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
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`

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

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

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

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

tChildGet 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$_{\texttt{enq-left}}$

tChildGet 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**

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

416: **else**

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

puteSuper 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**

iderRight 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.

appendEnd 602: head+=1

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

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. It 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`.

dPosition **Invariant 2** (headPosition)**.** If the value of `n.head` is `h` then, `n.blocks[i]=null` for `i>h` and `n.blocks[i]`$\neq$`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`.

Some value is written into `n.blocks[head]` only in Line 313. It is obvious that writing into `n.blocks[head]` presereves the invariant. The value of `n.head` is modified only in lines 318, 321. Depending on wether the `TryAppend()`in Line 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]`$\neq$`null` to prove the invariant still holds. In the first case the process applied a successful `TryAppend(new,h)` in line 314, which means `n.blocks[h]` is not null anymore. Note that wether 318 returns true or false after Line `n.head` we know has been incremented from Line 310. The failure case is also the same since it means some value is written into `n.blocks[head]` by some process. □

Explain More

dProgress **Lemma 3** (headProgress)**.** `n.head` *is non-decreasing over time and* `n.blocks[i].end`$_{\texttt{left}} \geq$ `n.blocks[i-1].end`$_{\texttt{left}}$ *and* `n.blocks[i].end`$_{\texttt{right}}$ $\geq$ `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 the lines 318 and 321 of the `Refresh()` .

Consider the block `b` written into `n.blocks[i]` by `TryAppend()` at Line 313. It is created by the `CreateBlock()` called at Line 311. Prior to this call to `CreateBlock()`, `n.head=i` at Line 310, so `n.blocks[i-1]` is already a non-null value `b'` by invariant 2. Thus the `CreateBlock(i-1)` that creates `b'` terminates before `CreateBlock(i)` that creates `b` is invoked. The value written into `b.end`$_{\texttt{left}}$ at Line 333 of `CreateBlock()` was read from `n.left.head` at Line 331 of `CreateBlock`. Similarly, the value in `n.blocks[i-1].end`$_{\texttt{left}}$ was read from `n.left.head`. Since `n.left.head` is non-decreasing `b'.end`$_{\texttt{left}} \leq$ `b.end`$_{\texttt{left}}$. The proof for `end`$_{\texttt{right}}$ is similar. □

:subblock **Definition 4** (Subblock)**.** Block `b` is a *direct subblock* of `n.blocks[i]` if it is in `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`$_{\texttt{right}}$`]` (The `end`$_{\texttt{left}}$`, end`$_{\texttt{right}}$ fields are written in 333 of `CreateBlock()`). 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 5 (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 6** (Operations of a block)**.** A leaf block `b` in a leaf represents operation `x` if `b.element=x`$\neq$`null`. 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 7. A block b in n.blocks is *established* at time t if the last value written into n.head before t is greater than the index of b in n.blocks at time t. $EST_{n,\ t}$ is the set of established blocks of node n at time t.

head **Observation 8.** *Once a block* b *is written in* n.blocks[i] *then* n.blocks[i] *never changes.*

Lemma 9. *Every block has 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 $\texttt{n.left.blocks[n.blocks[i-1].end}_{\texttt{left}}\texttt{..n.blocks[i].end}_{\texttt{left}}\texttt{]}$ and $\texttt{n.right.blocks[n.blocks[i-1].end}_{\texttt{right}}\texttt{..n.blocks[i]}$ (Lines lastLine, pr ??). Since $\texttt{end}_{\texttt{left}}\texttt{,end}_{\texttt{right}}$ are non-decreasing, so the range of the subblocks of n.blocks[i] which is $\texttt{(n.blocks[i-1].end}_{\texttt{dir}}\texttt{+1..n.blocks[i]}$ does not overlap with the range of the subblocks of n.blocks[i-1]. □

append **Corollary 10** (No Duplicates)**.** *If* op *is in* n.blocks[i] *then there is no* j≠i *such that* op∈ops(n.blocks[j])*.*

shedOrder **Lemma 11** (establishedOrder)**.** *If time* $t <$ *time* t'*, then* $\texttt{ops(EST}_{\texttt{n},\ \texttt{t}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n},\ \texttt{t}'}\texttt{)}$*.*

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

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.

