

1 Pseudocode

Algorithm Tree Fields Description

◇ Shared

- *Tree* *tree* : A binary tree of Nodes. *root* is the root node.

◇ Local

- *Node* *leaf* : process's leaf in the tree.

◇ Structures

► Node

- **Node* *left*, *right*, *parent* : initialized when creating the tree.
- *BlockList* *blocks* implemented with an array.
- *int* *head*= 1: #blocks in *blocks*(-1). *blocks*[0] is a block with all fields equal to zero.
- *int* *num_{propagated}*= 0 : # groups of blocks that have been propagated from the node to its parent. Since it is incremented after propagating, it may be behind by 1.
- *int*[] *super*: *super*[*i*] stores an approximate index of the superblock of the blocks in *blocks* whose *group* field have value *i*.

► *Block* ▷ For a block in a blocklist we define *the prefix for the block* to be the blocks in the *BlockList* up to and including the block.
put the definitions before the pseudocode

- *int* *group* : the value read from *num_{propagated}* when appending this block to the node.

► LeafBlock extends Block

- *Object* *element* : Each block in a leaf represents a single operation. If the operation is *enqueue*(*x*) then *element*=*x*, otherwise *element*=*null*.
- *int* *sum_{enq}*, *sum_{deq}* : # enqueue, dequeue operations in the prefix for the block

► InternalBlock extends Block

- *int* *end_{left}*, *end_{right}* : index of the last subblock of the block in the left and right child
- *int* *sum_{enq-left}* : # enqueue operations in the prefix for *left.blocks[end_{left}]*
- *int* *sum_{deq-left}* : # dequeue operations in the prefix for *left.blocks[end_{left}]*
- *int* *sum_{enq-right}* : # enqueue operations in the prefix for *right.blocks[end_{right}]*
- *int* *sum_{deq-right}* : # dequeue operations in the prefix for *right.blocks[end_{right}]*

► RootBlock extends InternalBlock

- *int* *size* : size of the queue after performing all operations in the prefix for this block
-

Abbreviations:

- *blocks*[*b*].*sum_x*=*blocks*[*b*].*sum_{x-left}*+*blocks*[*b*].*sum_{x-right}* (for *b*≥0 and *x* ∈ {*enq*, *deq*})
- *blocks*[*b*].*sum*=*blocks*[*b*].*sum_{enq}*+*blocks*[*b*].*sum_{deq}* (for *b*≥0)
- *blocks*[*b*].*num_x*=*blocks*[*b*].*sum_x*-*blocks*[*b-1*].*sum_x*
(for *b*>0 and *x* ∈ {∅, *enq*, *deq*, *enq-left*, *enq-right*, *deq-left*, *deq-right*}, *blocks*[0].*num_x*=0)

