

# 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* num<sub>propagated</sub>= 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 num<sub>propagated</sub> 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<sub>enq</sub>, sum<sub>deq</sub> : # enqueue, dequeue operations in the prefix for the block

#### ► InternalBlock extends Block

- *int* end<sub>left</sub>, end<sub>right</sub> : indices of the last subblock of the block in the left and right child
- *int* sum<sub>enq-left</sub> : # enqueue operations in the prefix for left.blocks[end<sub>left</sub>]
- *int* sum<sub>deq-left</sub> : # dequeue operations in the prefix for left.blocks[end<sub>left</sub>]
- *int* sum<sub>enq-right</sub> : # enqueue operations in the prefix for right.blocks[end<sub>right</sub>]
- *int* sum<sub>deq-right</sub> : # dequeue operations in the prefix for right.blocks[end<sub>right</sub>]

#### ► RootBlock extends InternalBlock

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

### Abbreviations:

- blocks[b].sum<sub>x</sub>=blocks[b].sum<sub>x-left</sub>+blocks[b].sum<sub>x-right</sub> (for b≥0 and x ∈ {enq, deq})
- blocks[b].sum=blocks[b].sum<sub>enq</sub>+blocks[b].sum<sub>deq</sub> (for b≥0)
- blocks[b].num<sub>x</sub>=blocks[b].sum<sub>x</sub>-blocks[b-1].sum<sub>x</sub>  
(for b>0 and x ∈ {∅, 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.sumenq= leaf.blocks[leaf.head].sumenq+1
205:   newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq
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.sumenq= leaf.blocks[leaf.head].sumenq
212:   newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq+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)
219:   if root.blocks[b-1].size + root.blocks[b].numenq - i < 0 then
220:     return null
221:   else
222:     e= i - root.blocks[b-1].size + root.blocks[b-1].sumenq
223:     return root.GetENQ(root.DSEARCH(e, b))
224:   end if
225: end FINDRESPONSE
```

deqRest

checkEmpty

computeE

findAnswer

---

## Algorithm Node

---

```

301: void PROPAGATE()
302:   if not REFRESH() then
303:     REFRESH()
304:   end if
305:   if this is not root then
306:     parent.PROPAGATE()
307:   end if
308: end PROPAGATE

309: boolean REFRESH()
310:   h= head
311:   <new, npleft, npright>= CREATEBLOCK(h)  ▷ npleft, npright are the
values read from the children's numpropagated 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[npdir], null, h)  ▷ Write would work too.
316:       CAS(dir.numpropagated, npdir, npdir+1)
317:     end for
318:     CAS(head, h, h+1)
319:     return true
320:   else
321:     CAS(head, h, h+1)  ▷ 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

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

```

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

---

## Algorithm Root

---

```

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].sumenq ≥ e do
804:     start= max(start-(end-start), 0)
805:   end while
806:   b= root.BSearch(sumenq, e, start, end)
807:   i= e- root.blocks[b-1].sumenq
808:   return <b,i>
809: end DSEARCH

```

Algorithm Node	
	<pre> <math>\rightsquigarrow</math> Precondition: <code>blocks[b].num<sub>enq</sub> ≥ i ≥ 1</code> </pre>
	<pre> 401: <i>element</i> GETENQ(<i>int</i> b, <i>int</i> i) <span style="float:right">▷ Returns the element of <math>E_{this,b,i}</math>.</span> 402:   <b>if</b> this is leaf <b>then</b> 403:     <b>return</b> blocks[b].element 404:   <b>else if</b> <math>i \leq \text{blocks}[b].\text{num}_{\text{enq-left}}</math> <b>then</b> <span style="float:right">▷ <math>E_{this,b,i}</math> is in the left child of this node.</span> 405:     subBlock= left.BSEARCH(sum<sub>enq</sub>, i+blocks[b-1].sum<sub>enq-left</sub>, blocks[b-1].end<sub>left</sub>+1, blocks[b].end<sub>left</sub>) 406:     <b>return</b> left.GETENQ(subBlock, i) 407:   <b>else</b> 408:     i= i-blocks[b].num<sub>enq-left</sub> 409:     subBlock= right.BSEARCH(sum<sub>enq</sub>, i+right.blocks[b-1].sum<sub>enq-right</sub>, blocks[b-1].end<sub>right</sub>+1, blocks[b].end<sub>right</sub>) 410:     <b>return</b> right.GETENQ(subBlock, i) 411:   <b>end if</b> 412: <b>end</b> GETENQ </pre>
