3.2 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.

► *Node*

- **Node* left, right, parent : Initialized when creating the tree.
- *Block[]* blocks : Initially blocks[0] contains an empty block with all fields equal to 0.
- *int* head= 1: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.

► *Block*

- *int* super : approximate index of the superblock, read from parent.head when appending the block to the node

► *RootBlock* extends *InternalBlock*

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

► *InternalBlock* extends *Block*

- *int* end_{left}, $\text{end}_{\text{right}}$: indices of the last subblock of the block in the left and right child
- *int* $\text{sum}_{\text{enq-left}}$: # enqueues in $\text{left.blocks}[1..\text{end}_{\text{left}}]$
- *int* $\text{sum}_{\text{deq-left}}$: # dequeues in $\text{left.blocks}[1..\text{end}_{\text{left}}]$
- *int* $\text{sum}_{\text{enq-right}}$: # enqueues in $\text{right.blocks}[1..\text{end}_{\text{right}}]$
- *int* $\text{sum}_{\text{deq-right}}$: # dequeues in $\text{right.blocks}[1..\text{end}_{\text{right}}]$

► *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

Abbreviations used in the code and the proof of correctness.

- $\text{blocks[b].sum}_{\text{x}}=\text{blocks[b].sum}_{\text{x-left}}+\text{blocks[b].sum}_{\text{x-right}}$ (for b$\geq$0 and x $\in$ {enq, deq})
- $\text{blocks[b].num}_{\text{x}}=\text{blocks[b].sum}_{\text{x}}-\text{blocks[b-1].sum}_{\text{x}}$
  (for b>0 and x $\in$ { enq, deq, enq-left, enq-right, deq-left, deq-right})

Algorithm *Queue*

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

208: Object Dequeue()          ▷ Creates a block with null value element, appends it to the tree and returns its response.
209:     block newBlock= new(LeafBlock)
210:     newBlock.element= null
211:     newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq
212:     newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq+1
213:     leaf.Append(newBlock)
214:     <b, i>= IndexDequeue(leaf.head, 1)
215:     output= FindResponse(b, i)
216:     return output
217: end Dequeue

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

Algorithm Root

⇝ Precondition: `root.blocks[end].sum`$_{\texttt{enq}}$ $\geq$ `e`

```
801: <int, int> DSearch(int e, int end)                    ▷ Returns <b,i> such that E_e(root) = E_i(root, b).
802:     start= end-1
803:     while root.blocks[start].sum_enq≥e do
804:         start= max(start-(end-start), 0)
805:     end while
806:     b= root.BinarySearch(sum_enq, e, start, end)
807:     i= e- root.blocks[b-1].sum_enq
808:     return <b,i>
809: end DSearch
```

Algorithm Leaf

```
601: void Append(block blk)                                ▷ Only called by the owner of the leaf.
602:     blocks[head]= blk
603:     head+=1
604:     parent.Propagate()
605: end Append
```

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:     for each dir in {left, right} do
312:         h_dir= dir.head
313:         if dir.blocks[h_dir]!=null then
314:             dir.Advance(h_dir)
315:         end if
316:     end for
317:     new= CreateBlock(h)
318:     if new.num==0 then return true
319:     end if
320:     result= blocks[h].CAS(null, new)
321:     this.Advance(h)
322:     return result
323: end Refresh

324: void Advance(int h)
325:     h_p= parent.head
326:     blocks[h].super.CAS(null, h_p
327:     head.CAS(h, h+1)
328: end Advance

      ⇝ Precondition: blocks[start..end] contains a block with field f
      greater than or equal to i
      ▷ Does a 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.
329: int BinarySearch(field f, int i, int start, int end)
330:     return min{j:  blocks[j].f≥i}
331: end BinarySearch