Algorithm Queue

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

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

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

Algorithm	Node
-----------	------

<pre> 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(s) 312: values read from the children's num_{propagated} field. 313: if new.num==0 then return true 314: else if blocks.tryAppend(new, h) then 315: for each dir in {left, right} do 316: CAS(dir.super[np_{dir}], null, h+1) 317: CAS(dir.num_{propagated}, np_{dir}, np_{dir}+1) 318: end for 319: CAS(head, h, h+1) 320: return true 321: else 322: CAS(head, h, h+1) 323: to increase the head. The winner might have fallen sleep before increasing 324: head. 325: end if 326: end REFRESH </pre>	<pre> 327: > Lemma Double Refresh 328: blocks. Returns the created block as well as values read from each child's 329: num_{propagated} field. These values are used for incrementing the children's 330: num_{propagated} field if the block was appended to blocks successfully. 331: block newBlock= NEW(block) 332: newBlock.group= num_{propagated} 333: for each dir in {left, right} do 334: index_{last}= dir.head 335: index_{prev}= blocks[i-1].end_{dir} 336: newBlock.end_{dir}= index_{last} 337: block_{last}= dir.blocks[index_{last}] 338: block_{prev}= dir.blocks[index_{prev}] 339: > newBlock includes dir.blocks[index_{prev}+1..index_{last}]. 340: np_{dir}= dir.num_{propagated} 341: newBlock.sum_{enq-dir}= blocks[i-1].sum_{enq-dir} + block_{last}.sum_{enq} 342: - block_{prev}.sum_{enq} 343: newBlock.sum_{deq-dir}= blocks[i-1].sum_{deq-dir} + block_{last}.sum_{deq} 344: - block_{prev}.sum_{deq} 345: end for 346: if this is root then 347: newBlock.size= max(root.blocks[i-1].size+ newBlock.num_{enq} - 348: newBlock.num_{deq}, 0) 349: end if 350: return <b, np_{left}, np_{right}> 351: end CREATEBLOCK </pre>	<pre> 352: > Creates a block to be inserted into as ith block in 353: blocks. Returns the created block as well as values read from each child's 354: num_{propagated} field. These values are used for incrementing the children's 355: num_{propagated} field if the block was appended to blocks successfully. 356: block newBlock= NEW(block) 357: newBlock.group= num_{propagated} 358: for each dir in {left, right} do 359: index_{last}= dir.head 360: index_{prev}= blocks[i-1].end_{dir} 361: newBlock.end_{dir}= index_{last} 362: block_{last}= dir.blocks[index_{last}] 363: block_{prev}= dir.blocks[index_{prev}] 364: > newBlock includes dir.blocks[index_{prev}+1..index_{last}]. 365: np_{dir}= dir.num_{propagated} 366: newBlock.sum_{enq-dir}= blocks[i-1].sum_{enq-dir} + block_{last}.sum_{enq} 367: - block_{prev}.sum_{enq} 368: newBlock.sum_{deq-dir}= blocks[i-1].sum_{deq-dir} + block_{last}.sum_{deq} 369: - block_{prev}.sum_{deq} 370: end for 371: if this is root then 372: newBlock.size= max(root.blocks[i-1].size+ newBlock.num_{enq} - 373: newBlock.num_{deq}, 0) 374: end if 375: return <b, np_{left}, np_{right}> 376: 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)
                                     ▷ Does binary search for the value
                                     i of the given prefix sum field. Returns the index of the leftmost block in
                                     blocks[start..end] whose field f is  $\geq i$ .
326: end BSEARCH

```

Algorithm Root

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

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

Algorithm Node

\leadsto Precondition: $\text{blocks}[b].\text{num}_{\text{enq}} \geq i$

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

\leadsto Precondition: b th block of the node has propagated up to the root and $\text{blocks}[b].\text{num}_{\text{enq}} \geq i$.

