

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_{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 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} : indices 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})

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:   <bdeq, ideq>= INDEXDEQ(leaf.head, 1)
215:   output= FINDRESPONSE(bdeq, ideq)
216:   return output
217: end DEQUEUE

218: <int, int> FINDRESPONSE(int bd, int id)
    ▷ Returns the the response to the  $D_{root, b_d, i_d}$ . Returns null if the queue is empty.
219:   if   root.blocks[bd-1].size + root.blocks[bd].numenq - id < 0
    then
220:     return null
221:   else
222:     renq= id - root.blocks[bd-1].size + root.blocks[bd-1].sumenq
223:     return root.GetENQ(root.DSEARCH(renq, bd))
224:   end if
225: end FINDRESPONSE
```

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, 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 the
race to increase the head. The winner might have fallen sleep before increasing
head.
322:     return false
323:   end if
324: end REFRESH

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 numpropagated field. These values are used for
incrementing the children's numpropagated field if the block was appended to
blocks successfully.
328:   block newBlock = NEW(block)
329:   newBlock.group = numpropagated
330:   for each dir in {left, right} do
331:     indexlast = dir.head
332:     indexprev = blocks[i-1].enddir
333:     newBlock.enddir = indexlast
334:     blocklast = dir.blocks[indexlast]
335:     blockprev = dir.blocks[indexprev]
336:     ▷ newBlock includes dir.blocks[indexprev+1..indexlast].
337:     npdir = dir.numpropagated
338:     newBlock.sumenq-dir = blocks[i-1].sumenq-dir + blocklast.sumenq
- blockprev.sumenq
339:     newBlock.sumdeq-dir = blocks[i-1].sumdeq-dir + blocklast.sumdeq
- blockprev.sumdeq
340:   end for
341:   if this is root then
342:     newBlock.size = max(root.blocks[i-1].size + newBlock.numenq
- newBlock.numdeq, 0)
343:   end if
344:   return <b, npleft, npright>
345: end CREATEBLOCK

```

↪ 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

$$\rightsquigarrow \text{Precondition: } \text{root.blocks}[\text{end}].\text{sum}_{\text{eng}} \geq e$$

```

801: <int, int> DSEARCH(int e, int end)
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

```

▷ Returns <b,i> if $E_{root,e} = E_{root,b,i}$

Algorithm Node	
	<pre> \rightsquigarrow Precondition: <code>blocks[b].num_{enq} ≥ i</code> 401: <i>element</i> GETENQ(<i>int</i> b, <i>int</i> i) ▷ Returns the element of $E_{this,b,i}$. 402: if this is leaf then 403: return blocks[b].element 404: else if $i \leq \text{blocks}[b].\text{num}_{\text{enq-left}}$ then ▷ $E_{this,b,i}$ is in the left child of this node. 405: subBlock= left.BSEARCH(sum_{enq}, i+left.blocks[blocks[b].end_{left}-1].sum_{enq}, blocks[b-1].end_{left}+1, blocks[b].end_{left}) 406: return left.GETENQ(subBlock, i) 407: else 408: i= i-blocks[b].num_{enq-left} 409: subBlock= right.BSEARCH(sum_{enq}, i+right.blocks[blocks[b].end_{right}-1].sum_{enq}, blocks[b-1].end_{right}+1, blocks[b].end_{right}) 410: return right.GETENQ(subBlock, i) 411: end if 412: end GETENQ \rightsquigarrow Precondition: bth block of the node has propagated up to the root and <code>blocks[b].num_{enq} ≥ i</code>. 413: <<i>int</i>, <i>int</i>> INDEXDEQ(<i>int</i> b, <i>int</i> i) ▷ Returns <x, y> if $D_{this,b,i} = D_{root,x,y}$. 414: if this is root then 415: return <b, i> 416: else 417: dir= (parent.left==n)? left: right ▷ check if this node is a left or a right child 418: superBlock= parent.BSEARCH(sum_{deq-dir}, i+blocks[b-1].sum_{deq}, super[blocks[b].group]-p, super[blocks[b].group]+p) ▷ superblock's group has at most p difference with the value stored in <code>super[]</code>. 419: if dir is right then 420: i+= blocks[superBlock].num_{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 </pre>
tBaseCase	
ftOrRight	
tChildGet	
tChildGet	
xBaseCase	
puteSuper	
iderRight	
Algorithm Leaf	
	<pre> 601: <i>void</i> APPEND(<i>block</i> 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 </pre>
appendEnd	
pendStart	
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 ith 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_{left}</code>, <code>end_{right}</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: return CAS(blocks[n], null, blk) 703: end 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 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. 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.

