n.blocks[h-1].end}_{\texttt{left}}\texttt{+1..h}_{\texttt{left,read}}\texttt{]}$. Since left.head is read after $\texttt{t}_\texttt{i}$ then $\texttt{h}_{\texttt{left,read}} > \texttt{h}_{\texttt{left,i}}$ which means $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(n.left.blocks [0..h}_{\texttt{left,read}}\texttt{])}$. After the successful TryAppend in line cas 313 we know all blocks in $\texttt{n.left.blocks[0..h}_{\texttt{left,read}}\texttt{-1}$ are subblocks of n.blocks[0..h] by the definition of subblock. At $\texttt{t}_\texttt{t}$ we have n.head>h by Lemma lem::headProgress 4. So n.blocks[1..h] are in $\texttt{EST}_{\texttt{n,t}_\texttt{t}}$ by definition of EST. Note that after line incrementHead2 321 we are sure that the head is incremented by Lemma lem::headInc 2) which means n.head=h+1 at t_t so the new block is established at t_t and the new block contains the new operations which is what we wanted to show. The proof for $\texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$ is the same. □

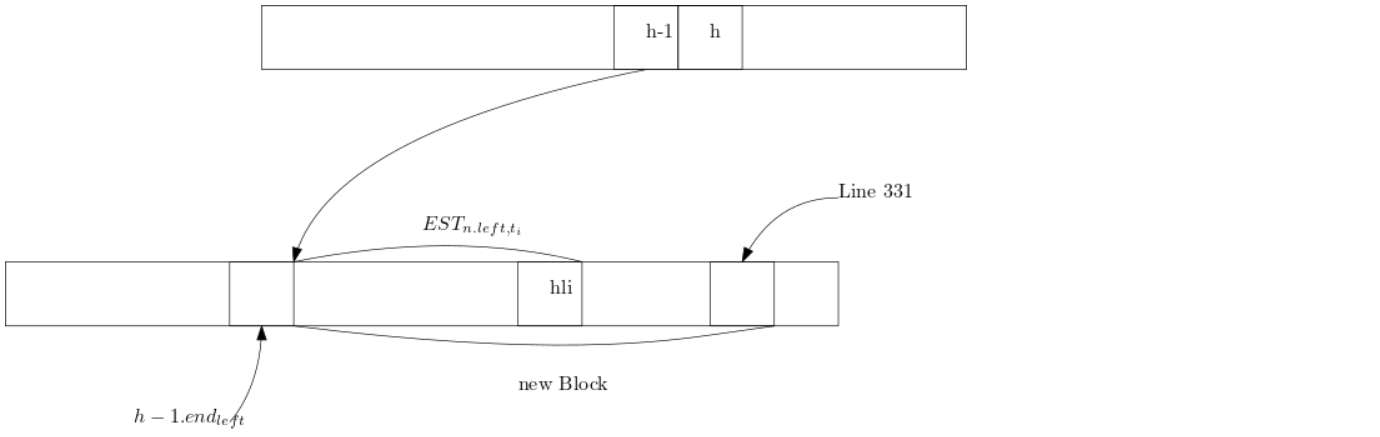


Figure 1: New established operations of the left child are in the new block.

ueRefresh **Lemma 15** (Stronger True Refresh)**.** *Let t_i be the time an instance of* n.Refresh() *read the head (Line readHead 310) and t_t be the time its* TryAppend(new, s) *terminates with and returns* true *(Line cas 313). We have* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(n.blocks)}$.

Lemma 16 (Double Refresh)**.** *Consider two consecutive instances R_1, R_2 of* `Refresh()` *on* `n` *by process p. Let t_1 be the time R_1 is invoked and t_2 be the time R_2 terminated. If R_1 and R_2 both fail and return false, then 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 cas 313 of R_1 or R_2 returns `true`, then the claim is held by Lemma lem::trueRefresh 14. Let R_1 read i and R_2 read $i+1$ from Line readHead 310. 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 incrementHead 318 or 321 before readHead 310 of R_2, from Lemma lem::trueRefresh 14 by the end of this instance $\texttt{ops(EST}_{\texttt{n.left, }t_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, }t_1}\texttt{)}$ has been propagated.