```
413: <int, int> INDEXDEQ(int b, int i)                                     ▷ Returns the rank of  $i$ th dequeue in the  $b$ th block of the node, among the dequeues in the root.
414:     if this is root then
415:         return <b, i>
416:     else
417:         dir= (parent.left==n)? left: right                                     ▷ check if a left or a right child
418:         superBlock= parent.BSEARCH(sumdeq-dir, i, super[blocks[b].group]-p, super[blocks[b].group]+p)
                                                                 ▷ superblock's group has at most  $p$  difference with the value stored in super[].
419:         if dir is right then
420:             i+= blocks[superBlock].sumdeq-left                               ▷ consider the dequeues from the right child
421:         end if
422:         return this.parent.INDEXDEQ(superBlock, i)
423:     end if
424: end INDEX
```

Algorithm Leaf

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

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

Algorithm BlockList

▷ : Supports two operations `blocks.tryAppend(Block b)`, `blocks[i]`. Initially empty, when `blocks.tryAppend(b, n)` returns true `b` is appended to `blocks[n]` and `blocks[i]` returns i th block in the blocks. If some instance of `blocks.tryAppend(b, n)` returns false there is a concurrent instance of `blocks.tryAppend(b', n)` which has returned true. `blocks[0]` contains an empty block with all fields equal to 0 and `endleft`, `endright` pointers to the first block of the corresponding children.

`blocks[]`: array of blocks

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

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

703: **end** TRYAPPEND

2 Proof of Linearizability

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

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

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

Question How to remove the notion of time? To say $\text{pre}(n, i)$ contains $n.\text{blocks}[0..i]$ instead of $\text{EST}(n, t)$ which $\text{head}=i$ 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 $n.\text{blocks}[i]=b$ we call i the *index* of block b . Block b is before block b' in node n if and only if the index of the b is smaller than the index of the b' 's.

Definition 2 (Subblock). Block b is a *direct subblock* of $n.\text{blocks}[i]$ if it is $\in n.\text{left.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1..n.\text{blocks}[i].\text{end}_{\text{left}}] \cup n.\text{right.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}+1..n.\text{blocks}[i].\text{end}_{\text{right}}]$ (See line 533 for the defined range). Block b is a subblock of a $n.\text{blocks}[i]$ if it is a direct subblock of it or subblock of a direct subblock of it.

Definition 3 (Superblock). Block b is direct superblock of block c if c is a direct subblock of b . Block b is superblock of block c if c is a subblock of b .

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

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

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

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

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

Proof. The first claim follows trivially from the pseudocode since $n.\text{head}$ is only incremented. Also when $n.\text{blocks}[i]$ is created its end_{left} , $\text{end}_{\text{right}}$ are greater than or equal to the values in $n.\text{blocks}[i-1]$. Since $\text{blocks}[i-1].\text{end}_{\text{dir}} < \text{dir.head} = \text{blocks}[i].\text{end}_{\text{dir}}$ (Lines 177, 178). □

Lemma 8. Every block has most one direct superblock.