::headInc **Lemma 12** (head Increment)**.** *If an n.Refresh instance reaches Line 313 instance and reads head=h (310) after it terminates head is greater than h.*

Proof. If Line 318 or 318 succeeded the claim holds, otherwise another process has incremented the head. □

ueRefresh **Lemma 13** (trueRefresh)**.** *Let t_i be the time an instance of* `n.Refresh()` *is invoked and t_t be the time it terminates. Suppose the* `TryAppend(new, s)` *of the* `n.Refresh()` *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. From Lemma 11 we know that $\texttt{ops(EST}_{\texttt{n, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$. So it remains to show the operations of $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)}$ - $\texttt{ops(EST}_{\texttt{n, t}_\texttt{i}}\texttt{)}$, which we call *new operations*, are all in $\texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$. If `TryAppend` returns true a block `new` is written into `n.blocks[h]` (Line 313). We show $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$. The proof for the right child's claim is the same. Let `n.left.head` at $\texttt{t}_\texttt{i}$ be `hli`. Let `n.Refresh()` read `head` equal to h(Line 310). By the lines 332,331 the `new` block in `n.blocks[h]` contains $\texttt{n.left.blocks[n.blocks[h-1].end}_{\texttt{left}}\texttt{+1..left.head]}$. Since `left.head` is read after $\texttt{t}_\texttt{i}$ then $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(n.left.blocks[0..left.head])}$. By Lemma 11 $\texttt{ops(n.left.blocks[0..n.blocks[h-1].end}_{\texttt{left}}\texttt{])} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$. Since after line 321 we are sure that the `head` is incremented (Lemma 12) and `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.

□

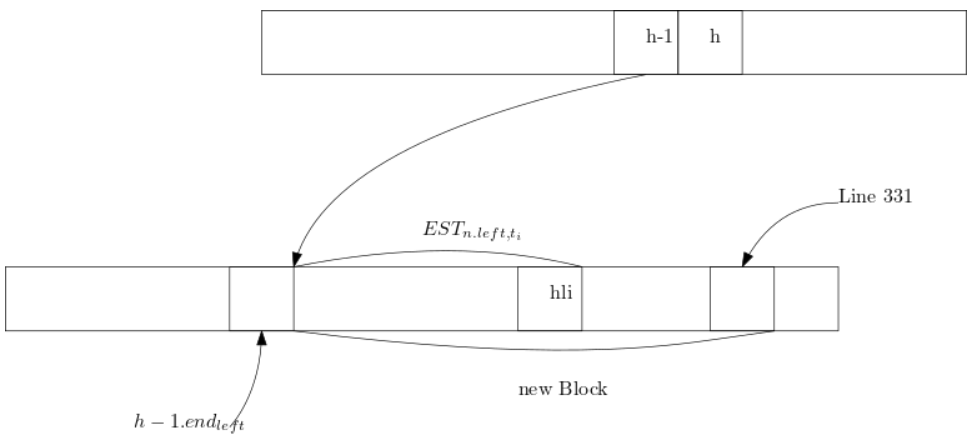


Figure 1: New established operations of the left child are in the new block.

ueRefresh **Lemma 14** (Precise True Refresh)**.** *Let t_i be the time an instance of* `n.Refresh()` *read the head (Line 310) and t_t be the time its* `TryAppend(new, s)` *terminates with and returns* `true` *(Line 313). We have* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(n.blocks)}$.

leRefresh **Lemma 15** (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* $\texttt{ops(EST}_{\texttt{n.left, }t_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, }t_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, }t_2}\texttt{)}$.

Proof.