Since R_2's `TryAppend()` returns `false` then there is another successful instance R_2' of `n.Refresh()` that has done `TryAppend()` successfully into `n.blocks[i+1]` before R_2 tries to append. Since R_2' creates the block after reading the value $i+1$ from `n.head`(Line readHead 310) and R_1 reads the value i from `n.head` and the `head`'s value is increaing by Lemma lem::headProgress 4 then $t_{R2'} \overset{\text{readHead}}{>} t_{R1} \overset{\text{readHead}}{>} t_1$ (See Figure fig::doubleRefres 2). By Lemma lem::prectrueRefresh 15 after R_2''s `CAS` we have $\texttt{ops(EST}_{\texttt{n.left, }t_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, }t_1}\texttt{)} \subseteq \texttt{ops(n.blocks)}$. Also by Lemma lem::headInc 2 on R_2 the value of `n.head` head is more than $i+1$ after R_2 terminates, so the block appended by R_2' to `n` is established by then (`n.head>=i+2>i+1`). To summarize t_1 is before R_2''s `read n.head` and R_2''s successful `CAS` is before R_2's termination. So by Lemma lem::prectrueRefresh 15 $\texttt{ops(EST}_{\texttt{n.left, }t_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, }t_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, }t_2}\texttt{)}$. □

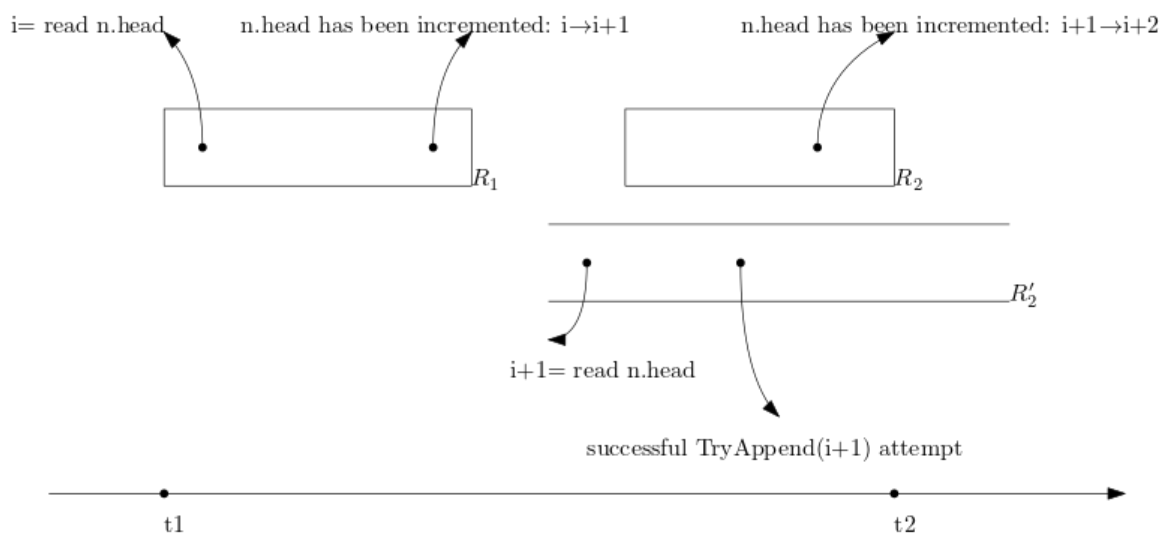


Figure 2: $t1 < r_1$ reading head < incrementing `n.head` from i to $i+1 < R_2'$ reading head < `TryAppend(i+1)` < incrementing `n.head` from $i+1$ to $i+2 < t2$

this chain with more depth should be in the proof

Definition 17. $t_{before\ line}$ is the immediate time before running Line *line*. $t_{before\ line}$ is the immediate time after running Line *line*.

Corollary 18. $\texttt{ops(EST}_{\texttt{n.left, }t_{\text{before 602}}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n.right, }t_{\text{before 602}}}\texttt{)} = \texttt{ops(EST}_{\texttt{n, }t_{\text{after 603}}}\texttt{)}$ secondRefresh

Proof. If the first `Refresh()` in line firstRefresh 302 returns `true` then by Lemma lem::trueRefresh 14 the claim holds. Also if first `Refresh()` failed and the second `Refresh()` succeeded the claim still holds by Lemma lem::trueRefresh 14. Finally, if both failed the claim is satisfied by Lemma doubleRefresh 16. □

lyRefresh **Corollary 19** (Propagate Step)**.** *All operations in* `n`*'s children's established blocks before running line* firstRefresh *302 of a* `Propagate` *routine are guaranteed to be in* `n`*'s established blocks after line* secondRefresh *303.*