Proof. To show this we are going to refer to the way $n.\text{blocks}[]$ is partitioned while propagating blocks up to $n.\text{parent}$. $n.\text{CreateBlock}(i)$ merges the blocks in $n.\text{left.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}..n.\text{blocks}[i].\text{end}_{\text{left}}]$ and $n.\text{right.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}..n.\text{blocks}[i].\text{end}_{\text{right}}]$ (Lines 177, 178). Since $\text{end}_{\text{left}}, \text{end}_{\text{right}}$ are non-decreasing, so the range of the subblocks of $n.\text{blocks}[i]$ which is $(n.\text{blocks}[i-1].\text{end}_{\text{dir}}+1..n.\text{blocks}[i].\text{end}_{\text{dir}})$ does not overlap with the range of the subblocks of $n.\text{blocks}[i-1]$. □

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

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

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

Some value is written into `n.blocks[head]` only in Line 313. It is obvious that writing into `n.blocks[head]` preserves the invariant. The value of `n.head` is modified only in lines ~~318~~^{incrementHead1}, ~~321~~^{readHead}. Depending on whether the `TryAppend()` in Line ~~313~~^{cas} succeeded or not we show that the claim holds after the increment lines of `n.head` in either case. If `head` is incremented to h it is sufficient to show `n.blocks[h] ≠ null` to prove the invariant still holds. In the first case the process applied a successful `TryAppend(new, h)` in line ~~314~~^{lockcas}, which means `n.blocks[h]` is not null anymore. Note that whether ~~318~~^{incrementHead1} returns true or false after Line `n.head` we know has been incremented from Line ~~310~~^{readHead}. The failure case is also the same since it means some value is written into `n.blocks[head]` by some process. □

Lemma 12 (head Increment).

If an $n.Refresh$ instance reaches Line cas 313 and reads $head=h$ (Line readHead 310) after it terminates $head$ is greater than h .

Proof. If Line incrementHead1 318 or incrementHead2 318 succeeded the claim holds, otherwise another process has incremented the head. \square

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 $ops(EST_{n.left, t_i}) \cup ops(EST_{n.right, t_i}) \subseteq ops(EST_n, t_t)$.

Proof. From Lemma lem::establishedOrder II we know that $ops(EST_n, t_i) \subseteq ops(EST_n, t_t)$. So it remains to show the operations of $ops(EST_{n.left, t_i}) \cup ops(EST_{n.right, t_i}) - ops(EST_n, t_i)$, which we call *new operations*, are all in $ops(EST_n, t_t)$. If $TryAppend$ returns **true** a block **new** is written into $n.blocks[h]$ (Line cas 313). We show $ops(EST_{n.left, t_i}) \subseteq ops(EST_n, t_t)$. The proof for the right child's claim is the same. Let $n.left.head$ at t_i be hli . Let $n.Refresh()$ read $head$ equal to h (Line readHead 310). By the lines prevLine 332, 331 the new block in $n.blocks[h]$ contains $n.left.blocks[n.blocks[h-1].end_{left}+1..left.head]$. Since $left.head$ is read after t_i then $ops(EST_{n.left, t_i}) \subseteq ops(n.left.blocks[0..left.head])$. By Lemma lem::establishedOrder III $ops(n.left.blocks[0..n.blocks[h-1].end_{left}]) \subseteq ops(EST_n, t_i) \subseteq ops(EST_n, t_t)$. Since after line incrementHead2 321 we are sure that the $head$ is incremented (Lemma lem::headInc 312) 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. \square

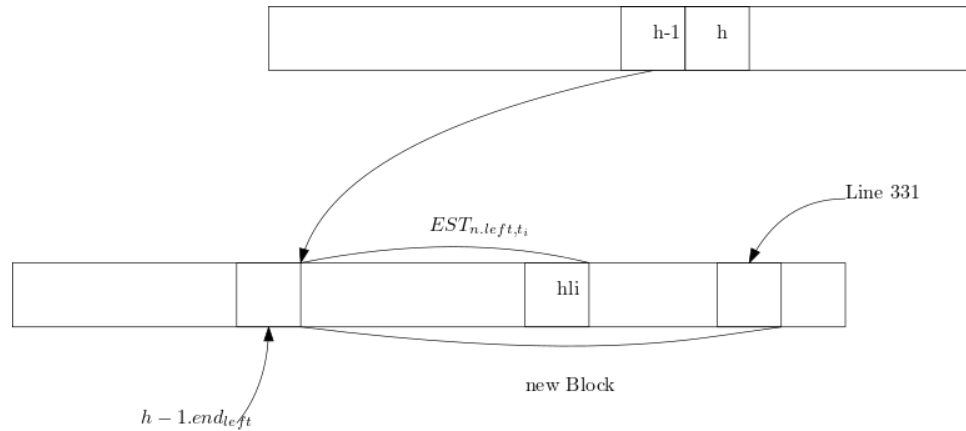


Figure 1: New established operations of the left child are in the new block.

Lemma 14 (Precise 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 $ops(EST_{n.left, t_i}) \cup ops(EST_{n.right, t_i}) \subseteq ops(n.blocks)$.