      ▷ Creates and returns the block to be installed in blocks[i].
      Created block includes left.blocks[index_prev+1..index_last] and
      right.blocks[index_prev+1..index_last].
332: Block CreateBlock(int i)
333:     block new= new(block)
334:     for each dir in {left, right} do
335:         index_last= dir.head-1
336:         index_prev= blocks[i-1].end_dir
337:         new.end_dir= index_last
338:         block_last= dir.blocks[index_last]
339:         block_prev= dir.blocks[index_prev]
340:         new.sum_enq-dir= blocks[i-1].sum_enq-dir +
                  block_last.sum_enq - block_prev.sum_enq
341:         new.sum_deq-dir= blocks[i-1].sum_deq-dir +
                  block_last.sum_deq - block_prev.sum_deq
342:     end for
343:     if this is root then
344:         new.size = max(root.blocks[i-1].size  + new.num_enq
                  - new.num_deq, 0)
345:     end if
346:     return new
347: end CreateBlock
```

Algorithm Node

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

401: *element* GetEnqueue(*int* b, *int* i) ▷ Returns the element of $E_i(this, b)$.

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

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

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

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

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

407: **else**

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

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

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

411: **end if**

412: **end** GetEnqueue

⇝ Precondition: bth block of the node has propagated up to the root and `blocks[b].num`$_{\texttt{deq}} \geq$ `i`.

413: <*int*, *int*> IndexDequeue(*int* b, *int* i) ▷ Returns <x, y> if $D_i(this, b) = D_y(root, x)$.

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

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

416: **else**

417: dir= (parent.left==n ? left: right)

418: sb= (parent.blocks[blocks[b].super].sum$_{\texttt{deq-dir}}$ > blocks[b].sum$_{\texttt{deq}}$? blocks[b].super: blocks[b].super+1)

419: **if** dir **is** left **then**

420: i+= blocks[b-1].sum$_{\texttt{deq}}$-parent.blocks[sb-1].sum$_{\texttt{deq-left}}$

421: **else**

422: i+= blocks[b-1].sum$_{\texttt{deq}}$-parent.blocks[sb-1].sum$_{\texttt{deq-right}}$

423: i+= parent.blocks[sb].num$_{\texttt{deq-left}}$

424: **end if**

425: **return** this.parent.IndexDequeue(sb, i)

426: **end if**

427: **end** IndexDequeue

4 Proof of Correctness

We adopt linearizability as our definition of correctness. In our case, where we create the linearization ordering in the root, we need to prove (1) the ordering is legal, i.e, for every execution on our queue if operation op_1 terminates before operation op_2 then op_1 is linearized before operation op_2 and (2) if we do operations sequentially in their the linearization order, operations get the same results as in our queue. The proof is structured like this. First, we define and prove some facts about blocks and the node's `head` field. Then, we introduce the linearization ordering formally. Next, we prove double `Refresh` on a node is enough to propagate its children's new operations up to the node, which is used to prove (1). After this, we prove some claims about the size and operations of each block, which we use to prove the correctness of `DSearch()`, `GetEnqueue()` and `IndexDequeue()`. Finally, we prove the correctness of the way we compute the response of a dequeue, which establishes (2).

4.1 Basic Properties

In this subsection we talk about some properties of blocks and fields of the tree nodes.

A block is an object storing some statistics, as described in Algorithm Queue. A block in a node implicitly represents a set of operations.

Definition 1 (Ordering of a block in a node)**.** Let b be n`.blocks[`i`]` and b' be n`.blocks[`j`]`. We call i the *index* of block b. Block b is *before* block b' in node n if and only if $i < j$. We define *the prefix for* block b in node n to be the blocks in n`.blocks[0..`i`]`.

Next, we show that the value of `head` in a node can only be increased. By the termination of a `Refresh`, `head` has been incremented by the process doing the `Refresh` or by another process.

Observation 2. *For each node n, n.*`head` *is non-decreasing over time.*

Proof. The claim follows trivially from the code since `head` is only changed by incrementing in Line 327 of `Advance`. □

Lemma 3. *Let R be an instance of* `Refresh` *on a node n. After R terminates, n*`.head` *is greater than the value read in line 310 of R.*

Proof. If the `CAS` in Line 327 is successful then the claim holds. Otherwise n.`head` has changed from the value that was read in Line 310. By Observation 2 this means another process has incremented n.`head`. □

Now we show $n.\texttt{blocks}[n.\texttt{head}]$ is either the last block written into node n or the first empty block in n.

Invariant 4 (headPosition)**.** If the value of $n.\texttt{head}$ is h then $n.\texttt{blocks}[i] = \texttt{null}$ for $i > h$ and $n.\texttt{blocks}[i] \neq \texttt{null}$ for $0 \leq i < h$.

Proof. Initially the invariant is true since $n.\texttt{head} = 1$, $n.\texttt{blocks}[0] \neq \texttt{null}$ and $n.\texttt{blocks}[x] = \texttt{null}$ for every $x > 0$. The truth of the invariant may be affected by writing into `n.blocks` or incrementing `n.head`. We show that if the invariant holds before such a change then it still holds after the change.

In the algorithm, $n.\texttt{blocks}$ is modified only on Line 320, which updates $n.\texttt{blocks}[h]$ where h is the value read from $n.\texttt{head}$ in Line 310. Since the `CAS` in Line 320 is successful it means $n.\texttt{head}$ has not changed from h before doing the `CAS`: if $n.\texttt{head}$ had changed before the `CAS` then it would be greater than h by Observation 2 and hence $n.\texttt{blocks}[h] \neq \texttt{null}$ and by the induction hypothesis, so the `CAS` would fail. Writing into $n.\texttt{blocks}[h]$ when $h = n.\texttt{head}$ preserves the invariant, since the claim does not talk about the content of $n.\texttt{blocks}[n.\texttt{head}]$.

The value of `n.head` is modified only in Line 327 of `Advance`. If $n.\texttt{head}$ is incremented to $h + 1$ it is sufficient to show $\texttt{n.blocks}[h] \neq \texttt{null}$. `Advance` is called in Lines 314 and 321. For Line 314, $n.\texttt{blocks}[\texttt{h}] \neq \texttt{null}$ because of the `if` condition in Line 313. For Line 321, Line 320 was finished before doing 321. Whether Line 320 is successful or not, $n.\texttt{blocks}[\texttt{h}] \neq \texttt{null}$ after the $n.\texttt{blocks}[h].\texttt{CAS}$. □