Proof. If firstRefresh 302 or secondRefresh 303 succeed, the claim is true by Lemma lem::trueRefresh 14. Otherwise Lines firstRefresh 302 and secondRefresh 303 satisfy the preconditions of Lemma doubleRefresh 16. □

actlyOnce **Corollary 20.** *After* `Append(blk)` *finishes* `ops(blk)`$\subseteq$`ops(root.blocks[x])` *for exactly one* `x`*.*

Proof. After `Append(blk)`'s termination, `blk` is in `root.blocks` since `blk` is established in the leaf it has been added to. By applying Lemma doublyRefresh 19 inductively it is propagated up to the root. Finally Lemma append 11 shows only one block in the root contains `blk`. □

blockSize **Lemma 21** (Block Size Upper Bound)**.** *Each block contains at most one operation of each processs.*

Proof. To derive a contradiction, assume there are 2 operations op_1 and op_2 from process p in block b in node n. WLOG op_1 is invoked earlier than op_2. A process cannot invoke more than one operations concurrently, so one operation has to be finished before the other starts. So op_1 has terminated before op_2 started. By Corollary 20 lem::appendExactlyOnce before appending op_2 to the tree op_1 exists in every node from the path of p's leaf to the root. Because op_1's `Append` is finished before op_2's `Append` starts. So there is some block b' before b in n containing op_1. op_1 existing in b an b' contradicts Lemma 11. append □

ocksBound **Lemma 22** (Subblocks Upperbound)**.** *Each block has at most p direct subblocks.*

Proof. It follows directly from Lemma 21 blockSize and the observation that each block appended to the tree contains at least one operation, induced from Line 312. addOP □

get **Lemma 23** (Get correctness)**.** *If* `n.blocks[b].num`$_{\texttt{enq}}$$\geq$`i` *then* `n.GetENQ(b,i)` *returns* $E_{n,b,i}$.

Proof. We are going to prove this lemma by induction on the height of node `n`. The base case, where `n` is a leaf, is straightforward (Line getBaseCase 403). Leaf blocks each contain exactly one operation, so only `n.GetENQ(b,1)` can be called where `n` is a leaf. `n.GetENQ(b,1)` returns the `element` of the `enqueue` operation stored in the bth block of leaf `n`.

For the induction step we prove `n.GetENQ(b,i)` returns $E_{n,b,i}$, if `n.child.GetENQ(b,i)` returns $E_{n.child,b,i}$. In the Line leftOrRight 404 it is decided for the non-leaf nodes that the `i`th enqueue in `b`th block of internal node `n` is in the `n.blocks[b]`'s left child or right child subblocks. From Definition ordering 13 of $E(n,b)$ we know enqueue operations in a block are ordered by their process id and since the leaves of the tree are ordered by process id from left to right, thus operations from the left subblocks come before operations from the right subblocks in a block (See Figure figGet 3). Furthermore the `num`$_{\texttt{enq-left}}$ field in a block stores the number of `enqueue()` operations from the blocks's subblocks in the left child of `n`. So ith enqueue operation is propagated from the right child if `i` is greater than `b.num`$_{\texttt{enq-left}}$. otherwise we should search for the ith enqueue in the left child. By definition def::obs and def::subblock 7 and 5 we need to search in subblocks of `n.blocks[b]` from the range `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` $\cup$ `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`.

If the ith `enqueue` of `n.blocks[b]` is in the left child it would be ith enqueue in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` by Definition def::subblock 5. Also we know there are $eb = n.blocks[b-1].sum_{enq-left}$ enqueues in the blocks before this range, so $E_{n,b,i}$ is $E_{n.left,i+eb}$ which is $E_{n.left,b',i'}$ for some b' and i'. We can compute b' search for $i+eb$th enqueue in `n.left` and i' is `i+eb-n.left.blocks[`$b'-1$`].sum`$_{\texttt{enq}}$. The parameters in leftChildGet 405 are for searching $E_{n.left,i+eb}$ in `n.left.block` in the expected range of blocks, so this `BSearch` returns the index of the subblock containing $E_{n,b,i}$.