Lemma 15 (Double Refresh). *Consider two consecutive failed instances R_1, R_2 of $\text{n.Refresh}()$ by some process. Let t_1 be the time R_1 is invoked and t_2 be the time R_2 terminated. We have $\text{ops}(\text{EST}_{\text{n.left}}, t_1) \cup \text{ops}(\text{EST}_{\text{n.right}}, t_1) \subseteq \text{ops}(\text{EST}_{\text{n}}, t_2)$.*

Proof.

If Line 313 of R_1 or R_2 returns **true**, then the claim is held by Lemma 13. Let R_1 read i and R_2 read $i+1$ from Line 310. If R_2 reads some value greater than $i+1$ in Line 310 it means a successful instance of $\text{Refresh}()$ started after Line 310 of R_1 and finished its Line 318 or 321 before 310 of R_2 , from Lemma 13 by the end of this instance $\text{ops}(\text{EST}_{\text{n.left}}, t_1) \cup \text{ops}(\text{EST}_{\text{n.right}}, t_1)$ has been propagated.

Since R_2 's $\text{TryAppend}()$ returns **false** then there is another successful instance R'_2 of $\text{n.Refresh}()$ that has done $\text{TryAppend}()$ successfully into $\text{n.blocks}[i+1]$ 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 14 after R'_2 's CAS we have $\text{ops}(\text{EST}_{\text{n.left}}, t_1) \cup \text{ops}(\text{EST}_{\text{n.right}}, t_1) \subseteq \text{ops}(\text{n.blocks})$. Also by Lemma 12 of R'_2 head is more than $i+1$ after R'_2 's 321 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 $\text{ops}(\text{EST}_{\text{n.left}}, t_1) \cup \text{ops}(\text{EST}_{\text{n.right}}, t_1) \subseteq \text{ops}(\text{EST}_{\text{n}}, t_2)$. \square

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

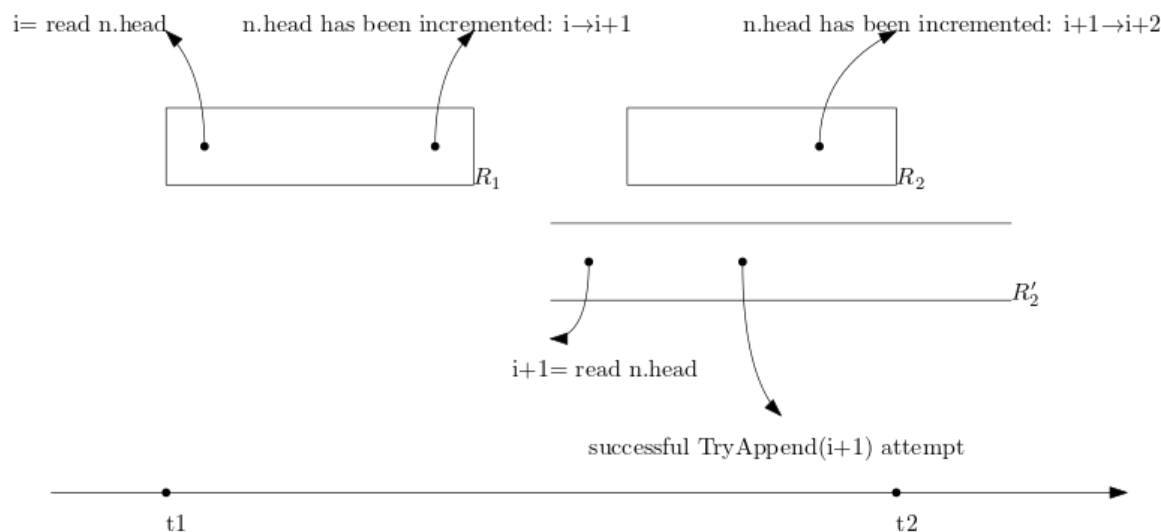


Figure 2: $t_1 < r_1$ reading head $<$ incrementing n.head from i to $i+1 < R'_2$ reading head $< \text{TryAppend}(i+1) <$ incrementing n.head from $i+1$ to $i+2 < t_2$

this chain with more depth should be in the proof

lyRefresh

Corollary 16 (Propagate Step). *All operations in \mathbf{n} 's children's established blocks before line 302 are guaranteed to be in \mathbf{n} 's established blocks after line 303.*

Proof. Lines 302 and 303 satisfy the preconditions of Lemma 15. □

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

Proof. Follows from Lemma 15, 9. □

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

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

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

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

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 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 $\in \text{ops}(\mathbf{n.blocks}[b])$ ordered by their process id.
- $E(n, b, i)$ is the i th enqueue in $E(n, b)$.
- $D(n, b)$ is the sequence of dequeue operations $\in \text{ops}(\mathbf{n.blocks}[b])$ ordered by their process id.
- $D(n, b, i)$ is the i th dequeue in $D(n, b)$.
- Order of the enqueue operations in n : $E(n) = E(n, 1).E(n, 2).E(n, 3)\dots$
- $E_{n,i}$ is the i th enqueue in $E(n)$.
- Order of the dequeue operations in n : $D(n) = D(n, 1).D(n, 2).D(n, 3)\dots$
- $D_{n,i}$ is the i th dequeue in $D(n)$.
- Linearization: $L = E(\text{root}, 1).D(\text{root}, 1).E(\text{root}, 2).D(\text{root}, 2).E(\text{root}, 3).D(\text{root}, 3)\dots$

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.

Preconditions of all invocation of **BSearch** are satisfied.

Lemma 21 (Get correctness). *If $n.\text{blocks}[b].\text{num}_{\text{enq}} \geq i$ then $n.\text{GetENQ}(b, i)$ returns $E(n, b, i)$.*