If Line cas 13 of R_1 or R_2 returns `true`, then the claim is held by Lemma lem::trueRefresh 13. Let R_1 read i and R_2 read $i+1$ from Line readHead 10. If R_2 reads some value greater than $i+1$ in Line readHead 10 it means a successful instance of `Refresh()` started after Line readHead 10 of R_1 and finished its Line incrementHead1 18 or incrementHead2 21 before readHead 10 of R_2, from Lemma lem::trueRefresh 13 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. 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. By Lemma lem::prectrueRefresh 14 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 12 of R_2 head is more than $i+1$ after R_2ś incrementHead2 21 line, so the block appended by R_2' to `n` is established by then. To summarized t_1 is before R_2''s read head and R_2''s `CAS` is before R_2's termination. So $\texttt{ops(EST}_{\texttt{n.left, }t_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, }t_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, }t_2}\texttt{)}$. □

last sentence need more detail and should be earlier. define i and tell why R2prime exists

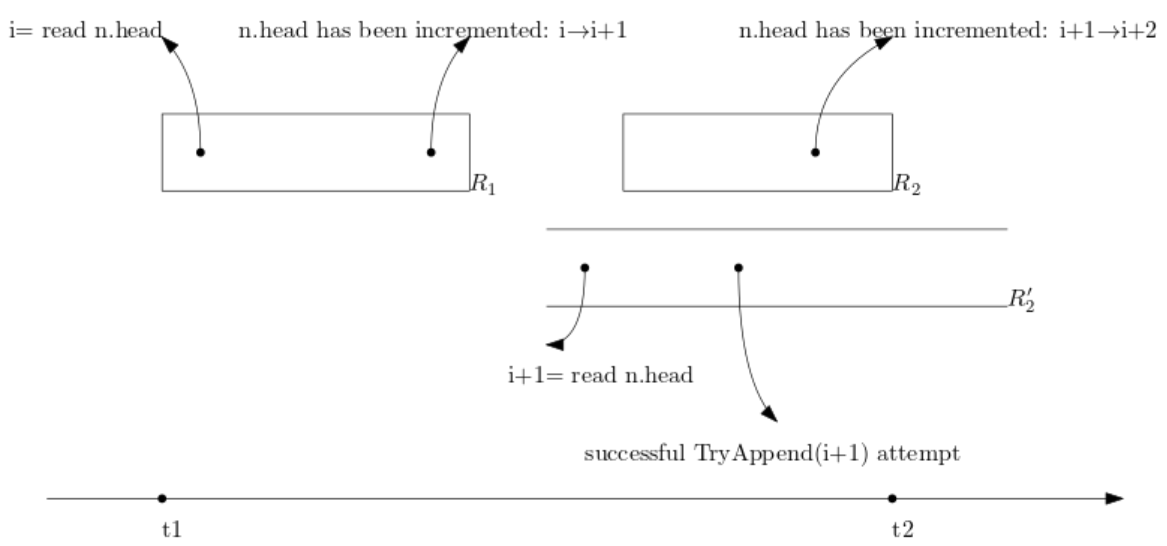


Figure 2: $t1 < r_1$ reading head < incrementing `n.head` from i to $i+1 < R_2'$ reading head < `TryAppend(i+1)` < incrementing `n.head` from $i+1$ to $i+2 < t2$

this chain with more depth should be in the proof

lyRefresh

Corollary 16 (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 15. □

Corollary 17. *After* `Append(blk)` *finishes* `ops(blk)`$\subseteq$`ops(root.blocks[x])` *for some* `x` *and only one* `x`*.*

Proof. Follows from Lemma doubleRefresh 15, 10. □

blockSize **Lemma 18** (Block Size Upper Bound)**.** *Each block contains at most one operation from each processs.*

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 an b' contradicts with append 10. □

ocksBound **Lemma 19** (Subblocks Upperbound)**.** *Each block has at most p direct subblocks.*

Proof. It follows directly from Lemma blockSize 18 and the observation that each block contains at least one operation, induced from Line addOP 312. □

ordering **Definition 20** (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 blockSize 18 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 $\in$ `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 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 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 order enqueues and dequeues among the operations of their own type. Since `GetENQ()` *only searches among enqueues and* `IndexDEQ()` *works on dequeues.*

get **Lemma 21** (Get correctness)**.** *If* `n.blocks[b].num`$_{\texttt{enq}}$`≥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` os 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 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 $E_{n.child,b,i}$. For non-leaf nodes it is decided that the `i`th enqueue in block `b` of internal node `n` is in the `n.blocks[b]`'s subblocks in the left child of `n` or in the `n.blocks[b]`'s subblocks in the right child of `n` (line leftOrRight 404). From Definition ordering 20 we know enqueue operations in a block are ordered by their process id and since in 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 `b.num`$_{\texttt{enq-left}}$ stores the number of `enqueue()` operations from the `n.blocks[b]`'s subblocks in the left child of `n`. So if `i` is greater than `b.num`$_{\texttt{enq-left}}$ it means ith operation is propagated from the right child, otherwise we should search for the ith enqueue in the left child. By definition def::op 6 and def::subblock 4 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`$_{\texttt{left}}$`]` ∪ `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`.