Else if the enqueue we are looking for is in the right child then there are `n.blocks[b].num`$_{\texttt{enq-left}}$ enqueues ahead of it in `n.blocks[b]` but not in `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`. So we need to search for `i-n.blocks[b].num`$_{\texttt{enq-left}}$`+ n.blocks[b-1].sum`$_{\texttt{enq-right}}$ (Line rightChildGet 409). Other parameters are assigned similar for the left child. So in both cases the direct subblock containing $E_{n,b,i}$ is computed in Lines leftChildGet 405 and rightChildGet 409.

Finally, `n.child.GetENQ()` is invoked on the subblock containing $E_{n,b,i}$ which returns $E_{n,b,i}$ by the hypothesis of the induction. □

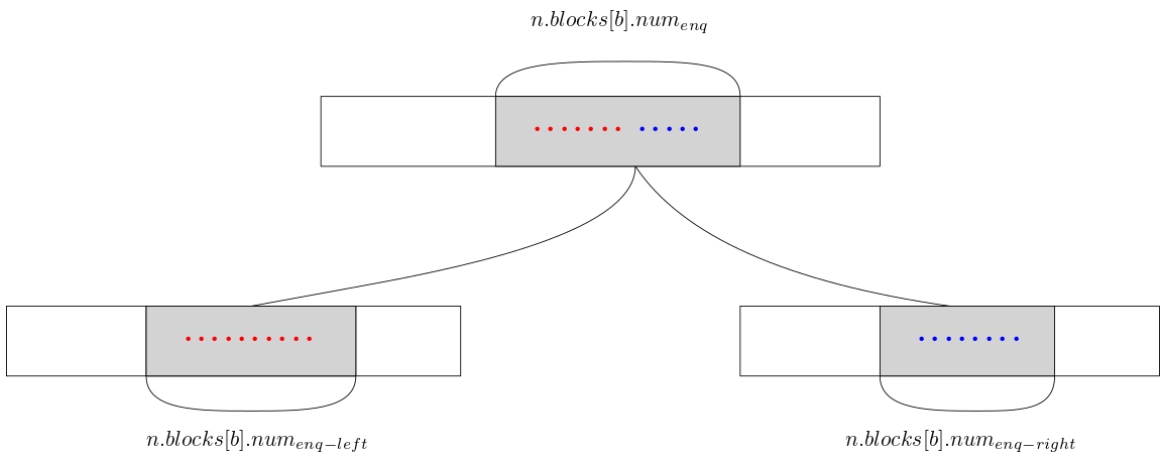


Figure 3: The number and ordering of the enqueue operations propagated from the left and the right child to `n.blocks[b]`. Enqueue operations from the left subblocks (colored red), are ordered before the enqueue operations from the right child (colored blue).

figGet

dsearch **Lemma 24** (DSearch correctness)**.** *Assume* root.blocks[end].sum$_{\texttt{enq}}\geq$e *and* $E_{root,e}$*'s* element *is the response to some* Dequeue() *operation in* root.blocks[end]. DSearch(e, end) *returns* <b, i> *such that* $E_{root,b,i} = E_{root,e}$.

Proof. It is trivial to see that the doubling search from root.blocks[end] to root.blocks[0] will find $E_{root,e}$ eventually. Because root.blocks[].sum$_{\texttt{enq}}$ is an increasing value from 0 to some value greater than e. So there is a b that root.blocks[b].sum$_{\texttt{enq}}> e$ but root.blocks[b-1].sum$_{\texttt{enq}}< e$.

First we show end-b$\leq$ 2$\times$root.blocks[b].size +root.blocks[end].size +1. From line addOP 312, we know that size of the every block in the tree is greater than 0. So each block in root.blocks[b..end] contains at least one Enqueue or at least one Dequeue. Suppose there were more than root.blocks[b].size Dequeues in root.blocks[b+1..end-1]. Then the queue would become empty at some point after blocks[b]'s last operations and before root.blocks[end]'s first operation. Which means the response to to a Dequeue in root.blocks[end] could not be in $E(n, b)$. Furthermore since the size of the queue would become root.blocks[end].size after the root.blocks[end], there cannot be more than root.blocks[b].size + root.blocks[end].size Enqueues. Because there can be at most root.blocks[b].size Dequeues and the final size is root.blocks[end].size . Overall there can be at most 2$\times$root.blocks[b].size + root.blocks[end].size operations in root.blocks[b+1..end-1] and since each block size is ≥ 1 thus there are at most 2$\times$root.blocks[b].size + root.blocks[end].size blocks in between root.blocks[b] and root.blocks[end]. So end-b$\leq$ 2$\times$root.blocks[b].size +root.blocks[end].size +1. See Figure end-b 5.