We define the subblocks of a block recursively.

Definition 5 (Subblock)**.** A block is a *direct subblock* of the ith block in node n if it is in

$$n.\texttt{left.blocks}[n.\texttt{blocks}[i-1].\texttt{end}_{\texttt{left}}\texttt{+1} \cdots n.\texttt{blocks}[i].\texttt{end}_{\texttt{left}}]$$

or in

$$n.\texttt{right.blocks}[n.\texttt{blocks}[i-1].\texttt{end}_{\texttt{right}}\texttt{+1} \cdots n.\texttt{blocks}[i].\texttt{end}_{\texttt{right}}].$$

Block b is a *subblock* of block c if b is a direct subblock of c or a subblock of a direct subblock of c. We say block b is *propagated* to node n if b is in *n.blocks* or is a subblock of a block in $n.\texttt{blocks}$.

The next lemma is used to prove the subblocks of two blocks in a node are disjoint.

Lemma 6. *If* $n.\texttt{blocks}[i] \neq \texttt{null}$ *and* $i > 0$ *then* $n.\texttt{blocks}[i].\texttt{end}_{\texttt{left}} \geq n.\texttt{blocks}[i-1].\texttt{end}_{\texttt{left}}$ *and* $n.\texttt{blocks}[i].\texttt{end}_{\texttt{right}} \geq n.\texttt{blocks}[i-1].\texttt{end}_{\texttt{right}}$.

Proof. Consider the block b written into `n.blocks[`i`]` by `CAS` at Line 320. Block b is created by the `CreateBlock(`i`)` called at Line 317. Prior to this call to `CreateBlock(`i`)`, `n.head` $= i$ at Line 310, so `n.blocks[`$i-1$`]` is already a non-null value b' by Invariant 4. Thus, the `CreateBlock(`$i-1$`)` that created b' terminated before the `CreateBlock(`i`)` that creates b is invoked. The value written into b`.end`$_{\texttt{left}}$ at Line 337 of `CreateBlock(`i`)` was one less than the value read at Line 335 of `CreateBlock(`i`)`. Similarly, the value in `n.blocks[`$i-1$`].end`$_{\texttt{left}}$ was one less than the value read from `n.left.head` during the call to `CreateBlock(`$i-1$`)`. By Observation 2, n`.left.head` is non-decreasing, so b'`.end`$_{\texttt{left}} \leq b$`.end`$_{\texttt{left}}$. The proof for `end`$_{\texttt{right}}$ is similar. □

Lemma 7. *Subblocks of any two blocks in node n do not overlap.*

Proof. We are going to prove the lemma by contradiction. Consider the lowest `node` n in the tree that violates the claim. Then subblocks of n`.blocks[`i`]` and n`.blocks[`j`]` overlap for some $i < j$. Since n is the lowest node in the tree violating the claim, direct subblocks of blocks of n`.blocks[`i`]` and n`.blocks[`j`]` have to overlap. Without loss of generality assume left child subblocks of n`.blocks[`i`]` overlap with the left child subblocks of n`.blocks[`j`]`. By Lemma 6 we have n`.blocks[`i`].end`$_{\texttt{left}} \leq n$`.blocks[`$j-1$`].end`$_{\texttt{left}}$, so the ranges $[n$`.blocks[`$i-1$`].end`$_{\texttt{left}}+1 \cdots n$`.blocks[`$i$`].end`$_{\texttt{left}}]$ and $[n$`.blocks[`$j-1$`].end`$_{\texttt{left}}+1 \cdots n$`.blocks[`$j$`].end`$_{\texttt{left}}]$ cannot overlap. Therefore, direct subblocks of n`.blocks[`i`]` and n`.blocks[`j`]` cannot overlap. □

Definition 8 (Superblock)**.** Block b is *superblock* of block c if c is a direct subblock of b.

Corollary 9. *Every block has at most one superblock.*

Proof. A block having more than one superblock contradicts Lemma 7. □

Now we can define the operations of a block using the definition of subblocks.

Definition 10 (Operations of a block)**.** A block b in a leaf represents an `Enqueue()` if b`.element` $\neq$ `null`. Otherwise, if b`.element` $=$ `null`, b represents a `Dequeue()`. The set of operations of block b is the union of the operations in leaf subblocks of b. We denote the set of operations of block b by $ops(b)$ and the union of operations of a set of blocks B by $ops(B)$. We also say b contains op if $op \in ops(b)$.

Operations are distinct `Enqueue`s and `Dequeue`s invoked by processes. The next lemma proves that each operation appears at most once in the blocks of a node.

Lemma 11. *If op is in $n.blocks[i]$ then there is no $j \neq i$ such that op is in $n.blocks[j]$.*

Proof. We prove this claim using Lemma 7. Assume *op* is in the subblocks of both $n.blocks[i]$ and $n.blocks[j]$. From Corollary 7 we know that the subblocks of these blocks are different, so there are two leaf blocks containing *op*. Since each process puts each operation in only one block of its leaf then *op* cannot be in two leaf blocks. This is a contradiction. □

Definition 12. $n.\texttt{blocks}[i]$ is *established* at time t if $n.\texttt{head} > i$. An operation is *established* in node n if it is in an established block of n. EST_t^n is the set of established operations in node n at time t.

Now we want to say the blocks of a node grow over time.

Observation 13. *If time $t <$ time t' (t is before t'), then $ops(n.blocks)$ at time t is a subset of $ops(n.blocks)$ at time t'.*

Proof. Blocks are only appended (not modified) with `CAS` to $n.\texttt{blocks}[n.\texttt{head}]$, so the set of blocks of a node after the `CAS` contains the the set of blocks before the `CAS`. □

Corollary 14. *If time $t <$ time t', then $EST_n^t \subseteq EST_n^{t'}$.*

Proof. From Observations 2, 13. □

4.2 Ordering Operations

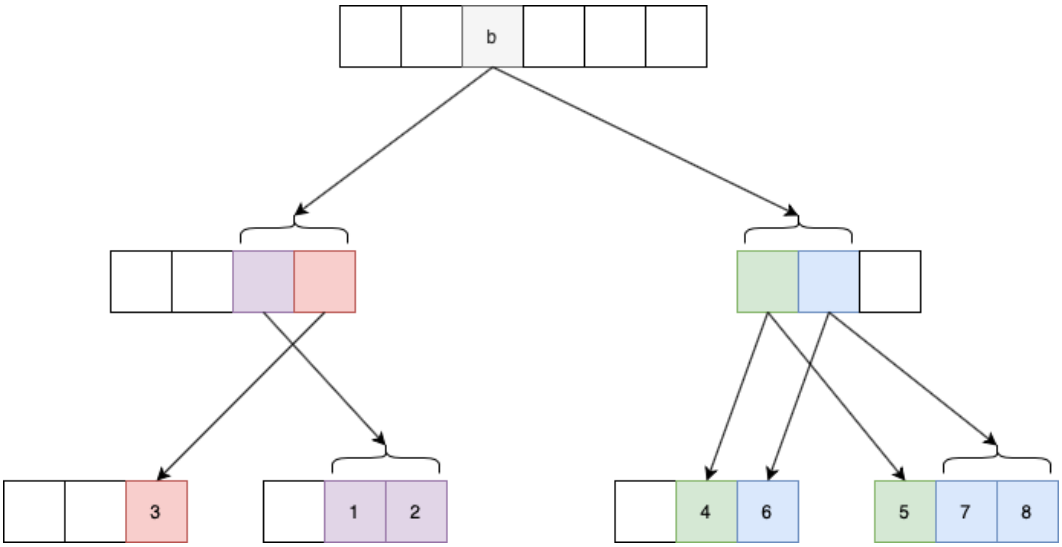

Figure 15: Order of operations in b. Operations in the leaves are ordered with numerical order shown in the drawing.

Now we define the ordering of operations stored in each node. In the non-root nodes we only need to order operations of a type among themselves. Processes are numbered from 1 to p and leaves of the tree are assigned from left to right. We will show in Lemma 28 that there is at most one operation from each process in a given block.

Definition 15 (Ordering of operations inside the nodes)**.**

- $E(n, b)$ is the sequence of enqueue operations in $ops(n.\texttt{blocks}[b])$ defined recursively as follows. $E(leaf, b)$ is the single enqueue operation in $ops(leaf.\texttt{blocks}[b])$ or an empty sequence if $leaf.\texttt{blocks}[b]$ represents a dequeue operation. If n is an internal node, then

$$\begin{aligned} E(n,b) =& E(n.\texttt{left}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{left}}+1)\cdots E(n.\texttt{left}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{left}})\cdot \\ & E(n.\texttt{right}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{right}}+1)\cdots E(n.\texttt{right}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{right}}). \end{aligned}$$

- $E_i(n, b)$ is the ith enqueue in $E(n, b)$.
- The order of the enqueue operations in the node n is $E(n) = E(n, 1) \cdot E(n, 2) \cdot E(n, 3) \cdots$
- $E_i(n)$ is the ith enqueue in $E(n)$.