tBaseCase	
ftOrRight	
tChildGet	
tChildGet	
	<pre> <math>\rightsquigarrow</math> Precondition: bth block of the node has propagated up to the root and <code>blocks[b].num<sub>enq</sub> ≥ i</code>. </pre>
	<pre> 413: &lt;<i>int</i>, <i>int</i>&gt; INDEXDEQ(<i>int</i> b, <i>int</i> i) <span style="float:right">▷ Returns &lt;x, y&gt; if <math>D_{this,b,i} = D_{root,x,y}</math>.</span> 414:   <b>if</b> this is root <b>then</b> 415:     <b>return</b> &lt;b, i&gt; 416:   <b>else</b> 417:     dir= (parent.left==n)? left: right <span style="float:right">▷ check if this node is a left or a right child</span> 418:     superBlock= parent.BSEARCH(sum<sub>deq-dir</sub>, i+blocks[b-1].sum<sub>deq</sub>, super[blocks[b].group]-p, super[blocks[b].group]+p) <span style="float:right">▷ superblock's group has at most <math>p</math> difference with the value stored in <code>super[]</code>.</span> 419:     <b>if</b> dir is right <b>then</b> 420:       i+= blocks[superBlock].num<sub>deq-left</sub> <span style="float:right">▷ consider the dequeues from the right child</span> 421:     <b>end if</b> 422:     <b>return</b> this.parent.INDEXDEQ(superBlock, i) 423:   <b>end if</b> 424: <b>end</b> INDEXDEQ </pre>
xBaseCase	
puteSuper	
iderRight	
Algorithm Leaf	
	<pre> 601: <i>void</i> APPEND(<i>block</i> blk) <span style="float:right">▷ Append is only called by the owner of the leaf.</span> 602:   blk.group= head 603:   blocks[head]= blk 604:   head+=1 605:   parent.PROPAGATE() 606: <b>end</b> APPEND </pre>
pendStart	
appendEnd	
Algorithm BlockList	
	<p>▷ : Supports two operations <code>blocks.tryAppend(Block b)</code>, <code>blocks[i]</code>. Initially empty, when <code>blocks.tryAppend(b, n)</code> returns true b is appended to <code>blocks[n]</code> and <code>blocks[i]</code> returns <math>i</math>th block in the blocks. If some instance of <code>blocks.tryAppend(b, n)</code> returns false there is a concurrent instance of <code>blocks.tryAppend(b', n)</code> which has returned true.blocks[0] contains an empty block with all fields equal to 0 and <code>end<sub>left</sub></code>, <code>end<sub>right</sub></code> pointers to the first block of the corresponding children.</p> <p><i>block[]</i> blocks: array of blocks</p> <p><i>int[]</i> super: super[i] stores an approximate index of the superblock of the blocks in blocks whose group field have value i.</p> <pre> 701: <i>boolean</i> TRYAPPEND(<i>block</i> blk, <i>int</i> n) 702:   <b>return</b> CAS(blocks[n], null, blk) 703: <b>end</b> TRYAPPEND </pre>

## 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. A block in a node's blocklist 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. 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.

**Lemma 2** (head Increment). *Let  $R$  be an instance of Refresh on node  $n$  that reaches Line 313. After  $R$  terminates  $n.\text{head}$  is greater than  $h$ , the value read in line 310 of  $R$ .*

*Proof.* If Line 318 or 321 are successful then the claim holds, otherwise another process has incremented the head from  $h$  to  $h+1$ .  $\square$

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

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

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

*Explain More*

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

*Proof.* The first claim follows trivially from the pseudocode since  $n.\text{head}$  is only incremented in the pseudocode in lines 318 and 321 of `Refresh()`.

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

**Definition 5** (Subblock). Block  $b$  is a *direct subblock* of  $n.\text{blocks}[i]$  if it is in  $n.\text{left.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1..n.\text{blocks}[i].\text{end}_{\text{left}}] \cup n.\text{right.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}+1..n.\text{blocks}[i].\text{end}_{\text{right}}]$ . Block  $b$  is a subblock of  $n.\text{blocks}[i]$  if  $b$  is a direct subblock of  $n.\text{blocks}[i]$  or a subblock of a direct subblock of  $n.\text{blocks}[i]$ .