Proof. We are going to prove this lemma by induction on the height of the tree. The base case for the leaves of the tree is pretty straight forward. Since leaf blocks contain exactly one operation then only $\text{GetENQ}(b, 1)$ can be called on leaves. $\text{leaf}.\text{GetENQ}(b, 1)$ returns the operation stored in the b th block of leaf l . For non leaf nodes in Line 404 it is decided that the i th enqueue in block b of internal node n resides in the left child or the right child of n . From Definition ordering 20 we know operations in a block are ordered by their process id. Furthermore $b.\text{sum}_{\text{enq-left}}$ stores the number of `enqueue()` operations from the b 's subblocks of the left child of n . So if i is greater than $b.\text{sum}_{\text{enq-left}}$ it means i th operation is propagated from the right child, otherwise we should search for the i th enqueue in the left child subblocks. By definition def::op::subblock 4 and 2 we need to search in subblocks of b which their range is $n.\text{left}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1..n.\text{blocks}[i].\text{end}_{\text{left}}] \cup n.\text{right}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}+1..n.\text{blocks}[i].\text{end}_{\text{right}}]$. If the enqueue we're looking for was in the right child as there are $b.\text{sum}_{\text{enq-left}}$ enqueues before it we need to search for $i-b.\text{sum}_{\text{enq-left}}$ (Line rightChildGet 409). By definition of $E(n, b)$ operations from the left child come before the operations of the right child. Having sum_{enq} , the prefix sum of the number of enqueues we can compute the direct subblock containing the enqueue we are finding for with binary search. Then $n.\text{child}.\text{GetENQ}(\text{block containing, order in the block})$ is invoked which returns the correct operation by the hypothesis of the induction. \square

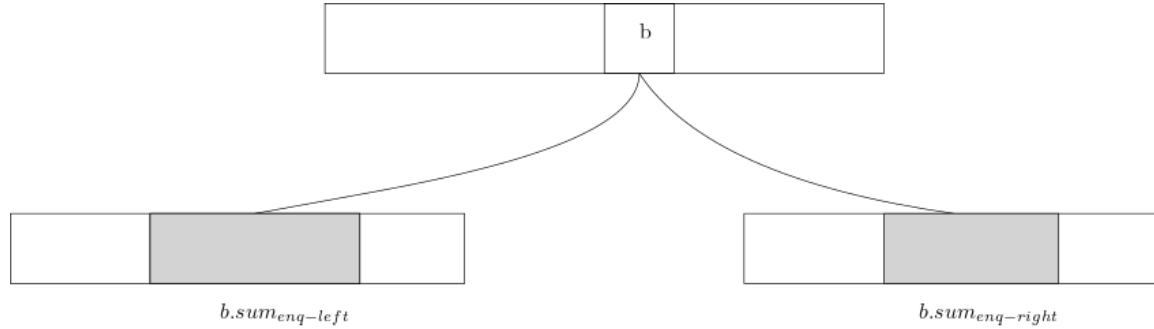


Figure 3: The number of enqueues from the left and the right child

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

dsearch

Lemma 22 (DSearch correctness). *If $\text{root.blocks}[\text{end}].\text{num}_{\text{enq}} \geq i$ and $E(\text{root}, i)$ is the response to some $\text{Dequeue}()$ in $\text{root.blocks}[\text{end}]$ then $\text{DSearch}(i, \text{end})$ returns b such that $\text{root.blocks}[b]$ contains $E(\text{root}, b, i)$ in $\Theta(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$ steps.*

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

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

Preconditions of all invocation of **BSearch** are satisfied.

Lemma 23 (Computing SuperBlock). *After computing line ^{computeSuper}418 of `n.IndexDEQ(b,i)`, `n.parent.blocks[superblock]` contains $D(n,b,i)$.*

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

► Values `npdir` read in lines ^{setNP}337, `super` are set before incrementing in lines ^{setSuperNP}315,316. So before incrementing `numpropagated`, `super[numpropagated]` is set so it cannot be null while reading.

2. `super[]` preserves order from child to parent; i.e. if in node `n` block `b` is before `c` then `b.group ≤ c.group` ► Line ^{setGroup}329. Since `numpropagated` is increasing.

3. Let `b`, `c` be in node `n`, if `b.group ≤ c.group` then `super[b.group] ≤ super[c.group]`

► Line ^{setSuper}315.

4. `super[i+1]-super[i] ≤ p`

► In a Refresh with successful CAS in line 46, `super` and `counter` are set for each child in lines 48,49. Assume the current value of the counter in node `n` is `i+1` and still `super[i+1]` is not set. If an instance of successful `Refresh(n)` finishes `super[i+1]` is set a new value and a block is added after `n.parent[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 .