- $D(n, b)$ is the sequence of dequeue operations in $ops(n.\texttt{blocks}[b])$ defined recursively as follows. $D(leaf, b)$ is the single dequeue operation in $ops(leaf.\texttt{blocks}[b])$ or an empty sequence if $leaf.\texttt{blocks}[b]$ represents an enqueue operation. If n is an internal node, then

$$\begin{aligned} D(n,b) =& D(n.\texttt{left}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{left}}+1) \cdots D(n.\texttt{left}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{left}}) \cdot \\ & D(n.\texttt{right}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{right}}+1) \cdots D(n.\texttt{right}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{right}}). \end{aligned}$$

- $D_i(n, b)$ is the ith enqueue in $D(n, b)$.
- The order of the dequeue operations in the node n is $D(n) = D(n, 1) \cdot D(n, 2) \cdot D(n, 3)...$
- $D_i(n)$ is the ith dequeue in $D(n)$.

The linearization ordering is given by the order that operations appear in the blocks in the root.

Definition 16 (Linearization)**.**

$$L = E(root, 1) \cdot D(root, 1) \cdot E(root, 2) \cdot D(root, 2) \cdot E(root, 3) \cdot D(root, 3) \cdots$$

Observation 17. *For any node n and indices $i < j$ of* `blocks` *in in, we have*

$$n.\texttt{blocks}[j].\texttt{sum}_{\texttt{x}} - n.\texttt{blocks}[i].\texttt{sum}_{\texttt{x}} = \sum_{k=i+1}^{j} n.\texttt{blocks}[k].\texttt{num}_{\texttt{x}}$$

where `x` *in* $\{\texttt{enq}, \texttt{deq}, \texttt{enq-left}, \texttt{enq-right}, \texttt{deq-left}, \texttt{deq-right}\}$.

Next claim is also true if we replace `enq` with `deq` and E with D.

Lemma 18. *Let B, B' be $n.\texttt{blocks}[b]$, $n.\texttt{blocks}[b-1]$ respectively.*

(1) If n is an internal node $B.\texttt{num}_{\texttt{enq-left}} = \left|E(n.\texttt{left}, B'.\texttt{end}_{\texttt{left}}+1) \cdots E(n.\texttt{left}, B.\texttt{end}_{\texttt{left}})\right|$.

(2) If n is an internal node $B.\texttt{num}_{\texttt{enq-right}} = \left|E(n.\texttt{right}, B'.\texttt{end}_{\texttt{right}}+1) \cdots E(n.\texttt{right}, B.\texttt{end}_{\texttt{right}})\right|$.

(3) $B.\texttt{num}_{\texttt{enq}} = \left|E(n, b)\right|$.

Proof. We prove the claim by induction on height of node n. Base case (3) for leaves is trivial. Supposing

the claim is true for n's children, we prove the correctness of the claim for n.

$$\begin{aligned}
B.\texttt{num}_{\texttt{enq-left}} =& B.\texttt{sum}_{\texttt{enq-left}} - B'.\texttt{sum}_{\texttt{enq-left}} && \text{Definition of } \texttt{num}_{\texttt{enq}} \\
=& B'.\texttt{sum}_{\texttt{enq-left}} + n.\texttt{left.blocks}[B.\texttt{end}_{\texttt{left}}].\texttt{sum}_{\texttt{enq}} \\
& - n.\texttt{left.blocks}[B'.\texttt{end}_{\texttt{left}}].\texttt{sum}_{\texttt{enq}} - B'.\texttt{sum}_{\texttt{enq-left}} && \texttt{CreateBlock} \\
=& n.\texttt{left.blocks}[B.\texttt{end}_{\texttt{left}}].\texttt{sum}_{\texttt{enq}} - n.\texttt{left.blocks}[B'.\texttt{end}_{\texttt{left}}].\texttt{sum}_{\texttt{enq}} \\
=& \sum_{i=B'.\texttt{end}_{\texttt{left}}+1}^{B.\texttt{end}_{\texttt{left}}} n.\texttt{left.blocks}[i].\texttt{num}_{\texttt{enq}} && \text{Observation 17} \\
=& \left| E(n.\texttt{left}, B'.\texttt{end}_{\texttt{left}} + 1) \cdots E(n.\texttt{left}, B.\texttt{end}_{\texttt{left}}) \right| && \text{Induction hypothesis (3)}
\end{aligned}$$

The last line holds because of the induction hypothesis (3). (2) is similar to (1). Now we prove (3) starting from the Definition of $E(n, b)$.

$$\begin{aligned}
E(n, b) =& E(n.\texttt{left}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{left}} + 1) \cdots E(n.\texttt{left}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{left}}) \cdot \\
& E(n.\texttt{right}, n.\texttt{blocks}[b-1].\texttt{end}_{\texttt{right}} + 1) \cdots E(n.\texttt{right}, n.\texttt{blocks}[b].\texttt{end}_{\texttt{right}}).
\end{aligned}$$

By (1) and (2) we have $\left|E(n, b)\right| = B.\texttt{num}_{\texttt{enq-left}} + B.\texttt{num}_{\texttt{enq-right}} = B.\texttt{num}_{\texttt{enq}}$. □

Next claim is also true if we replace `enq` with `deq` and E with D.

Corollary 19. *Let B be $n.\texttt{blocks}[b]$ and* `enq` *be in* $\{\texttt{enq}, \texttt{deq}\}$.

(1) If n is an internal node $B.\texttt{sum}_{\texttt{enq-left}} = \left|E(n.\texttt{left}, 1) \cdots E(n.\texttt{left}, B.\texttt{end}_{\texttt{left}})\right|$

(2) If n is an internal node $B.\texttt{sum}_{\texttt{enq-right}} = \left|E(n.\texttt{right}, 1) \cdots E(n.\texttt{right}, B.\texttt{end}_{\texttt{right}})\right|$

(3) $B.\texttt{sum}_{\texttt{enq}} = \left|E(n, 1) \cdot E(n, 2) \cdots E(n, b)\right|$

4.3 Propagating Operations to the Root

In this section we explain why two `Refresh`es are enough to propagate a nodes operations to its parent.

Definition 20. Let t^{op} be the time *op* is invoked, ${}^{op}t$ be the time *op* terminates, t_l^{op} be the time immediately before running Line l of operation *op* and ${}_l^{op}t$ be the time immediately after running Line l of operation *op*. We sometimes suppress *op* and write t_l or ${}_lt$ if *op* is clear in the context. In the text v_l is the value of variable `v` immediately after line l for the process we are talking about and v_t is the value of variable `v` at time t.

Definition 21 (Successful Refresh)**.** An instance of `Refresh` is *successful* if its `CAS` in Line 320 returns `true`. If a successful instance of `Refresh` terminates, we say it is *complete.*

In the next two results we show for every successful `Refresh`, all the operations established in the children before the `Refresh` are in the parent after the `Refresh`'s successful `CAS` at Line 320.

Lemma 22. *If R is a successful instance of n.`Refresh`, then we have $EST_{n.\texttt{left}}^{t^R} \cup EST_{n.\texttt{right}}^{t^R} \subseteq ops(n.\texttt{blocks}_{320})$.*

Proof. We show

$$EST_{n.\texttt{left}}^{t^R} = ops(n.\texttt{left.blocks[0..n.left.head}_{309} - 1\texttt{]})$$
$$\subseteq ops(n.\texttt{blocks}_{320}) = ops(n.\texttt{blocks[0..}n.\texttt{head}_{320}\texttt{]}).$$

Line 320 stores a block `new` in n that has $\texttt{end}_{\texttt{left}} = n.\texttt{left.head}_{335} - 1$. Therefore, by Definition 5, after the successful `CAS` in Line 320 we know all blocks in $n.\texttt{left.blocks[}1 \cdots n.\texttt{left.head}_{335} - 1\texttt{]}$ are subblocks of $\texttt{n.blocks[}1 \cdots n.\texttt{head}_{310}\texttt{]}$. Because of Lemma 2 we have $n.\texttt{left.head}_{309} - 1 < n.\texttt{left.head}_{335} - 1$ and $n.\texttt{head}_{310} < n.\texttt{head}_{320}$. From Observation 13 the claim follows. The proof for the right child is the same. □

Corollary 23. *If R is a complete instance n.`Refresh`, then we have $EST_{n.\texttt{left}}^{t^R} \cup EST_{n.\texttt{right}}^{t^R} \subseteq EST_n^{R}t$.*