If 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 4. 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 $E_{n,b,i}$.

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 and rightChildGet 405 and 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. □

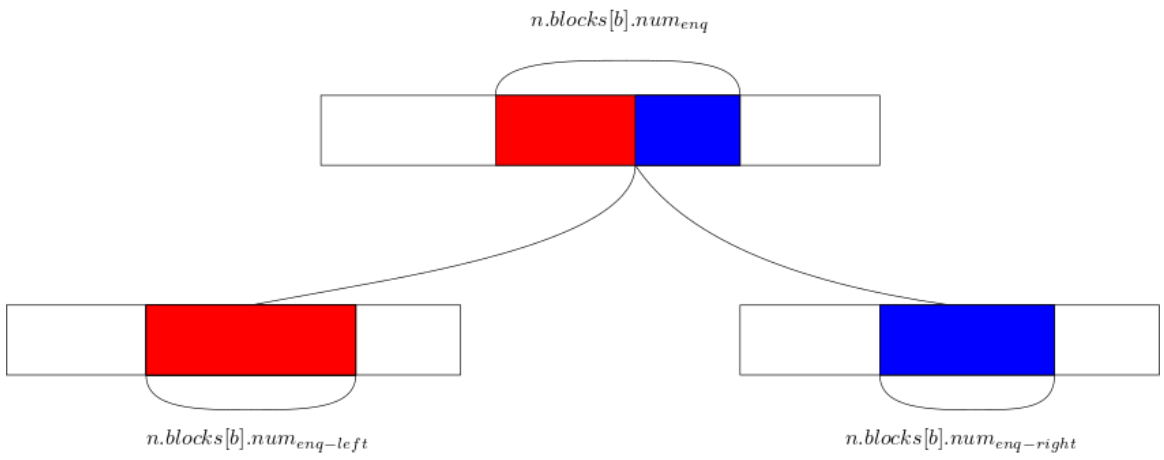


Figure 3: The number and ordering of the enqueues propagated from the left and the right child to `n.blocks[b]`. Enqueue operations from the left subblocks, colored red, are ordered before the enequeue operations from the right child, colored blue.

figGet

dsearch **Lemma 22** (DSearch correctness)**.** *Assume* `root.blocks[end].num`$_{\texttt{enq}}$$\geq$`e` *and* $E_{root,e}$ *is the response to some* `Dequeue()` *in* `root.blocks[end].DSearch(e, end)` *returns* `<b, i>` *such* $E_{root,b,i} = E_{root,e}$ *in* $\Theta\big(\log($`root.blocks[b].size + root.blocks[end].size`$)\big)$ *steps.*

Proof. 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()`. There cannot be more than `root.blocks[b].size Dequeue()`s in `root.blocks[b+1..end-1]`, since the elements of the queue after `blocks[b]` would become dequeued at some point, after `root.blocks[b]` operations finish end before `root.blocks[end]` operations start. And the response to to some `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` in the end and there cannot be more than `root.blocks[b].size Dequeue()`s so there cannot be more than `root.blocks[b].size + root.blocks[end].size enqueue()` operations in `root.blocks[b..end]`. Cause if there were more, then the `root.blocks[end]`'s size would become more than `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` . Cause if otherwise, then `(end-c)/2` satisfied the `root.blocks[(end-c)/2].sum`$_{\texttt{enq}}$`<e`. Note that in the line doubling 804 the distance between `end` and `stat` 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. □