**Definition 6** (Superblock). Block  $b$  is *direct superbloc* 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 7** (Operations of a block). A leaf block  $b$  in a leaf represents `enqueue(x)` if  $b.element \neq \text{null}$ . Else if  $b.element = \text{null}$   $b$  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  $n.head > \text{index of } b \text{ at time } t$ .  $EST_{n, t}$  is the set of established blocks of node  $n$  at time  $t$ .

**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 superbloc.

*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  $n.left.blocks[n.blocks[i-1].end_{left}..n.blocks[i].end_{left}]$  and  $n.right.blocks[n.blocks[i-1].end_{right}..n.blocks[i].end_{right}]$ . (Lines lastLine, pr). Since  $end_{left}, end_{right}$  are non-decreasing ( $n.blocks[i].end_{left|right} > n.blocks[i-1].end_{left|right}$ ), so the range of the subblocks of  $n.blocks[i]$  which is  $(n.blocks[i-1].end_{dir}+1..n.blocks[i].end_{dir})$  does not overlap with the range of the subblocks of  $n.blocks[i-1]$ .  $\square$

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

**Lemma 12** (establishedOrder). If time  $t < \text{time } t'$ , then  $ops(EST_{n, t}) \subseteq ops(EST_{n, t'})$ .

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

*useless?*

ueRefresh

**Lemma 13** (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  $ops(EST_{n.left, t_i}) \cup ops(EST_{n.right, t_i}) \subseteq ops(EST_n, t_t)$ .*

*Proof.* Since  $TryAppend$  returns true a block **new** is written into  $n.blocks[h]$  in Line <sup>cas</sup>313.

We show  $ops(EST_{n.left, t_i}) \subseteq ops(EST_n, t_t)$ . Let  $h$  be the value  $n.Refresh()$  reads from  $n.head$  at line <sup>readHead</sup>310,  $h_{left,i}$  be the value of  $n.left.head$  at  $t_i$  and  $h_{left,read}$  be the value read from  $n.left.head-1$  at line <sup>lastLine</sup>331.  $end_{left}$  field of the block returned by  $CreateBlock(i)$  is  $h_{left,read}$ . By lines <sup>prevLine</sup>332 and <sup>lastLine</sup>331 the new block in  $n.blocks[h]$  contains  $n.left.blocks[n.blocks[h-1].end_{left}+1..h_{left,read}]$ . Since  $left.head$  is read after  $t_i$  then  $h_{left,read} > h_{left,i}$  which means  $ops(EST_{n.left, t_i}) \subseteq ops(n.left.blocks[0..h_{left,read}])$ . After the successful  $TryAppend$  in line <sup>cas</sup>313 we know all blocks in  $n.left.blocks[0..h_{left,read}-1]$  are subblocks of  $n.blocks[0..h]$  by the definition of subblock. At  $t_t$  we have  $n.head > h$  by Lemma <sup>lem:headProgress</sup>4. So  $n.blocks[1..h]$  are in  $EST_{n,t_t}$  by definition of  $EST$ . Note that after line <sup>incrementHead2</sup>321 we are sure that the head is incremented by Lemma <sup>lem:headInc</sup>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  $ops(EST_{n.right, t_i}) \subseteq ops(EST_n, t_t)$  is the same. □

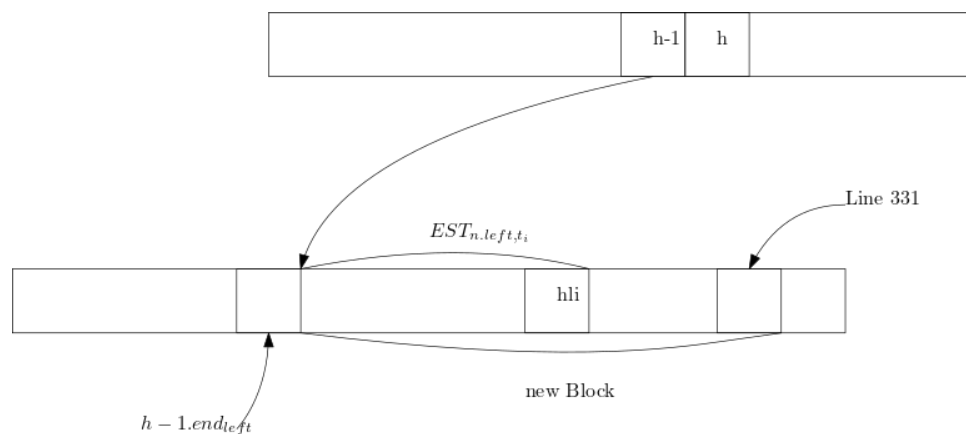


Figure 1: New established operations of the left child are in the new block.