Proof. The left hand side is the same as Lemma 22, so it is sufficient to show when R terminates the established blocks in n are a superset of $n.\texttt{blocks}_{320}$. Line 320 writes the block `new` in $n.\texttt{blocks[}h\texttt{]}$ where h is value of $n.\texttt{head}$ read at Line 310. Because of Lemma 3 we are sure that $n.\texttt{head} > h$ when R terminates. So the block `new` appended to n at Line 320 is established at ${}^{R}t$. □

In the next lemma we show that if two consecutive instances of `Refresh` by the same process on node n fail, then the blocks established in the children of n before the first `Refresh` are guaranteed to be in n after the second `Refresh`.

Lemma 24. *Consider two consecutive terminating instances R_1, R_2 of* `Refresh` *on internal node n by process p. If neither R_1 nor R_2 is a successful* `Refresh`*, then we have $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq EST^{^{R_2}t}_{n}$.*

Proof. Let R_1 read i from $n.\texttt{head}$ at Line 310. By Lemma 3, R_1 and R_2 both cannot read the same value i. By Observation 2, R_2 reads a larger value of $n.\texttt{head}$ than R_1.

Consider the case where R_1 reads i and R_2 reads $i+1$ from Line 310. As R_2's `CAS` in Line 320 returns `false`, there is another successful instance R'_2 of $n.\texttt{Refresh}$ that has done a `CAS` successfully into $n.\texttt{blocks}[i+1]$ before R_2 tries to `CAS`. R'_2 creates its block `new` after reading the value $i+1$ from $n.\texttt{head}$ (Line 310) and R_1 reads the value i from $n.\texttt{head}$. By Observation 2 we have ${}^{R_1}t < t^{R_1}_{310} < t^{R2'}_{310}$ (see Figure 16). By Lemma 23 we have $EST^{n.\texttt{left}}_{\substack{R'_2\\310}t} \cup EST^{n.\texttt{right}}_{\substack{R'_2\\310}t} \subseteq ops(n.\texttt{blocks}_{\substack{R'_2\\320}})$. Also by Lemma 3 on R_2, the value of `n.head` is more than $i+1$ after R_2 terminates, so the block appended by R'_2 to `n` is established by the time R_2 terminates. To summarize, ${}^{R_1}t$ is before R'_2's read of $n.\texttt{head}$ ($t^{R'_2}_{310}$) and R'_2's successful `CAS` ($t^{R'_2}_{320}$) is before R_2's termination (t^{R_2}), so by Observation and Lemma 3 we have 13 $EST^{t^{R_1}}_{n.\texttt{left}} \cup EST^{t^{R_1}}_{n.\texttt{right}} \subseteq ops(n.\texttt{blocks}_{\substack{R'_2\\320}}) \subseteq EST^{^{R_2}t}_{n.\texttt{left}}$.

If R_2 reads some value greater than $i+1$ in Line 310 it means $n.\texttt{head}$ has been incremented more than two times since ${}^{R_1}_{310}t$. By Lemma 4, when $n.\texttt{head}$ is incremented from $i+1$ to $i+2$, $n.\texttt{blocks[i+1]}$ is non-null. Let R_3 be the `Refresh` on n that has put the block in $n.\texttt{blocks}[i+1]$. R_3 read $n.\texttt{head} = i+1$ at Line 310 and has put its block in $n.\texttt{blocks}[i+1]$ before R_2's read of $n.\texttt{head}$ at Line 310. So we have $t^{R_1} <^{R_3}_{310} t <^{R_3}_{320} t < t^{R_2}_{310} <^{R}_{2} t$. From Observation 13 on the operations before and after R_3's `CAS` and Lemmas 22, 3 on R_3 the claim holds. □

Corollary 25. $EST^{^{302}t}_{n.\texttt{left}} \cup EST^{^{302}t}_{n.\texttt{right}} \subseteq EST^{t_{303}}_{n}$

Proof. If the first `Refresh` in line 302 returns `true` then by Lemma 23 the claim holds. If the first `Refresh` failed and the second `Refresh` succeeded the claim still holds by Lemma 23. Otherwise both failed and the claim is satisfied by Lemma 24. □

Now we show that after `Append(`b`)` on a leaf finishes, the operation contained in b will be established in `root`.

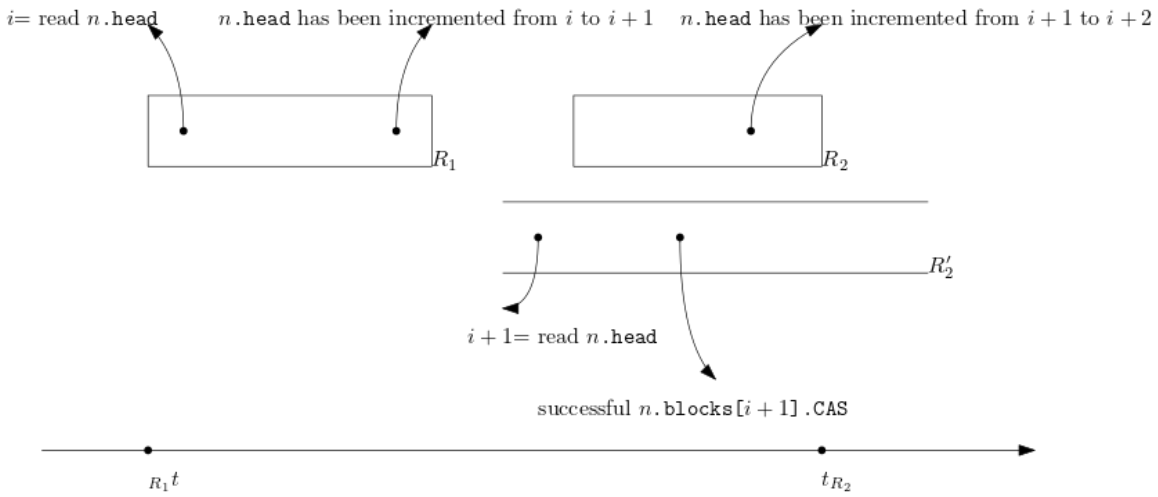

Figure 16: $_{R_1}t < t_{310}^{R_1} <$ incrementing n.head from i to $i+1 < t_{310}^{R'_2} < t_{320}^{R'_2} <$ incrementing n.head from $i+1$ to $i+2 < t_{R_2}$

Corollary 26. *For $A = l$.Append(b) we have $ops(b) \subseteq EST_n^{t^A}$ for each node n in the path from l to* root.

Proof. A adds b to the assigned leaf of the process, establishes it at Line 603 and then calls Propagate on the parent of the leaf where it appended b. For every node n, n.Propagate appends b to n, establishes it in n by Corollary 25 and then calls n.parent.Propagate untill n is root. □

Corollary 27. *After l.Append(b) finishes, b is subblock of exactly one block in each node along the path from l to the* root.

Proof. By the previous corollary and Lemma 27 there is exactly one block in each node containing b. □

4.4 Correctness of GetEnqueue

First we prove some claims about the size and operations of a block. These lemmas will be used later for the correctness and analysis of `GetEnqueue()`.

Lemma 28. *Each block contains at most one operation of each process, and therefore at most p operations in total.*

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process p in block b in node n. Without loss of generality op_1 is invoked earlier than op_2. Process p cannot invoke more than one operation concurrently, so op_1 has to be finished before op_2 begins. By Corollary 27, before op_2 calls `Append`, op_1 exists in every node of the tree on the path from p's leaf to the root. Since b contains op_2, it must be created after op_2 is invoked. This means there is some block b' before b in n containing op_1. The existence of op_1 in b and b' contradicts Lemma 11. □

Lemma 29. *Each block has at most p direct subblocks.*

Proof. The claim follows directly from Lemma 28 and the observation that each block appended to an internal node contains at least one operation, due to the test on Line 318. We can also see the blocks in the leaves have exactly one operation in the `Enqueue()` and `Dequeue()` routines. □

`DSearch(`e`, `end`)` returns a pair `<`b`, `i`>` such that the ith `Enqueue` in the bth block of the root is the eth `Enqueue` in the entire sequence stored in the root.

Lemma 30 (`DSearch` Correctness)**.** *If* `root.blocks[`end`]` $\neq$ `null` *and* $1 \leq e \leq$ `root.blocks[`end`].sum`$_{\texttt{enq}}$*,* `DSearch(`e`, `end`)` *returns* `<`b`, `i`>` *such that* $E_i(root, b) = E_e(root)$.

Proof. From Lines 340 and 341 we know the `sum`$_{\texttt{enq-left}}$ and `sum`$_{\texttt{enq-right}}$ fields of `blocks` in each node are sorted in non-decreasing order. Since `sum`$_{\texttt{enq}}$ = `sum`$_{\texttt{enq-left}}$ + `sum`$_{\texttt{enq-right}}$, the `sum`$_{\texttt{enq}}$ values of `root.blocks[`$0 \cdots end$`]` are also non-decreasing. Furthermore, since `root.blocks[0].sum`$_{\texttt{enq}} = 0$ and `root.blocks[`end`].sum`$_{\texttt{enq}} \geq e$, there is a b such that `root.blocks[`b`].sum`$_{\texttt{enq}} \geq e$ and `root.blocks[`$b-1$`].sum`$_{\texttt{enq}} < e$ by Lemma 19. Block `root.blocks[`b`]` contains $E_i(root, b)$. Lines 802–805 doubles the search range in Line 804 and will eventually reach `start` such that `root.blocks[start].sum`$_{\texttt{enq}} \leq e \leq$ `root.blocks[end].sum`$_{\texttt{enq}}$. Then, in Line 806, the binary search finds the b such that `root.blocks[`$b-1$`].sum`$_{\texttt{enq}} < e \leq$ `root.blocks[`b`].sum`$_{\texttt{enq}}$. By Corollary 19, `root.blocks[`b`]` is the block that contains $E_e(root)$. Finally i is computed using the definition of `sum`$_{\texttt{enq}}$ and Corollary 19. □