Invariant 2 (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 $n.\text{head}$.

Some value is written into $n.\text{blocks}[\text{head}]$ only in Line 313. It is obvious that writing into $n.\text{blocks}[\text{head}]$ preserves the invariant. The value of $n.\text{head}$ is modified only in lines 318, 321. Depending on whether the `TryAppend()` in Line 313 succeeded or not we show that the claim holds after the increment lines of $n.\text{head}$ in either case. If 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 returns true or false after Line $n.\text{head}$ we know has been incremented from Line 310. The failure case is also the same since it means some value is written into $n.\text{blocks}[\text{head}]$ by some process. \square

Explain More

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

Consider the block b written into $n.\text{blocks}[i]$ by `TryAppend()` at Line 313. It is created by the `CreateBlock()` called at Line 311. Prior to this call to `CreateBlock()`, $n.\text{head}=i$ at Line 310, so $n.\text{blocks}[i-1]$ 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 $b.\text{end}_{\text{left}}$ at Line 333 of `CreateBlock()` was read from $n.\text{left.head}$ at Line 331 of `CreateBlock`. Similarly, the value in $n.\text{blocks}[i-1].\text{end}_{\text{left}}$ was read from $n.\text{left.head}$. 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 4 (Subblock). Block b is a *direct subblock* of $n.\text{blocks}[i]$ if it is in $n.\text{left.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1..n.\text{blocks}[i].\text{end}_{\text{left}}] \cup n.\text{right.blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}+1..n.\text{blocks}[i].\text{end}_{\text{right}}]$ (The end_{left} , $\text{end}_{\text{right}}$ fields are written in 333 of `CreateBlock()`). 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 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 .

Definition 6 (Operations of a block). A leaf block b in a leaf represents operation x if $b.\text{element}=x \neq \text{null}$. 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 $\text{ops}(b)$.

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

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 superbblock.

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, 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 $n.blocks[i-1]$. □

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

shedOrder **Lemma 11** (establishedOrder). If time $t < time t'$, 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 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.

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

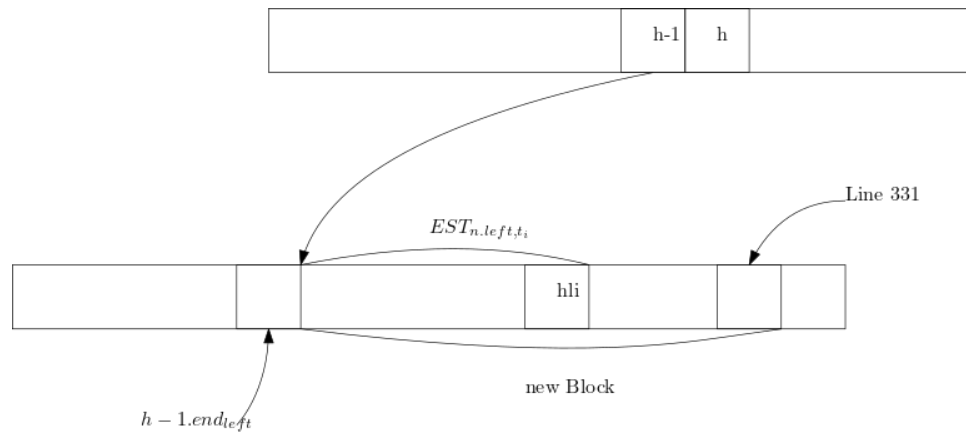


Figure 1: New established operations of the left child are in the new block.