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

► `super[i]` is the index of the superblock of a block containing block `b`, followed by Lemma ^{superCounter}26. It is trivial to see that `n.super` and `n.b.counter` are increasing. `super(b)` is the real superblock of `b`. `super(t)` is the index of the superblock of the last block with time `t`. If `b.time` is `t` we have:

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

□

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

Lemma 24 (Index correctness). $n.\text{IndexDEQ}(b, i)$ returns the rank in $D(\text{root})$ of $D(n, b, i)$.

Proof. We can see the base case $\text{root}.\text{IndexDEQ}(b, i)$ is trivial. $n.\text{IndexDEQ}(b, i)$ computes the superblock of the i th Dequeue in the b th block of n in $n.\text{parent}(\text{Lemma } \text{superBlock } 23)$. Then the order in $D(n.\text{parent}, \text{superblock})$ is computed and $\text{index}()$ is called on $n.\text{parent}$ recursively. It is easy to see why the second is correct. Correctness of computing superblock comes from Lemma $\text{superBlock } 23.\text{size}$ \square

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

computeHead

Lemma 25 (Computing Queue's Head block). Let S be the state of an empty queue if the operations in prefix in L of i th dequeue in $D(\text{root}, b)$ are applied on it. $\text{FindResponse}()$ returns (b, i) which $E(\text{root}, b, i)$ is the the head of the queue in S . If the queue is empty in S then it returns $\langle -1, -- \rangle$.

Proof. The sizeof the queue if the operations in the prefix for the b th block in the root are applied with the order of L is stored in the $\text{root.blocks}[b].\text{size}$. It is computed while creating the block in Line $\text{computeLength } 542$. If the sizeof a queue is greater than 0 then a $\text{Dequeue}()$ would decrease the sizeof the queue, otherwise the sizeof the queue remains 0. Having sizeof the queue after the previous block and number of enqueues and dequeues in the block, Line $\text{computeLength } 542$ computes whether the queue becomes empty or the sizeof it. \square

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

Lemma 26 (Validity of super and counter). *If $\text{super}[i] \neq \text{null}$, then $\text{super}[i]$ in node \mathbf{n} is the index of the superblock of a block with $\text{time} = i \pm p$.*

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

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

Lemma 28. *Operations in a block have a time point in common (There is a time t all the operations are running).*

Lemma 29 (satisfiability). *L can be a linearization ordering.*

Proof. 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 (previous lemma). Furthermore in L we arbitrary choose the order to be by process id, since it makes computations in the blocks faster. □

Lemma 30 (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}9. The time op_1 ends it is in root, before op_2 , by Definition ^{ordering}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 ^{computeHead}25 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}21 we know that the returning response is the computed index element.

□

Lemma 31 (Time analysis). `n.GetEnq(b,i)`, `n.Index(b,i)` take $O(\log^2 p)$ steps. Search in the root may take $O(\log Q + p^2)$ steps. Helping is done every p^2 block appended to the root and takes $p \times \log^2 p$ steps. Amortized time consumed for helping by each process is $O(\log^2 p)$.

rootRange

Lemma 32 (Root search range). `root.size-root.FindMostRecentDone()` is $O(p^2 + q)$, which p is # processes and q is the size of the queue.