ueRefresh

**Lemma 14** (Stronger True Refresh). *Let  $t_i$  be the time an instance of  $n.Refresh()$  read the head (Line <sup>readHead</sup>310) and  $t_t$  be the time its  $TryAppend(new, s)$  terminates with and returns **true** (Line <sup>cas</sup>313). We have  $ops(EST_{n.left, t_i}) \cup ops(EST_{n.right, t_i}) \subseteq ops(n.blocks)$ .*

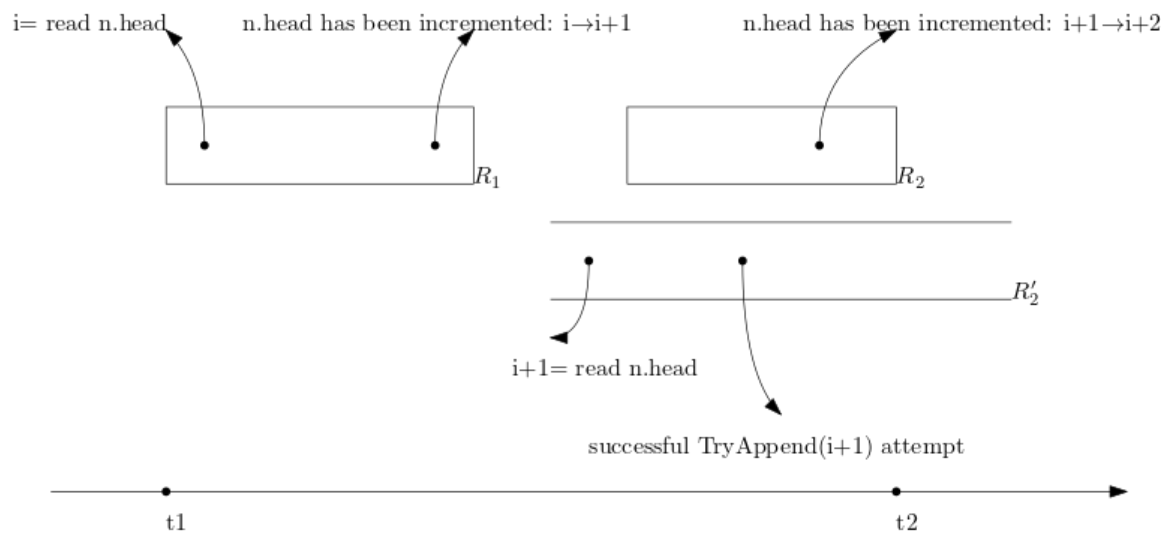
leRefresh

**Lemma 15** (Double Refresh). *Consider two consecutive instances  $R_1, R_2$  of  $\text{Refresh}()$  on  $\mathbf{n}$  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  $\text{ops}(\text{EST}_{\mathbf{n}, \text{left}}, t_1) \cup \text{ops}(\text{EST}_{\mathbf{n}, \text{right}}, t_1) \subseteq \text{ops}(\text{EST}_{\mathbf{n}}, t_2)$ .*

*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  $\text{ops}(\text{EST}_{\mathbf{n}, \text{left}}, t_1) \cup \text{ops}(\text{EST}_{\mathbf{n}, \text{right}}, t_1)$  has been propagated.