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 $\mathbf{n.Refresh}()$ by some process. Let t_1 be the time R_1 is invoked and t_2 be the time R_2 terminated. We have $\text{ops}(\text{EST}_{\mathbf{n.left}}, t_1) \cup \text{ops}(\text{EST}_{\mathbf{n.right}}, t_1) \subseteq \text{ops}(\text{EST}_{\mathbf{n}}, t_2)$.*

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

Since R_2 's $\mathbf{TryAppend}()$ returns **false** then there is another successful instance R'_2 of $\mathbf{n.Refresh}()$ that has done $\mathbf{TryAppend}()$ successfully into $\mathbf{n.blocks}[i+1]$ before R_2 tries to append. In Figure 1 we see why the block R'_2 is appending contains established block in the \mathbf{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 $\text{ops}(\text{EST}_{\mathbf{n.left}}, t_1) \cup \text{ops}(\text{EST}_{\mathbf{n.right}}, t_1) \subseteq \text{ops}(\mathbf{n.blocks})$. 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 \mathbf{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 $\text{ops}(\text{EST}_{\mathbf{n.left}}, t_1) \cup \text{ops}(\text{EST}_{\mathbf{n.right}}, t_1) \subseteq \text{ops}(\text{EST}_{\mathbf{n}}, t_2)$. \square

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

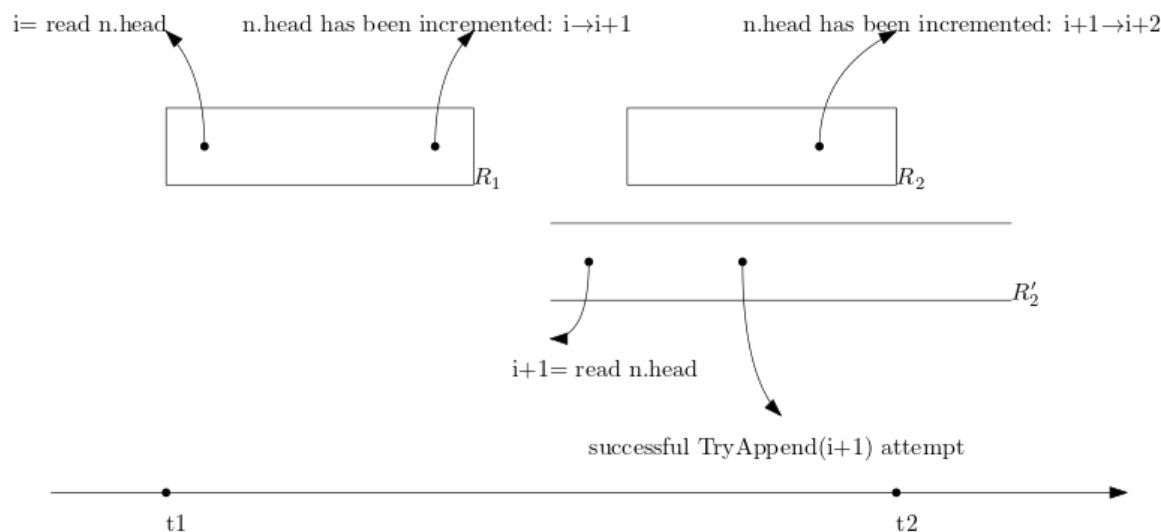


Figure 2: $t_1 < r_1$ reading head $<$ incrementing $\mathbf{n.head}$ from i to $i+1 < R'_2$ reading head $< \mathbf{TryAppend}(i+1) <$ incrementing $\mathbf{n.head}$ from $i+1$ to $i+2 < t_2$

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, 10. □

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 110. \square

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

Proof. It follows directly from Lemma blockSize 118 and the observation that each block contains at least one operation, induced from Line addOP 512. \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 118 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.