Now that we know there are at most root.blocks[b].size +root.blocks[end].size blocks in between root.blocks[b] and root.blocks[end] then with doubling search in $\Theta\big(\log($root.blocks[b].size +root.blocks[end].size $)\big)$ steps we reach start=c that the root.blocks[c].sum$_{\texttt{enq}}$ is less than e and end-c is not more than 2$\times$root.blocks[b].size +root.blocks[end].size . Beause otherwise, then (end-c)/2 satisfied the root.blocks[(end-c)/2].sum$_{\texttt{enq}}<$e. In line doubling 804 the differnece between end and start is doubled. See Figure fig::doubling 4.

After computing b, the value i is computed via the definition of sum$_{\texttt{enq}}$ in constant time (Line DSearchComputei 807). So the routine non constant part is the binary search which takes $\Theta(\log$root.blocks[b].size +root.blocks[end].size $))$ steps from the first paragraph.

□

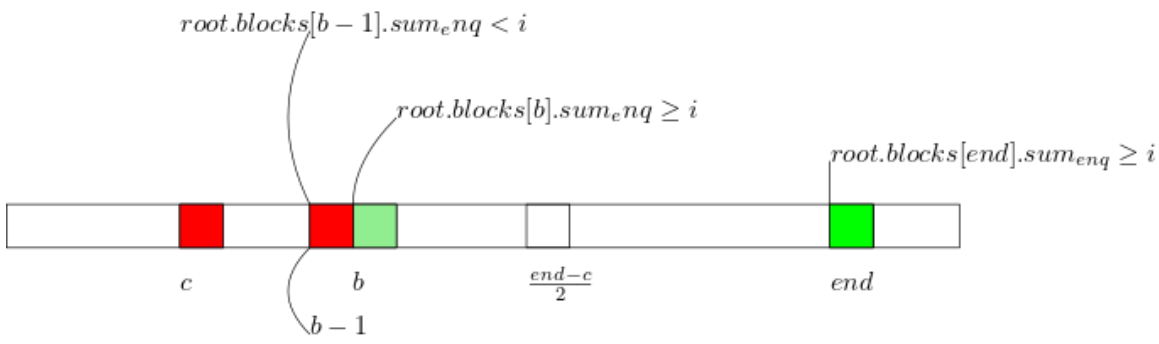


Figure 4: Distance relations between b, c, end

:doubling

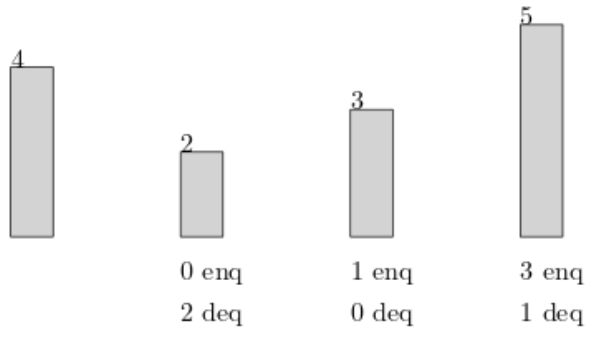


Figure 5: The number written on top of the bars is the queue size. the first block is b and the last block is end.

end-b

Lemma 25 (Index correctness)**.** `n.IndexDEQ(b,i)` *returns the rank in* $D(root)$ *of* $D_{n,b,i}$.

Proof. We will prove this by induction on the distance of `n` from the `root`. We can see the base case `root.IndexDEQ(b,i)` is trivial (Line indexBaseCase 415). In the non-root nodes `n.IndexDEQ(b,i)` computes the superblock of the ith Dequeue in the bth block of `n` in `n.parent` by Lemma superBlock computeSuper 26 (Line 418). After that the order in $D(n.parent, superblock)$ is computed and `index()` is called on `n.parent` recursively. Then if the operation was propagated from the right child the number of dequeues from the left child are added to it (Line considerRight 420), because the left child operations come before the right child operations (Definition ordering 13). □

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? Make sure to show preconditions of all invocation of `BSearch` *are satisfied.*

uperBlock **Lemma 26** (Computing SuperBlock)**.** *After computing line* computeSuper *418 of* `n.IndexDEQ(b,i)`, `n.parent.blocks[superblock]` *contains* $D(n, b, i)$.