Since  $R_2$ 's  $\text{TryAppend}()$  returns **false** then there is another successful instance  $R'_2$  of  $\mathbf{n}.\text{Refresh}()$  that has done  $\text{TryAppend}()$  successfully into  $\mathbf{n}.\text{blocks}[i+1]$  before  $R_2$  tries to append. Since  $R'_2$  creates the block after reading the value  $i+1$  from  $\mathbf{n}.\text{head}$  (Line 310) and  $R_1$  reads the value  $i$  from  $\mathbf{n}.\text{head}$  and the **head**'s value is increasing by Lemma 4 then  $t_{R'_2, 310} > t_{R_1, 310} > t_1$  (See Figure 2). By Lemma 14 after  $R'_2$ 's CAS we have  $\text{ops}(\text{EST}_{\mathbf{n}, \text{left}}, t_1) \cup \text{ops}(\text{EST}_{\mathbf{n}, \text{right}}, t_1) \subseteq \text{ops}(\mathbf{n}.\text{blocks})$ . Also by Lemma 2 on  $R'_2$  the value of  $\mathbf{n}.\text{head}$  head is more than  $i+1$  after  $R'_2$  terminates, so the block appended by  $R'_2$  to  $\mathbf{n}$  is established by then ( $\mathbf{n}.\text{head} \geq i+2 > i+1$ ). To summarize  $t_1$  is before  $R'_2$ 's read  $\mathbf{n}.\text{head}$  and  $R'_2$ 's successful CAS is before  $R_2$ 's termination. So by Lemma 14  $\text{ops}(\text{EST}_{\mathbf{n}, \text{left}}, t_1) \cup \text{ops}(\text{EST}_{\mathbf{n}, \text{right}}, t_1) \subseteq \text{ops}(\text{EST}_{\mathbf{n}}, t_2)$ .  $\square$



leRefresh

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

*this chain with more depth should be in the proof*

**Definition 16.**  $t_{\text{before line}}$  is the immediate time before running Line *line*.  $t_{\text{after line}}$  is the immediate time after running Line *line*.

**Corollary 17.**  $\text{ops}(\text{EST}_{\mathbf{n}, \text{left}}, t_{\text{before 302}}) \cup \text{ops}(\text{EST}_{\mathbf{n}, \text{right}}, t_{\text{before 302}}) \subseteq \text{ops}(\text{EST}_{\mathbf{n}}, t_{\text{after 303}})$

*Proof.* If the first  $\text{Refresh}()$  in line 302 returns **true** then by Lemma 13 the claim holds. Also if first  $\text{Refresh}()$  failed and the second  $\text{Refresh}()$  succeeded the claim still holds by Lemma 13. Finally, if both failed the claim is satisfied by Lemma 15.  $\square$



lyRefresh

**Corollary 18** (Propagate Step). *All operations in  $\mathbf{n}$ 's children's established blocks before running line 302 of a **Propagate** routine are guaranteed to be in  $\mathbf{n}$ 's established blocks after line 303.*

*Proof.* If 302 or 303 succeed, the claim is true by Lemma 13. Otherwise Lines 302 and 303 satisfy the preconditions of Lemma 15.  $\square$

actlyOnce

**Corollary 19.** *After **Append**(blk) finishes  $\text{ops}(\text{blk}) \subseteq \text{ops}(\text{root.blocks}[\mathbf{x}])$  for exactly one  $\mathbf{x}$ .*

*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 18 inductively it is propagated up to the root. Finally Lemma 11 shows only one block in the root contains blk.  $\square$

**blockSize** **Lemma 20** (Block Size Upper Bound). *Each block contains at most one operation of each process.*

*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 lem::appendExactlyOnce 19 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  $b$  and  $b'$  contradicts Lemma append 11.  $\square$

**blocksBound** **Lemma 21** (Subblocks Upperbound). *Each block has at most  $p$  direct subblocks.*

*Proof.* It follows directly from Lemma blockSize 20 and the observation that each block appended to the tree contains at least one operation, induced from Line addOP 312.  $\square$

**Definition 22** (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 20 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.*

**Lemma 23** (Get correctness). *If  $n.blocks[b].num_{enq} \geq i$  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 operation stored in the  $b$ th 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 22 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  $b.num_{enq-left}$  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  $b.num_{enq-left}$  it means  $i$ th operation is propagated from the right child, otherwise we should search for the  $i$ th enqueue in the left child. By definition def::opdef::subblock 7 and  $b$  we need to search in subblocks of  $n.blocks[b]$  from the range  $n.left.blocks[n.blocks[i-1].end_{left}+1..n.blocks[i].end_{left}] \cup n.right.blocks[n.blocks[i-1].end_{right}+1..n.blocks[i].end_{right}]$ .