Lemma 31 (`GetEnqueue` correctness). *If* $1 \leq i \leq n.\texttt{blocks[}b\texttt{]}.\texttt{num}_{\texttt{enq}}$ *then* $n.\texttt{GetEnqueue(}b\texttt{, }i\texttt{)}$ *returns* $E_i(n,b).\texttt{element}$.

Proof. We are going to prove this lemma by induction on the height of node n. For the base case, suppose n is a leaf. Leaf blocks each contain exactly one operation, $n.\texttt{blocks[}b\texttt{]}.\texttt{sum}_{\texttt{enq}} \leq 1$, which means only $n.\texttt{GetEnqueue(b,1)}$ can be called when n is a leaf. Line 403 of $n.\texttt{GetEnqueue(}b\texttt{, 1)}$ returns the `element` of the `Enqueue` operation stored in the bth block of leaf n, as required.

For the induction step we prove if $n.\texttt{child.GetEnqueue(}b'\texttt{, }i\texttt{)}$ returns $E_i(n.\texttt{child}, b')$ then $n.\texttt{GetEnqueue(}b\texttt{, }i\texttt{)}$ returns $E_i(n,b)$. From Definition 15 of $E(n,b)$, so operations from the left subblocks come before the operations from the right subblocks in a block (see Figure 17). By Observation 18, the $\texttt{num}_{\texttt{enq-left}}$ field in $n.\texttt{blocks[}b\texttt{]}$ is the number of `Enqueue()` operations from the blocks's subblocks in the left child of n. So the ith `Enqueue` operation in $n.\texttt{blocks[}b\texttt{]}$ is propagated from the right child if and only if i is greater than $n.\texttt{blocks[}b\texttt{]}.\texttt{num}_{\texttt{enq-left}}$. Line 404 decides whether the ith enqueue in the bth block of internal node n is in the left child or right child subblocks of $n.\texttt{blocks[}b\texttt{]}$. By Definitions 5 and 10 to find an operation in the subblocks of $n.\texttt{blocks[}i\texttt{]}$ we need to search in the range

$$n.\texttt{left.blocks[}n.\texttt{blocks[}i\texttt{-1]}.\texttt{end}_{\texttt{left}}\texttt{+1..}n.\texttt{blocks[}i\texttt{]}.\texttt{end}_{\texttt{left}}\texttt{]} \text{ or}$$

$$n.\texttt{right.blocks[}n.\texttt{blocks[}i\texttt{-1]}.\texttt{end}_{\texttt{right}}\texttt{+1..}n.\texttt{blocks[}i\texttt{]}.\texttt{end}_{\texttt{right}}\texttt{]}.$$

First we consider the case where the `Enqueue` we are looking for is in the left child. There are $eb = n.\texttt{blocks[}b-1\texttt{]}.\texttt{sum}_{\texttt{enq-left}}$ `Enqueue`s in the blocks of $n.\texttt{left}$ before the left subblocks of $n.\texttt{blocks[}b\texttt{]}$, so $E_i(n,b)$ is $E_{i+eb}(n.\texttt{left})$ which is $E_{i'}(n.\texttt{left}, b')$ for some b' and i'. We can compute b' and then search for the $i+eb$th enqueue in `n.left`, where i' is $i + eb - n.\texttt{left.blocks[}b'-1\texttt{]}.\texttt{sum}_{\texttt{enq}}$. The parameters in Line 405 are for searching $E_{i+eb}(n.left)$ in $n.\texttt{left.blocks}$ in the range of left subblocks of $n.\texttt{blocks[}b\texttt{]}$, so this `BinarySearch` returns the index of the subblock containing $E_i(n,b)$.

Otherwise, the enqueue we are looking for is in the right child. Because `Enqueue`s from the left subblocks are ordered before the `Enqueue`s from the right subblocks, there are $\texttt{n.blocks[b]}.\texttt{num}_{\texttt{enq-left}}$ enqueues ahead of $E_i(n,b)$ from the left child. So we need to search for $i - n.\texttt{blocks[}b\texttt{]}.\texttt{num}_{\texttt{enq-left}} + n.\texttt{blocks[}b-1\texttt{]}.\texttt{sum}_{\texttt{enq-right}}$ in the right child (Line 409). Other parameters for the right child are chosen similarly to the left child.

So, in both cases the direct subblock containing $E_i(n,b)$ is computed in Line 405 or 409. Finally, $n.\texttt{child.GetEnqueue(}\textit{subblock}\texttt{, }i\texttt{)}$ is invoked on the subblock containing $E_i(n,b)$ and it returns $E_i(n,b).\texttt{element}$

by the hypothesis of the induction. □

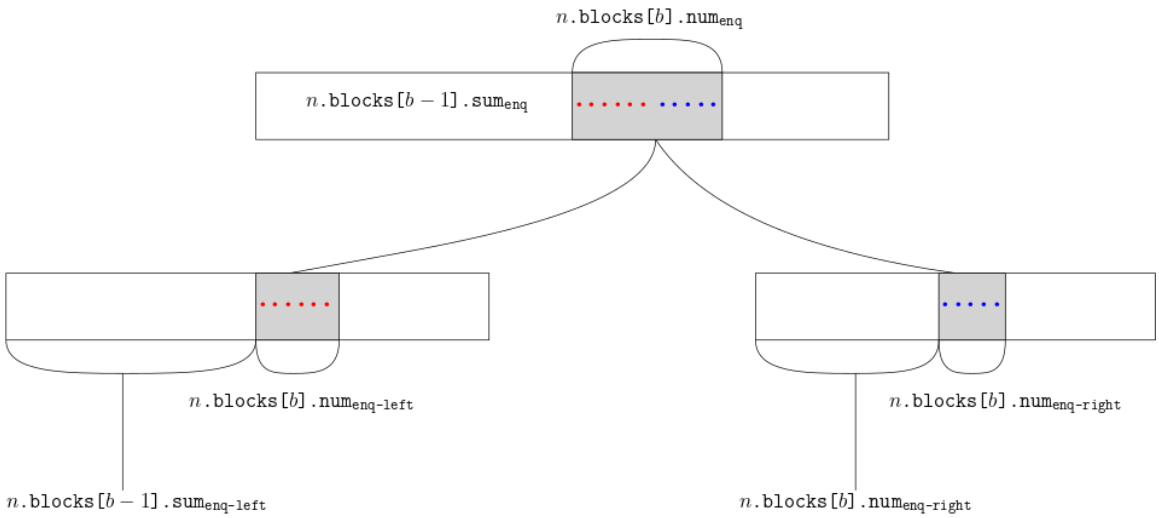

Figure 17: The number and ordering of the enqueue operations propagated from the left and the right child to $n.\texttt{blocks}[b]$. Both $n.\texttt{blocks}[b]$ and its subblocks are shown in grey. `Enqueue` operations from the left subblocks (colored red), are ordered before the `Enqueue` operations from the right child (colored blue).

4.5 Correctness of IndexDequeue

The next few results show that the `super` field of a block is accurate within one of the actual index of the block's superblock in the parent node. Then we explain how it is used to compute the rank of a given `dequeue` in the root.

Definition 32. If a `Refresh` instance R_1 does its `CAS` at Line 320 earlier than `Refresh` instance R_2 we say R_1 has *happened before* R_2.

Observation 33. *After* n`.blocks[`i`].CAS(null,` B`)` *succeeds,* n`.head` *cannot increase from* i *to* $i+1$ *until* B`.super` *is set.*

Proof. From Observation 2 we know the n`.head` changes only by the increment on Line 327. Before an instance of `Advance` increments `n.head` on Line 327, Line 326 ensures that `n.blocks[head].super` was set at Line 326. □

Corollary 34. *If* n`.blocks[`i`].super` *is* `null`*, then* n`.head` $\leq i$ *and* n`.blocks[`$i+1$`]` *is* `null`*.*

Proof. By Lemma 4 and Observation 33. □

Now let us consider how the `Refresh`es that took place on the parent of node n after block B was stored in n will help to set B`.super` and propagate B to the parent.

Observation 35. *If the block created by an instance* R_p *of* n`.parent.Refresh` *contains block* $B = n$`.blocks[`b`]` *then* R_p *reads a value greater than* b *from* n`.head` *in Line 335.*

Lemma 36. *If* $B = n$`.blocks[`b`]` *is a direct subblock of* n`.parent.blocks[`sb`]` *then* B`.super` $\leq sb$*.*

Proof. Let R_p be the instance of n`.parent.Refresh` that stores n`.parent.blocks[`sb`]`. By 35 if R_p propagates B it has to read a greater value than b from n`.head`, which means n`.head` was incremented from b to $b+1$ in Line 327. By Observation 33 B`.super` was already set in Line 326. The value written in B`.super` was read in Line 325,s before the `CAS` that sets B`.super`. From Observation 2 we know n`.parent.head` is non-decreasing so B`.super` $\leq sb$, since n`.parent.head` is still equal to sb when R_p executes its `CAS` at Line 320 by Invariant 6. The reader may wonder when the case b`.super` $= sb$ happens. This can happen when n`.parent.blocks[`B`.super]` = `null` when B`.super` is written and R_p puts its created block into n`.parent.blocks[`B`.super]` afterwards. □