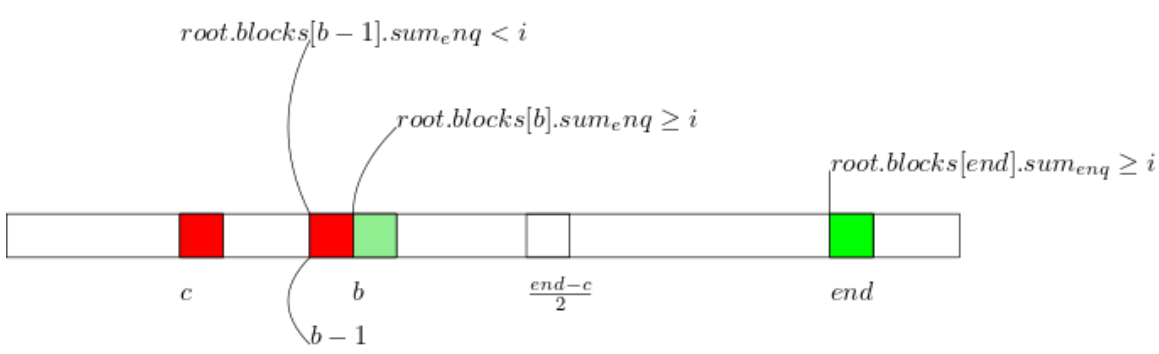


Figure 4: Distance relations between b, c, end

:doubling

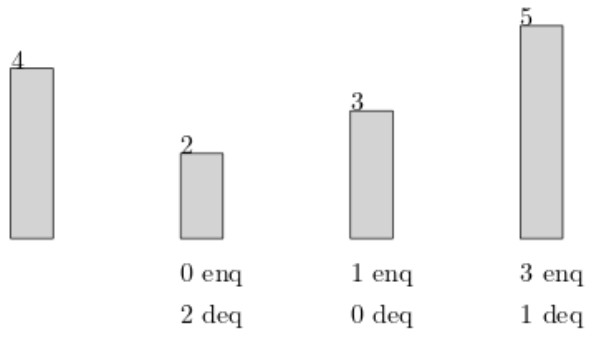


Figure 5: The number written on top of the bars is the queue size. the first block is b and the last block is end.

end-b

Lemma 23 (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 24 (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 20). □

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 24** (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 25. *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 26. `super[]` *preserves order from child to parent; i.e. if in node* `n` *block* `b` *is before* `c` *then* `b.group` $\leq$ `c.group`

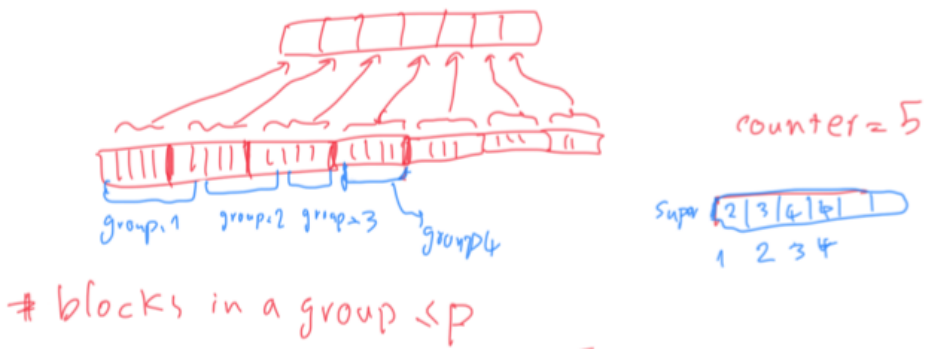
Proof. Line setGroup 329. Since $\texttt{num}_{\texttt{propagated}}$ is increasing. □

Lemma 27. *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 28. *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. □