If the  $i$ th enqueue of  $n.blocks[b]$  is in the left child it would be  $i$ th enqueue in  $n.left.blocks[n.blocks[i-1].end_{left}+1..n.blocks[i].end_{left}]$  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  $i+eb-n.left.blocks[b'-1].sum_{enq}$ . 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  $n.blocks[b].num_{enq-left}$  enqueues ahead of it in  $n.blocks[b]$  but not in  $n.right.blocks[n.blocks[i-1].end_{right}+1..n.blocks[i].end_{right}]$ . So we need to search for  $i - n.blocks[b].num_{enq-left} + n.blocks[b-1].sum_{enq-right}$  (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.  $\square$

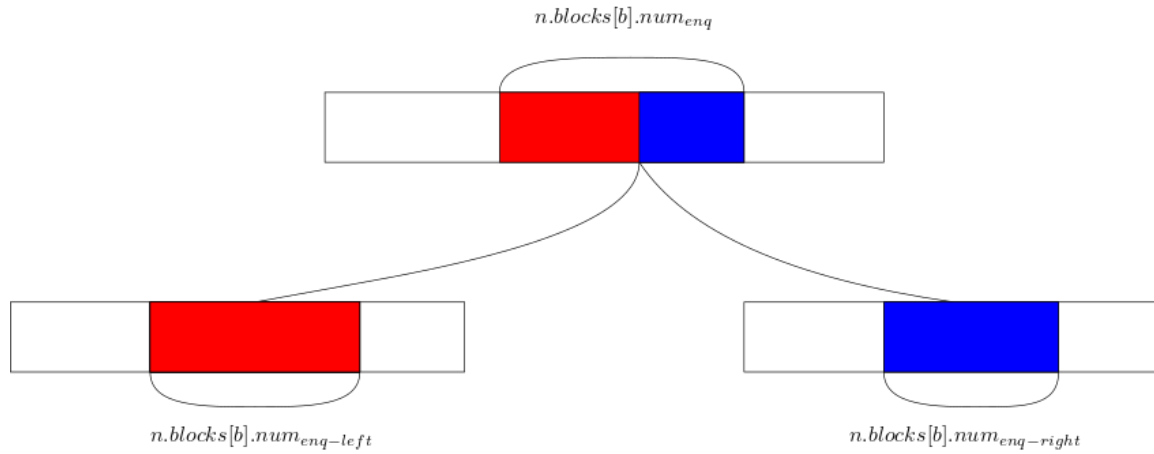


Figure 3: The number and ordering of the enqueues propagated from the left and the right child to  $n.blocks[b]$ . Enqueue operations from the left subblocks, colored red, are ordered before the enqueue operations from the right child, colored blue.

dsearch

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

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

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

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

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

□

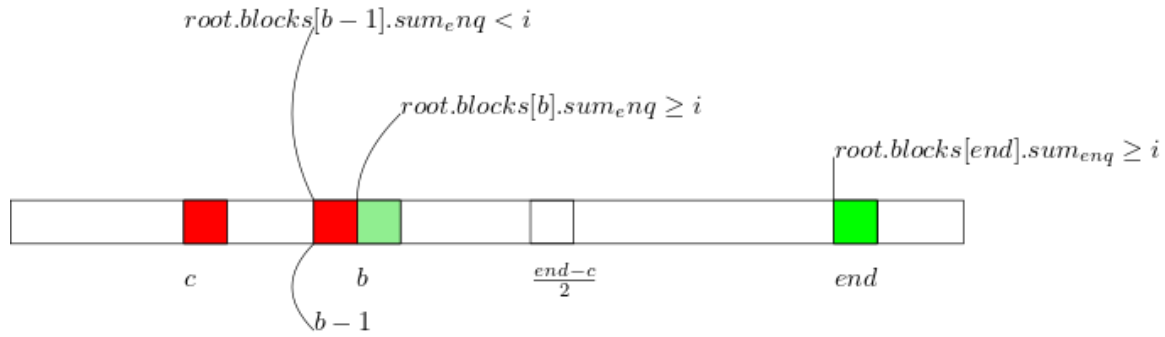


Figure 4: Distance relations between  $b, c, \text{end}$

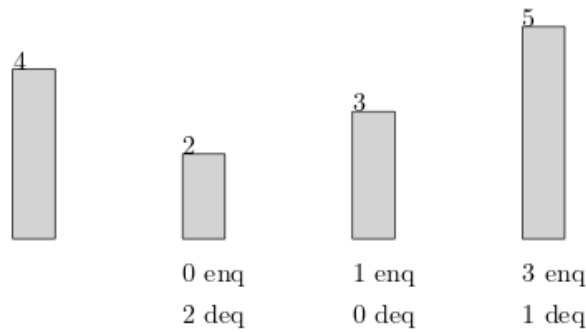


Figure 5: The number written on top of the bars is the queue size. the first block is  $b$  and the last block is  $\text{end}$ .