getSearch

Lemma 21 (Get BSearch correctness). *Preconditions of invocation of BSearch in GetENQ are satisfied i.e. if the direct subblock containing $E_{n,b,i}$ is computed in both cases in Lines leftChildGet 405 and rightChildGet 409.*

Proof. There are two cases that the subblock we're looking for is in the left child or the right child. If i th enqueue of $n.blocks[b]$ was in the left child it would be in $n.left.blocks[n.blocks[i-1].end_{left}+1..n.blocks[i].end_{left}]$ by definition def::subblock 4. sum_{enq} is the sum of the number of enqueues in the prefix of a block. So $E_{n,b,i}$ is $E_{n.left,i+n.left.blocks[n.blocks[b].end_{left}-1]}$ which is $E_{n.left,b',i'}$ for some b' and i' . The parameters in leftChildGet 405 are correct so this BSearch returns the subblock containing $E_{n,b,i}$.

If the enqueue we were looking for was in the right child as 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 $i-n.blocks[b].num_{enq-left}+n.right.blocks[n.blocks[b].end_{right}-1]$ (Line rightChildGet 409). Other parameters are the same as the left child. \square

get

Lemma 22 (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 the tree. The base case, the leaves of the tree, is pretty straight forward (Line getBaseCase 403). Since leaf blocks contain exactly one operation thus only $GetENQ(b,1)$ can be called on a leaf block b . $l.GetENQ(b,1)$ returns the operation stored in the b th block of leaf l . In the next paragraph we prove $n.GetENQ()$ returns $E_{n,b,i}$, if $n.child.GetENQ()$ works properly.

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 operations in a block are ordered by their process id and since in leaves of the tree are ordered ascendingly 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::ord 6 and def::subblock 4 we need to search in subblocks of $n.blocks[b]$ which their range is $n.left.blocks[n.blocks[i-1].end_{left}+1..n.blocks[i].end_{left}] \cup n.right.blocks[n.blocks[i-1].end_{right}+1..n.blocks[i].end_{right}]$. We can compute the direct subblock containing the enqueue we are finding for with binary search by Lemma getSearch 21. 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

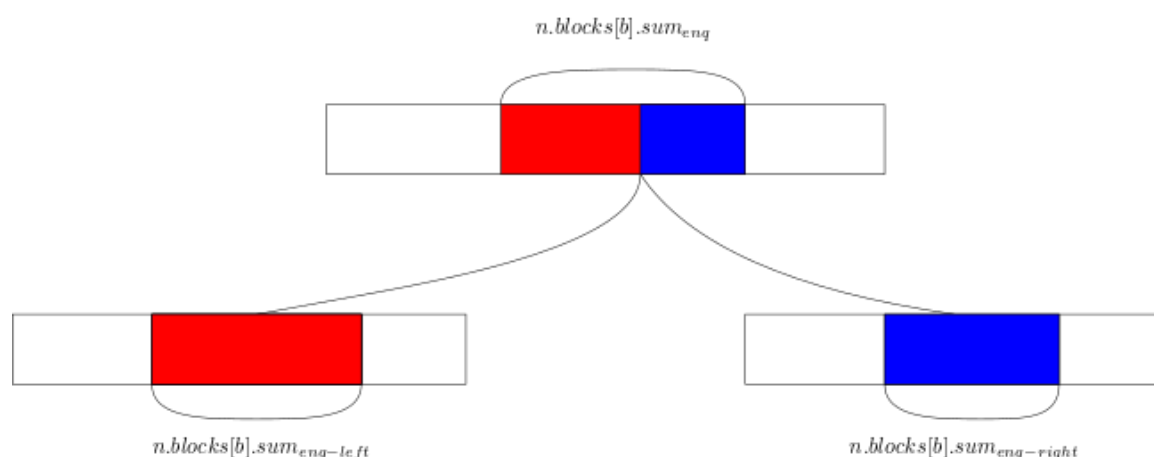


Figure 3: The number and ordering of the enqueues propagated from the left and the right child to $n.blocks[b]$. $n.blocks[b]$ is colored in the top node by the color of the subblocks propagated to it. Left subblocks in red come before the right subblocks in blue and $n.blocks[b].sum_{enq} = n.blocks[b].sum_{enq-left} + n.blocks[b].sum_{enq-right}$.

figGet

dsearch

Lemma 23 (DSearch correctness). Assume $\text{root.blocks}[\text{end}].\text{num}_{\text{enq}} \geq e$ and $E_{\text{root},e}$ is the response to some $\text{Dequeue}()$ in $\text{root.blocks}[\text{end}]$.