Proof. Lemmas 28,29,30,31. □

Lemma 27. *Value read for* `super[b.group]` *in line 418 is not null.*

Proof. Values $\texttt{np}_{\texttt{dir}}$ read in lines setNP 337, `super` are set before incrementing in lines setSuperNP 315,316. So before incrementing $\texttt{num}_{\texttt{propagated}}$, $\texttt{super[num}_{\texttt{propagated}}\texttt{]}$ is set so it cannot be null while reading. □

Lemma 28. `super[]` *preserves order from child to parent; i.e. if in node* `n` *block* `b` *is before* `c` *then* `b.group` $\leq$ `c.group`

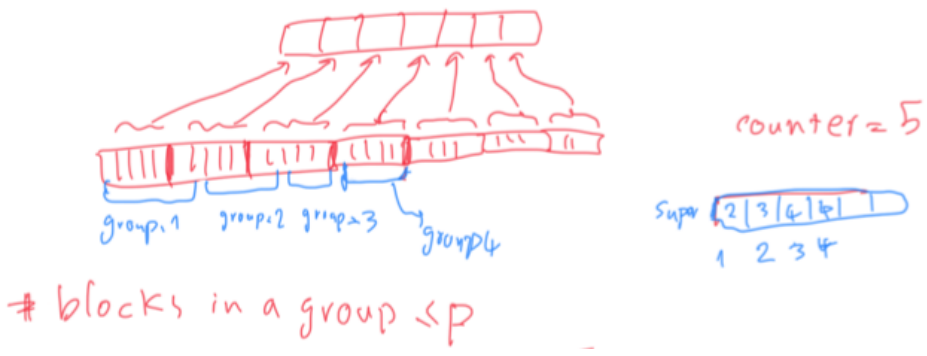
Proof. Line setGroup 329. Since $\texttt{num}_{\texttt{propagated}}$ is increasing. □

Lemma 29. *Let* `b, c` *be in node* `n`*, if* `b.group` $\leq$ `c.group` *then* `super[b.group]` $\leq$ `super[c.group]`

Proof. Line setSuper 315. □

Lemma 30. *The number of the* `blocks` *with* `group=i` *in a node is* $\leq p$.

Proof. For the sake of simplicity we assumed all the blocks are propagated from the left child. □

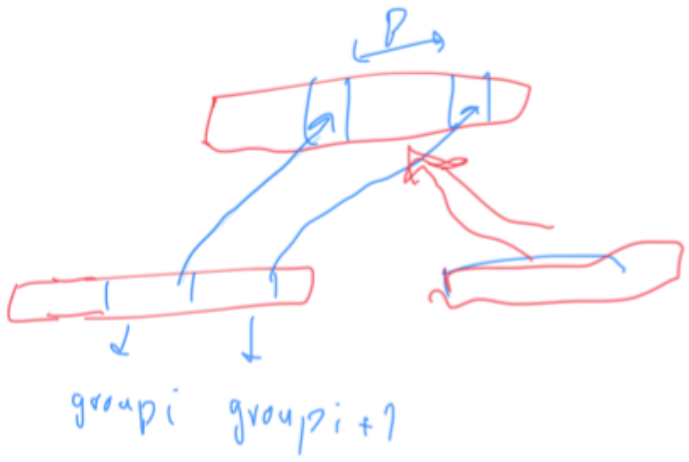


Lemma 31. `super[i+1]-super[i]` $\leq p$

Proof. 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[sup[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. □

erCounter **Lemma 32** (super property)**.** *If* `super[i]` $\neq$ `null` *in node* `n`*, then* `super[i]` *is the index of the superblock of a block with* `time=i` *in* `n.parent.blocks`.

uperRange **Lemma 33.** *Superblock of* `b` *is within range* $\pm 2p$ *of the* `super[b.time]`.



Proof. `super[i]` is the index of the superblock of a block containing block b, followed by Lemma 32. `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:

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

□

Lemma 34. *Search in each level of* `IndexDeq()` *takes* $O(\log p)$ *steps.*

Proof. Show preconditions are satisfied and the range is p. □

Definition 35. Assume the operations in L are applied on an empty queue. If element of `enqueue e` is the response to `dequeue d` then we say `R(d)=e`. If `d` 's response id `null` (queue is empty) then `R(d)=null`.