end-b

**Lemma 25** (Index correctness).  $n.\text{IndexDEQ}(b,i)$  returns the rank in  $D(\text{root})$  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  $\text{root}.\text{IndexDEQ}(b,i)$  is trivial (Line indexBaseCase 415). In the non-root nodes  $n$ ,  $\text{IndexDEQ}(b,i)$  computes the superblock of the  $i$ th Dequeue in the  $b$ th block of  $n$  in  $n.\text{parent}$  by Lemma superBlockcomputeSuper 26 (Line 418). After that the order in  $D(n.\text{parent}, \text{superblock})$  is computed and  $\text{index}()$  is called on  $n.\text{parent}$  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 22).  $\square$

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

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

*Proof.* Lemmas 28,29,30,31.  $\square$

**Lemma 27.** Value read for  $\text{super}[b.\text{group}]$  in line 418 is not null.

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

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

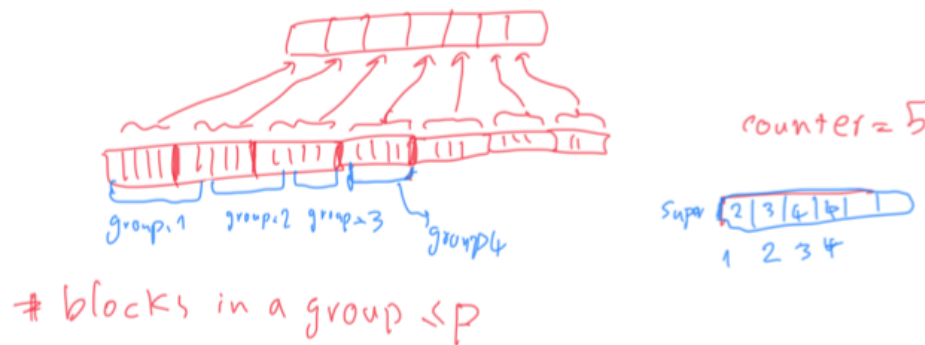
*Proof.* Line setGroup 329. Since  $\text{num}_{\text{propagated}}$  is increasing.  $\square$

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

*Proof.* Line setSuper 315.  $\square$

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

*Proof.* For the sake of simplicity we assumed all the blocks are propagated from the left child.  $\square$

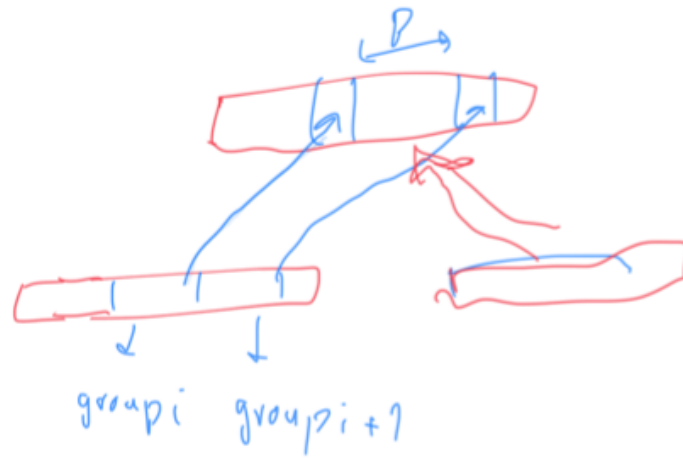


**Lemma 31.**  $\text{super}[i+1] - \text{super}[i] \leq p$

*Proof.* In a Refresh with successful CAS in line 46,  $\text{super}$  and  $\text{counter}$  are set for each child in lines 48,49. Assume the current value of the counter in node  $n$  is  $i+1$  and still  $\text{super}[i+1]$  is not set. If an instance of successful  $\text{Refresh}(n)$  finishes  $\text{super}[i+1]$  is set a new value and a block is added after  $n.\text{parent}[\text{super}[i]]$ . There could be at most  $p$  successful unfinished concurrent instances of  $\text{Refresh}()$  that have not reached line 49. So the distance between  $\text{super}[i+1]$  and  $\text{super}[i]$  is less than  $p$ .  $\square$

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

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



*Proof.*  $\text{super}[i]$  is the index of the superblock of a block containing block  $b$ , followed by Lemma <sup>superCounter</sup>32.  $\text{super}(b)$  is the real superblock of  $b$ .  $\text{super}(t)$  is the index of the superblock of the last block with time  $t$ . If  $b.\text{time}$  is  $t$  we have:

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

□