$\text{DSearch}(e, \text{end})$ returns $\langle b, i \rangle$ such $E_{\text{root},b,i} = E_{\text{root},e}$ in $\Theta(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$ steps.

Proof. First we show $\text{end}-b \leq 2 \times \text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size} + 1$. From line ^{addOP} 312, we know that size of the every block in the tree is greater than 0. So each block in $\text{root.blocks}[b..\text{end}]$ contains at least one $\text{Enqueue}()$ or at least 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 elements of the queue after $\text{blocks}[b]$ would become dequeued at some point, after $\text{root.blocks}[b]$ operations finish end before $\text{root.blocks}[\text{end}]$ operations start. And the response to to some $\text{dequeue}()$ in $\text{root.blocks}[\text{end}]$ could not be in $E(n, b)$. Furthermore since the size of the queue would become $\text{root.blocks}[\text{end}].\text{size}$ in the end and there cannot be more than $\text{root.blocks}[b].\text{size}$ $\text{Dequeue}()$ s so there cannot be more than $\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}$ $\text{enqueue}()$ operations in $\text{root.blocks}[b..\text{end}]$. Cause if there were more, then the $\text{root.blocks}[\text{end}]$'s size would become more than $\text{root.blocks}[\text{end}].\text{size}$. 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 ≥ 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 ^{end-b} 4.

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 $2 \times \text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}$. Cause if otherwise, then $(\text{end}-c)/2$ satisfied the $\text{root.blocks}[(\text{end}-c)/2].\text{sum}_{\text{enq}} < e$. 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_{enq} in constant time (Line ^{DSearchCompute i} 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. \square

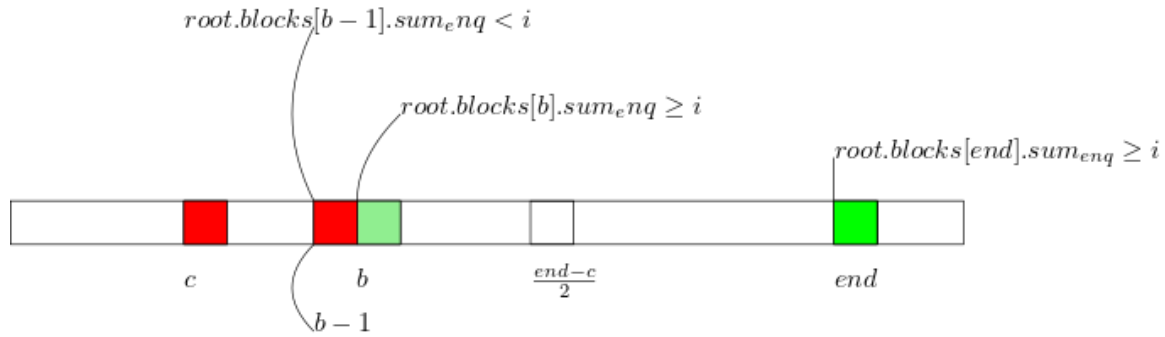


Figure 4: Distance relations between b, c, end

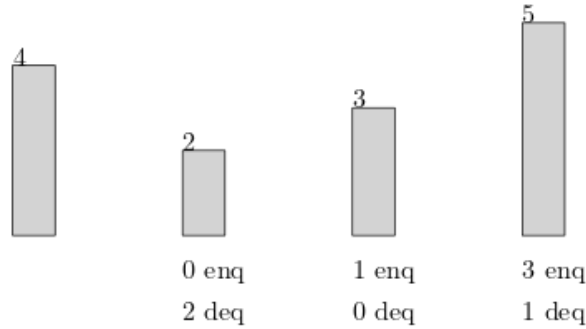


Figure 5: The number written on top of the bars is the queue size. the first block is b and the last block is end .

end-b

Lemma 24 (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 25 (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 20). \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 25 (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 26. Value read for $\text{super}[b.\text{group}]$ in line 418 is not null.