Definition 36. In an execution on a queue, the dequeue operations that return some value are called *non-null dequeues*.

responseToADeq **Observation 37.** *kth non-null dequeue in an execution returns the* `element` *of kth enqueue.*

correctness **Lemma 38.** `root.blocks[b].size` *is the size of the queue if the operations in the prefix for the bth block in the root are applied with the order of L.*

Proof. need to say? :: If the size of a queue is greater than 0 then a `Dequeue()` would decrease the size of the queue, otherwise the size of the queue remains 0. By definition ordering 13 enqueue operations come before dequeue operations in a block in L.

We prove the claim by induction on `b`. Base case `b=0` is trivial since the queue is initialy empty and `root.blocks[0].size=0`. For `b=i` we are going to use the hypothesis for `b=i-1`. If there are more than `root.blocks[i-1].size+ root.blocks[i].sum`$_{\texttt{enq}}$ dequeue operations in `root.blocks[i]` then the queue would become empty after `root.blocks[i]`. Otherwise we can compute the size of the queue after bth block using with this equality `root.blocks[b].size= root.blocks[b-1].size+ root.blocks[b].sum`$_{\texttt{enq}}$`- root.blocks[b].sum`$_{\texttt{deq}}$ (Line computeLength 342). See Table qhistory 1 for an example of running some blocks of operations on an empty queue. □

numberOfNND **Lemma 39** (Duality of #non-null dequeues and `block.size`)**.** *If the operations are applied with the order of L, the number of non-null dequeues in the prefix for a block* `b` *is* `b.sum`$_{\texttt{enq}}$`-b.size`

Proof. There are `b.sum`$_{\texttt{enq}}$ enqueue operations in the prefix for `b`, then the size of the queue after the prefix for `b` is `#enqs - #non-null dequeues` in the prefix for `b`, by Observation 35. So `#non-null dequeues` is `b.sum`$_{\texttt{enq}}$`-b.size` . The correctness of the `block.size` field is shown in Lemma sizeCorrectness 38. □

nullReturn **Lemma 40.** `R(D`$_{\texttt{root,b,i}}$`)` *is* `null` *iff* `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0`.

computeHead **Lemma 41** (Computing Response)**.** `FindResponse(b,i)` *returns* `R(D`$_{\texttt{root,b,i}}$`).element`.

Proof. First note that by Definition ordering 13 the linearization ordering of operations will not change as new operations come so instead of talking about the linearization of operations before the $E_{root,b,i}$ we talk about what if the whole operation in the linearization are applied on a queue.

$D_{root,b,i}$ is $D_{root,root.blocks[b-1].sum_{deq}+i}$ from the definition ordering 13 and sum_{enq}. $D_{root,b,i}$ returns `null` if `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0` by Lemma nullReturn 40 (Line checkEmpty 220). Otherwise if it is d'th non-null dequeue in L it returns d'th enqueue by Observation responseToADeq 37 . By Lemma numberOfNND 39 there are `root.blocks[b-1].sum`$_{\texttt{enq}}$ `- root.blocks[b-1].size` non-null dequeue operations before prefix for `root.blocks[b-1]`. Note that the dequeues in `root.blocks[b]` before the ith dequeue are non-null dequeues. So the response is $E_{root,i-root.blocks[b-1].size+root.blocks[b-1].sum_{\texttt{deq}}}$ (Line computeE 222). See figure computeResponseDetail 6.

After computing `e` we can find `b,i` such that $E_{root,b,i} = E_{root,e}$ using `DSearch` and then find its `element` using `GetEnq` (Line findAnswer 223). □

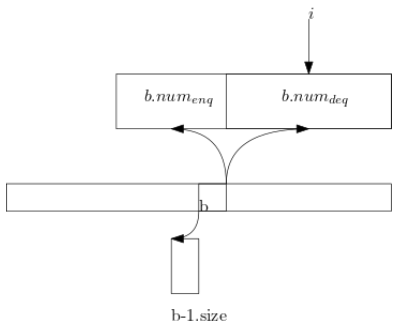


Figure 6: The position of $E_{root,b,i}$.

responseDetail

	DEQ()	ENQ(5), ENQ(2), ENQ(1), DEQ()	ENQ(3), DEQ()	ENQ(4), DEQ(), DEQ(), DEQ(), DEQ()
#enqueues	0	3	1	1
#dequeues	1	1	1	4
#non-null dequeues	0	1	2	5
size	0	2	2	0

qhistory

Table 1: An example of root blocks fields. Blocks are from left to right and operations in the blocks are also from the left to right.