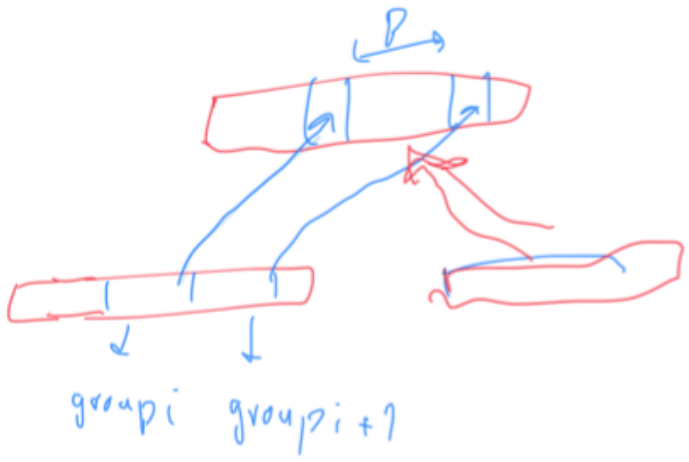


Lemma 29. `super[i+1]-super[i]` $\leq p$

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 30** (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 31.** *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 30. `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 32. *Search in each level of* `IndexDeq()` *takes* $O(\log p)$ *steps.*

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

Definition 33. In an execution on a queue, the dequeue operations that return some value are called *non-null dequeues*.

nseToADeq **Observation 34.** *kth non-null dequeue in an execution returns the* `element` *of kth enqueue.*

rrectness **Lemma 35.** `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. 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 of L in definition ordering 20 enqueue operations come before dequeue operations in a block. If the queue won't become empty after bth block 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). Otherwise `root.blocks[b-1].size+ root.blocks[b].sum`$_{\texttt{enq}}$`- root.blocks[b].sum` would become negative. Since there are more dequeue operations than the size of the queue. See table qhistory 1. □

mberOfNND **Lemma 36.** *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. If there are *enqs* enqueue operations in the prefix for `b`, then the size of the queue after prefix for `b` is `enqs - non-null deqs` by Observation 35. The correctness of the feild size is showed in Lemma sizeCorrectness 35. □

ullReturn **Lemma 37.** $D_{root,b,i}$ *returns* `null` *iff* `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0`.

mputeHead **Lemma 38** (Computing Response)**.** *Let S be the state of an empty queue if the operations in L are applied on it.* `FindResponse(b,i)` *returns the element that is the the response to* $D_{root,b,i}$ *in S. If the queue is empty in S then it returns* `null`.

Proof. First note that by Definition ordering 20 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 20 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 checkEmpty 37 (Line 220). Otherwise if it is d'th non-null dequeue in L it returns d'th enqueue by Observation responseToADeq 34 . By Lemma numberOfNND 36 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). □

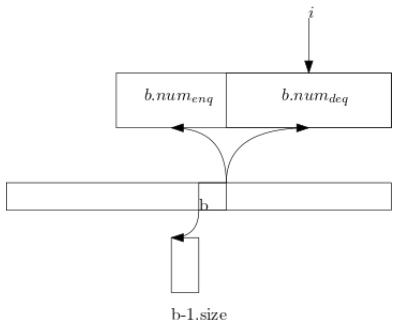


Figure 6: The position of $E_{root,b,i}$.

nseDetail

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

Proof. We choose L in Definition 20 to be linearization ordering of operations and prove if we linearize operations as L the queue works consistently. □

Lemma 40 (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 $\texttt{op}_1\texttt{.Append()}$ is terminated before $\texttt{op}_2\texttt{.Append()}$ starts. From Lemma 10 $\texttt{op}_1$ is in `root.blocks` before $\texttt{op}_2$ propagates so $\texttt{op}_1$ is linearized before $\texttt{op}_2$ by Definition 20.

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 41 (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 10. The time op_1 ends it is in root, before op_2, by Definition 20 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 38 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 21 we know that the returning response is the computed index element.

□

Lemma 42 (Amortized time analysis)**.** `Enqueue()` *and* `Dequeue` *take* $O(\log^2 p + q)$ *steps (amortized anlysis), which* p *is the number of processes and* q *is the size of the queue at the time of invocation.*

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 firstRefresh, secondRefresh 302, 303). Each `Refresh` takes $O(1)$ steps since creating a block is done in constant time and does $O(1)$ `CAS`es. Since the height of the tree is $O(\log p)$, `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 operations, and returns the response. The first two part is similar to an `Enqueue` operation. To compute the order there are some constant steps and `IndexDeq` is called. `IndexDeq` does a search with range p in each level (Lemma superRange 31) 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 dsearch 22, 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 subBlocksBound 19), 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. □