Lemma 37. *Let R_n be a* `Refresh` *that puts B in n.*`blocks[`b`]` *at Line 320. Then, the block created by one of the next two successful n.*`parent.Refresh`*es according to Definition 32 contains B and B.*`super` *is set when the second successful n.*`parent.Refresh` *reaches Line 317.*

Proof. Let R_{p1} be the first successful n.`parent.Refresh` after R_n and R_{p2} be the second next successful n.`parent.Refresh`. To derive a contradiction assume B was not propagated to n.`parent` by R_{p1} nor by R_{p2}.

Since R_{p2}'s created block does not contain B, by Observation 35 the value R_{p2} reads from n.`head` in Line 335 is at most b. From Observation 2 the value R_{p2} reads in Line 312 is also at most b.

R_n puts B into n.`blocks[`b`]` so R_n reads the value b from n.`head`. Since R_{p2}'s `CAS` into n.`parent.blocks` is successful there should be a `Refresh` instance R'_p on n.`parent` that increments n.`parent` (Line 327) after R_{p1}'s Line 320 and before R_{p2}'s Line 310. We assumed $t^{R_n}_{320} < t^{R_{p1}}_{320} < t^{R_{p2}}_{320}$ by Definition 32. Finally, Line 312 is after Line 310 and R_{p2}'s 310 is after R'_p's Line 327, which is after R_n's n.`blocks.CAS`.

$$\left.\begin{array}{r} {}^{R_n}_{320}t <^{R_{p1}}_{320} t \\ {}^{R_{p1}}_{320}t <^{R_{p'}}_{327} t <^{R_{p2}}_{310} t \\ {}^{R_{p2}}_{310}t <^{R_{p2}}_{312} t \end{array}\right\} \Longrightarrow {}^{R_n}_{320}t <^{R_{p2}}_{312} t$$

So R_{p2} reads a value greater than or equal to b for n.`head` by Lemma 2.

Therefore R_{p2} reads n.`head` $= b$. R_{p2} calls n.`Advance` at Line Line 314, which ensures n.`head` is incremented from b. So the value R_{p2} reads in Line 335 of `CreateBlock` is greater than b and R_{p2}'s created block contains B. This is contradiction with our hypothesis.

Furthermore, if B.`super` was not set earlier it is set by R_{p2} call to n.`Advance` invoked from Line 314.

□

Corollary 38. *If $B = n$.*`blocks[`b`]` *is propagated to n.*`parent`*, then B.*`super` *is equal to or one less than the index of the superblock of B.*

Proof. Let R_n be the n.`Refresh` that put B in n.`blocks` and let R_{p1} be the first successful n.`parent.Refresh` after R_n and R_{p2} be the second next successful n.`parent.Refresh`. Before B can be propagated to n's `parent`, n.`head` must be greater than b, so by Observation 33 B.`super` is set. From thr previous Lemma we know that B is propagated by second next successful `Refresh`'s `CAS` on n.`parent.blocks`. To summarize we have n.`parent.head`$_{{}^{R_{p2}}_{320}t}$ $= n$.`parent.head`$_{{}^{R_{p1}}_{320}t} + 1$ and by Definition 32 and Observation 2

$n.\texttt{parent.head}_{\substack{R_{p1}\\320}t} \leq n.\texttt{parent.head}_{\substack{R_n\\320}t}$. The value that is set in $B.\texttt{super}$ is read from $n.\texttt{parent.head}$ after $\substack{R_n\\320}t$. So $B.\texttt{super}$ is equal to or one less than the index of the superblock of B. □

Now using Corollary 38 on each step of the `IndexDequeue` we prove its correctness.

Lemma 39 (`IndexDequeue` correctness)**.** *If* $1 \leq i \leq n.\texttt{blocks}[b].\texttt{num}_{\texttt{deq}}$ *then* $n.\texttt{IndexDequeue}(b,i)$ *returns* $< x, y >$ *such that* $D_i(n,b) = D_y(\texttt{root}, x)$.

Proof. We will prove this by induction on the distance of n from the `root`. The base case where n is `root` is trivial (see Line 415). For the non-root nodes $n.\texttt{IndexDequeue}(b,\ i)$ computes sb, the index of the superblock of the bth block in n, in Line 418 by Corollary 38. After that, the position of $D_i(n,b)$ in $D(n.\texttt{parent}, sb)$ is computed in Lines 419–424. By Definition 15, `Dequeues` in a block are ordered based on the order of its subblocks from left to right. If $D_i(n,b)$ was propagated from the left child, the number of dequeus in the left subblocks of $n.\texttt{parent.blocks}[sb]$ before $n.\texttt{blocks}[b]$ is considered in Line 420 (see Figure 18). Otherwise, if $D_i(n,b)$ was propagated from the right child, the number of dequeues in the subblocks from the left child is considered to be ahead of the computed index (Line 421) (see Figure 19). Finally `IndexDequeue` is called on $n.\texttt{parent}$ recursively and it returns the correct response by induction hypothesis. □

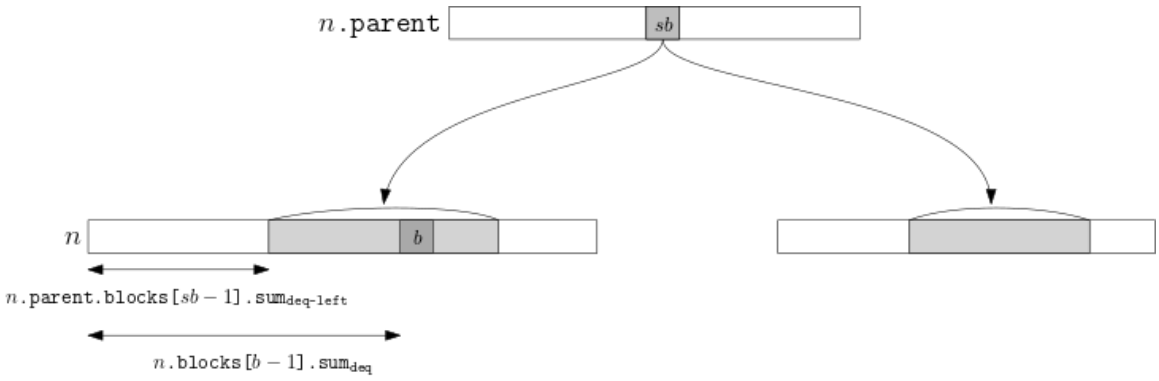

Figure 18: The number of `dequeue` operations before $E_i(n,b)$ shown in the case where n is a left child.

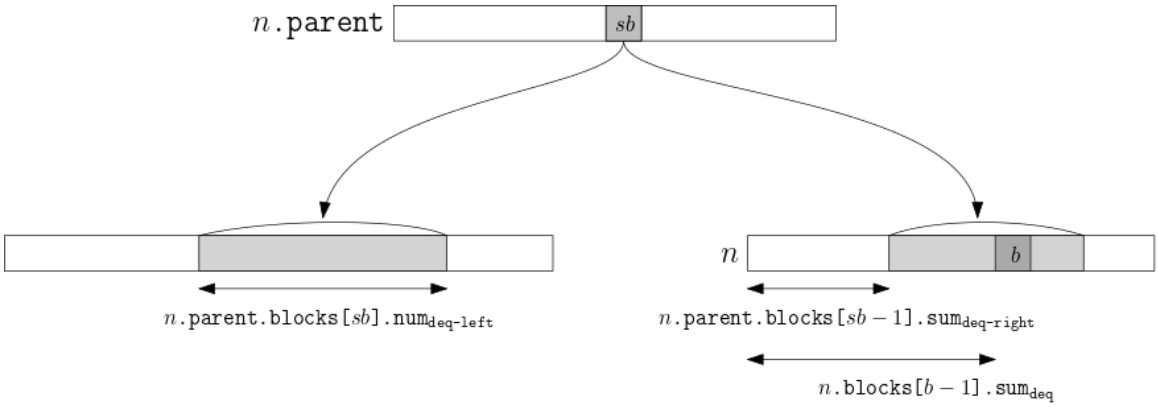

Figure 19: The number of `Dequeue` operations before $E_i(n, b)$ shown in the case where n is a right child.

4.6 Linearizability

We now prove the two properties needed for linearizability.

Lemma 40. *L is a legal linearization ordering.*

Proof. We must show that, every operation that terminates is in L exactly once and if op_1 terminates before op_2 starts in execution then op_1 is before op_2 in the linearization. The first claim is directly reasoned from Lemma 27. For the latter, if op_1 terminates before op_2 starts, op_1`.Append` has terminated before op_2`.Append` started. From Lemma 26, op_1 is in `root.blocks` before op_2 starts to propagate. By definition of L, op_1 is linearized before op_2. □

Once some operations are aggregated in one block, they will get propagated up to the root together and they can be linearized in any order among themselves. We have chosen to put `Enqueue`s in a block before `Eequeue`s (see Definition 15).

Definition 41. If a `Dequeue` operation returns null it is called a *null* `Dequeue`, otherwise it is called *non-null* `Dequeue`.

Next we define the responses that `Dequeue`s should return, according to the linearization.

Definition 42. Assume the operations in `root.blocks` are applied sequentially on an empty queue in the order of L. $Resp(d) = e$`.element` if the element of `Enqueue` e is the response to `Dequeue` d. Otherwise if d is a null dequeue then $Resp(d) =$ `null`.

In the next lemma we show that the `size` field in each `root block` is computed correctly.

Lemma 43. `root.blocks[`b`].size` *is the size of the queue if the operations in* `root.blocks[`$0 \cdots b$`]` *are applied in the order of* L.