Theorem 42 (Main)**.** *The queue implementation is linearizable.*

Proof. We choose L in Definition ordering 13 to be linearization ordering of operations and prove if we linearize operations as L the queue works consistently. □

Lemma 43 (satisfiability)**.** *L can be a linearization ordering.*

Proof. To show this we need to say if in an execution, $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts then $\texttt{op}_1$ is linearized before $\texttt{op}_2$. If $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts it means $\texttt{op}_1$`.Append()` is terminated before $\texttt{op}_2$`.Append()` starts. From Lemma append 11 $\texttt{op}_1$ is in `root.blocks` before $\texttt{op}_2$ propagates so $\texttt{op}_1$ is linearized before $\texttt{op}_2$ by Definition ordering 13.

Once some operations are aggregated in one block they will be propagated together up to the root and we can linearize them in any order among themselves. Furthermore in L we arbitrary choose the order to be by process id, since it makes computations in the blocks faster . □

Lemma 44 (correctness)**.** *If operations are applied as L on a sequential queue, the sequence of the responses would be the same as our algorithm.*

Proof. Old parts to review We show that the ordering L stored in the root, satisfies the properties of a linearizable ordering.

1. If op_1 ends before op_2 begins in E, then op_1 comes before op_2 in T.

   ▶ This is followed by Lemma append 11. The time op_1 ends it is in root, before op_2, by Definition ordering 13 op_1 is before op_2.

2. Responses to operations in E are same as they would be if done sequentially in order of L.

   ▶ Enqueue operations do not have any response so it does no matter how they are ordered. It remains to prove Dequeue d returns the correct response according to the linearization order. By Lemma computeHead 41 it is deduced that the head of the queue at time of the linearization of d is computed properly. If the Queue is not empty by Lemma get 23 we know that the returning response is the computed index element.

□

Lemma 45 (Amortized time analysis)**.** `Enqueue()` *and* `Dequeue()`*, each take* $O(\log^2 p + \log q)$ *steps in amortized analysis. Where* p *is the number of processes and* q *is the size of the queue at the time of invocation of operation.*

Proof. `Enqueue(x)` consists of creating a `block(x)` and appending it to the tree. The first part takes constant time. To propagate `x` to the root the algorithm tries two `Refresh`es in each node of the path from the leaf to the root (Lines 302, 303). We can see from the code that each `Refresh` takes constant number of steps since creating a block is done in constant time and does $O(1)$ `CAS`es. Since the height of the tree is $\Theta(\log p)$, `Enqueue(x)` takes $O(\log p)$ steps.

A `Dequeue()` creates a block with null value element, appends it to the tree, computes its order among enqueue operations, and returns the response. The first two part is similar to an `Enqueue` operation. To compute the order of a `dqueue` in $D(n)$ there are some constant steps and `IndexDeq()` is called. `IndexDeq` does a search with range p in each level (Lemma 33) which takes $O(log^2 p)$ in the tree. In the `FindResponse()` routine `DSearch()` in the root takes $\Theta(\log($`root.blocks[b].size +root.blocks[end].size` $)$ by Lemma 24, which is $O($log size of the queue when `enqueue` is invoked$)$ + log size of the queue when `dequeue` is invoked). Each search in `GetEnq()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma 22), so `GetEnq()` takes $O(\log^2 p)$ steps.

If we split `DSearch` time cost between the corresponding `Enqueue`, `Dequeue`, in amortized we have `Enqueue` takes $O(\log p + q)$ and `Dequeue` takes $O(\log^2 p + q)$ steps. □

Lemma 46 (CASes invoked)**.** *An* `Enqueue()` *or* `Dequeue()` *operation, does at most* $4 \log p$ `CAS` *operations.*

Proof. In each height of the tree at most 2 times `Refresh()` is invoked and every `Refresh()` has 2 `CAS`es, one in Line 313 and one in Lines 318 or 321. □