**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 is `null` (queue is empty) then  $R(d)=\text{null}$ .

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

**Observation 37.**  $k$ th non-null dequeue in an execution returns the element of  $k$ th enqueue.

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

*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 22 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 initially 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].sumenq` 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  $b$ th block using with this equality `root.blocks[b].size= root.blocks[b-1].size+ root.blocks[b].sumenq- root.blocks[b].sumdeq` (Line computeLength 342). See Table qhistory 1 for an example of running some blocks of operations on an empty queue.  $\square$

**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.sumenq-b.size`

*Proof.* There are `b.sumenq` 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.sumenq-b.size`. The correctness of the `block.size` field is shown in Lemma sizeCorrectness 38.  $\square$

**Lemma 40.**  $R(D_{\text{root},b,i})$  is null iff `root.blocks[b-1].size + root.blocks[b].numenq- i < 0`.

**Lemma 41** (Computing Response). `FindResponse(b,i)` returns  $R(D_{\text{root},b,i}).\text{element}$ .

*Proof.* First note that by Definition ordering 22 the linearization ordering of operations will not change as new operations come so instead of talking about the linearization of operations before the  $E_{\text{root},b,i}$  we talk about what if the whole operation in the linearization are applied on a queue.

$D_{\text{root},b,i}$  is  $D_{\text{root},\text{root.blocks}[b-1].\text{sum}_{\text{deq}}+i}$  from the definition ordering 22 and `sumenq`.  $D_{\text{root},b,i}$  returns null if `root.blocks[b-1].size + root.blocks[b].numenq- i < 0` by Lemma nullReturnCheckEmpty 40 (Line 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].sumenq - 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  $i$ th dequeue are non-null dequeues. So the response is  $E_{\text{root},i-\text{root.blocks}[b-1].\text{size}+\text{root.blocks}[b-1].\text{sum}_{\text{deq}}}$  (Line computeE 222). See figure computeResponseDetail 6.

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

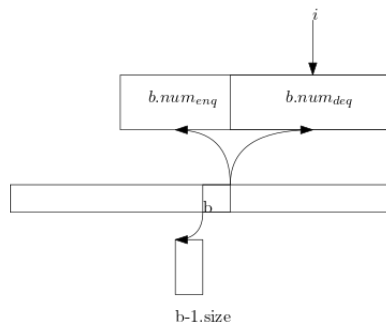


Figure 6: The position of  $E_{\text{root},b,i}$ .



	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

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.

qhistory

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

*Proof.* We choose  $L$  in Definition [22](#) to be linearization ordering of operations and prove if we linearize operations as  $L$  the queue works consistently.  $\square$

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

*Proof.* To show this we need to say if in an execution,  $op_1$  terminates before  $op_2$  starts then  $op_1$  is linearized before  $op_2$ . If  $op_1$  terminates before  $op_2$  starts it means  $op_1.\text{Append}()$  is terminated before  $op_2.\text{Append}()$  starts. From Lemma [11](#)  $op_1$  is in `root.blocks` before  $op_2$  propagates so  $op_1$  is linearized before  $op_2$  by Definition [22](#).  $\square$

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.  $\square$

**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 [11](#). The time  $op_1$  ends it is in root, before  $op_2$ , by Definition [22](#)  $op_1$  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 [11](#) 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 [23](#) we know that the returning response is the computed index element.

$\square$

**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 **Refreshes** in each node of the path from the leaf to the root (Lines <sup>firstRefresh</sup>302, <sup>secondRefresh</sup>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  $O(1)$  CASes. 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 <sup>superRange</sup>33) which takes  $O(\log^2 p)$  in the tree. In the **FindResponse()** routine **DSearch()** in the root takes  $\Theta(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$  by Lemma <sup>dsearch</sup>24, which is  $O(\log \text{size of the queue when enqueue is invoked}) + \log \text{size of the queue when dequeue is invoked})$ . Each search in **GetEnq()** takes  $O(\log p)$  since there are  $\leq p$  subblocks in a block (Lemma <sup>subBlocksBound</sup>21), 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 CASes, one in Line <sup>cas</sup>313 and one in Lines <sup>incrementInherentHead2</sup>318 or <sup>incrementInherentHead</sup>321. □