Proof. We prove the claim by induction on b. The base case when $b = 0$ is trivial since the queue is initially empty and `root.blocks[0].size` $= 0$. We are going to show the correctness when $b = i$ assuming correctness when $b = i - 1$. By Definition 15 `Enqueue` operations come before `Dequeue` operations in a block. By Lemma 18 $\texttt{num}_{\texttt{enq}}$ and $\texttt{num}_{\texttt{deq}}$ fields in a block show ther number of `Enqueue` and `Dequeue` operations in it. If there are more than `root.blocks[`$i-1$`].size` $+$ `root.blocks[`i`].num`$_{\texttt{enq}}$ dequeue operations in `root.blocks[`i`]` then the queue would become empty after `root.blocks[`i`]`. Otherwise the size of the queue after the bth block in the root is `root.blocks[`$b-1$`].size` $+$ `root.blocks[`b`].num`$_{\texttt{enq}}$ $-$ `root.blocks[`b`].num`$_{\texttt{deq}}$. In both cases, this is same as the assignment on Line 344. □

The next lemma is useful to compute the number of non-null dequeues.

Lemma 44. *If operations in the root are applied with the order of* L*, the number of non-null* `Dequeue`*s in* `root.blocks[`$0 \cdots b$`]` *is* `root.blocks[`b`].sum`$_{\texttt{enq}}$ $-$ `root.blocks[`b`].size`.

Proof. There are `root.blocks[`b`].sum`$_{\texttt{enq}}$ enqueue operations in `root.blocks[`$0 \cdots b$`]`. The size of the queue after doing `root.blocks[`$0 \cdots b$`]` in order L is the number of *enqueues* in `root.blocks[`$0 \cdots b$`]` minus the number of *non-null* `Dequeue`*s* in `root.blocks[`$0 \cdots b$`]`. By the correctness of the `size` field from Lemma 43 and $\texttt{sum}_{\texttt{enq}}$ field from Lemma 18, the number of *non-null* `Dequeue`*s* is `root.blocks[`b`].sum`$_{\texttt{enq}}$ $-$ `root.blocks[`b`].size`. □

Corollary 45. *If operations in the root are applied with the order of* L*, the number of non-null dequeues in* `root.blocks[`b`]` *is* `root.blocks[`b`].num`$_{\texttt{enq}}$ $-$ `root.blocks[`b`].size` $+$ `root.blocks[`$b-1$`].size`.

Lemma 46. $Resp(D_i(\texttt{root}, b))$ *is* `null` *iff* `root.blocks[`$b-1$`].size` $+$ `root.blocks[`b`].num`$_{\texttt{enq}} - i < 0$.

Proof. From Corollary 45 and Lemma 18. □

Lemma 47. `FindResponse(`b`,` i`)` *returns* $Resp(D_i(root, b))$.

Proof. $D_i(root, b)$ is $D_{\texttt{root.blocks[}b-1\texttt{].sum}_{\texttt{deq}}+i}(root)$ by Definition 15 and Lemma 19. $D_i(root, b)$ returns `null` at Line 220 if `root.blocks[`$b-1$`].size` $+$ `root.blocks[`b`].num`$_{\texttt{enq}} - i < 0$ and $Resp(D_i(root, b)) =$ `null` in this case by Lemma 46. Otherwise, if $D_i(root, b)$ is the eth non-null dequeue in L it should return the eth enqueued value. By Lemma 44 there are `root.blocks[`$b-1$`].sum`$_{\texttt{enq}}$ $-$ `root.blocks[`$b-1$`].size` non-null

`Dequeue` operations in `root.blocks`$[0 \cdots b-1]$. The `Dequeues` in `root.blocks`$[b]$ before $D_i(root, b)$ are non-null dequeues. So $D_i(root, b)$ is the eth non-null `Dequeue` where $e = i + $ `root.blocks`$[b-1]$`.sum`$_{\texttt{deq}} -$ `root.blocks`$[b-1]$`.size` (Line 222). See Figure 20.

After computing e at Line 222, the code finds b,i such that $E_i(root, b) = E_e(root)$ using `DSearch` and then finds its `element` using `GetEnqueue` (Line 223). Correctness of `DSearch` and `GetEnqueue` routines are shown in Lemmas 30 and 31. □

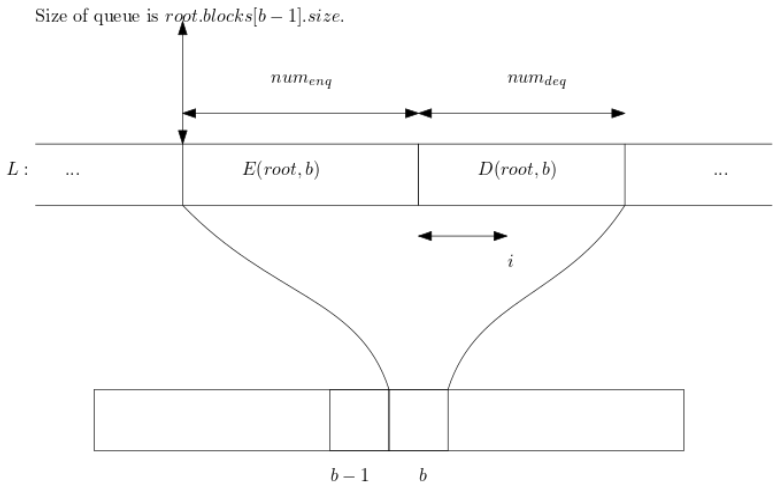

Figure 20: The position of $D_i(root, b)$.

Lemma 48. *The responses to operations in our algorithm would be the same as in the sequential execution in the order given by L.*

Proof. `Enqueue` operations do not return any value. By Lemma 47 response of a `Dequeue` in our algorithm is same as the response from the sequential execution of L. □

Theorem 49 (Main)**.** *The queue implementation is linearizable.*

Proof. The theorem follows from Lemmas 40 and 48. □

Remark In fact our algorithm is strongly linearizable as defined in [5]. By Definition 15 the linearization ordering of operations will not change as blocks containing new operations are appended to the root.

5 Analysis

Lemma 50 (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.*

Lemma 51 (`DSearch` Analysis)**.** *If the* `element` *enqueued by* $E_i(root, b) = E_e(root)$ *is the response to some* `Dequeue()` *operation in* `root.blocks[end]`*, then* `DSearch(`e`,` end`)` *takes* $O\big(\log(\texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size})\big)$ *steps.*

Proof. First we show $end - b - 1 \leq 2 \times \texttt{root.blocks[}b-1\texttt{].size} + \texttt{root.blocks[}end\texttt{].size}$. Suppose there were more than `root.blocks[b].size` `Dequeue`s in $\texttt{root.blocks[}b+1 \cdots end-1\texttt{]}$. Then the element in the queue which is the response to the `Dequeue()` would become dequeued at some point before $\texttt{root.blocks[}end\texttt{]}$'s first `Dequeue()`. Furthermore in the execution of queue operations in the linearization ordering, the size of the queue becomes $\texttt{root.blocks[}end\texttt{].size}$ after the operations of $\texttt{root.blocks[}end\texttt{]}$. There can be at most $\texttt{root.blocks[}b\texttt{].size}$ `Dequeue`s in $\texttt{root.blocks[}b+1 \cdots end-1\texttt{]}$; otherwise all elements enqueued by $\texttt{root.blocks[}b\texttt{]}$ would be dequeued before $\texttt{root.blocks[}end\texttt{]}$. The final size of the queue after $\texttt{root.blocks[}1 \cdots end\texttt{]}$ is $\texttt{root.blocks[}end\texttt{].size}$. After an execution on a queue the *size* of the queue is greater than or equal to $\#enqueues - \#dequeues$ in the execution.We know the number of dequeues in $\texttt{root.blocks[}b+1 \cdots end-1\texttt{]}$ is less than $\texttt{root.blocks[}b\texttt{].size}$, therefore there cannot be more than $\texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size}$ `Enqueue`s in $\texttt{root.blocks[}b+1 \cdots end-1\texttt{]}$. Overall there can be at most $2 \times \texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size}$ operations in $\texttt{root.blocks[}b+1 \cdots end\texttt{]}$ and since from line 318 we know that `num` field of the every block in the tree is greater than 0, each block has at least one operation, there are at most $2 \times \texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size}$ blocks in between $\texttt{root.blocks[}b\texttt{]}$ and $\texttt{root.blocks[}end\texttt{]}$. So $end - b - 1 \leq 2 \times \texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size}$.

So the doubling search reaches `start` such that the $\texttt{root.blocks[start].sum}_{\texttt{enq}}$ is less than e in $O\big(\log(\texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size})\big)$ steps. See Figure 21. After Line 805, the binary search that finds b also takes $O\big(\log(\texttt{root.blocks[}b\texttt{].size} + \texttt{root.blocks[}end\texttt{].size})\big)$. Next, `i` is computed via the definition of $\texttt{sum}_{\texttt{enq}}$ in constant time (Line 807). So the claim is proved. □

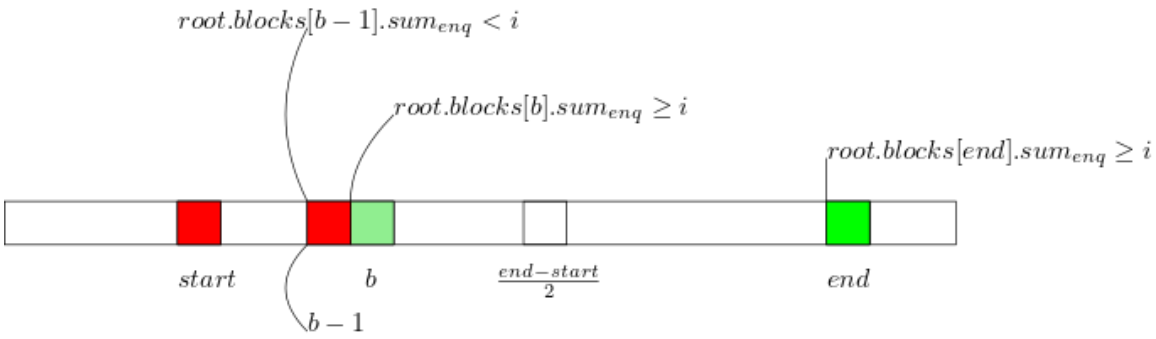

Figure 21: Distance relations between `start`, b, end.

5.1 Garbage Collection or Getting rid of the infinite Arrays