Proof. Values np_{dir} read in lines setNP 337, 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 27. $\text{super}[]$ 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 28. 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 29. 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

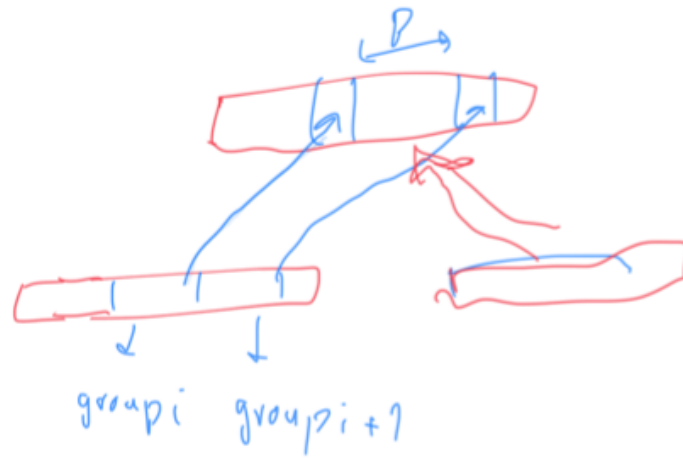


Lemma 30. $\text{super}[i+1] - \text{super}[i] \leq p$

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 $\text{super}[i+1]$ 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 31 (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 32. 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 ^{superCounter} 31. $\text{super}(b)$ 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 33. Search in each level of `IndexDeq()` takes $O(\log p)$ steps.

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

□

Update the findresponse returning arguments.

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.

computeHead

Lemma 34 (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. `FindResponse()` returns (b, i) which $E(\text{root}, b, i)$ is the head of the queue in S . If the queue is empty in S then it returns $\langle -1, -- \rangle$.*

Proof. 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 is stored in the `root.blocks[b].size`. It is computed while creating the block in Line ^{computeLength}542. 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. Having size of the queue after the previous block and number of enqueues and dequeues in the block, Line ^{computeLength}542 computes whether the queue becomes empty or the size of it.

$$r_{\text{enq}} = (i_d + \text{root.blocks}[b_d-1].\text{sum}_{\text{deq}}) - (\text{root.blocks}[b_d-1].\text{size} - \text{root.blocks}[b_d-1].\text{sum}_{\text{enq}} + \text{root.blocks}[b_d-1].\text{sum}_{\text{deq}})$$

□

[HOW?] How to prove mathematically that `ax(root.blocks[i-1].size + b.numenq - b.numdeq, 0)` is the size of the queue after the block. I can only explain it here.

Theorem 35 (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 36 (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 $op_1.Append()$ is terminated before $op_2.Append()$ starts. From Lemma ^{append}10 op_1 is in `root.blocks` before op_2 propagates so op_1 is linearized before op_2 by Definition ^{ordering}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 37 (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}10. 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}34 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}22 we know that the returning response is the computed index element. □

Lemma 38 (Amortized time analysis). **Enqueue()** and **Dequeue** take $O(\log^2 p + q)$ steps (amortized analysis), 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 **Refreshes** in each node of the path from the leaf to the root (Lines ~~302, 303~~ ^{firstRefresh, Refresh}). Each **Refresh** takes $O(1)$ steps since creating a block is done in constant time and does $O(1)$ CASes. 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 ~~32~~ ^{superRange}) 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 ~~23~~ ^{dsearch}, 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 ~~19~~ ^{subBlocksBound}), 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. □