1 Pseudocode

Algorithm Fields description

◇ *Shared*

- *Tree* `tree` : A binary tree of `Node`s. `root` is a pointer to the root node.

◇ *Local*

- **Node* `leaf:` a pointer to the process's leaf in the tree.

◇ *Structures*

▶ *Node*

- **Node* `left, right, parent` : initialized when creating the tree.
- *BlockList* `blocks` implemented with an array.
- *int* `size`= 1: #`blocks` in `blocks`.
- *int* $\texttt{num}_{\texttt{propagated}}$= 0 : # groups of blocks that have been propagated from the node to its parent. Since it is incremented after propagating, it may be behind by 1.
- *int[]* `super`: `super[i]` stores an approximate index of the superblock of the blocks in `blocks` whose `group` field have value `i`.

▶ *Leaf* `extends` *Node*

- *int* $\texttt{last}_{\texttt{done}}$

  Stores the index of the block in the root such that the process that owns this leaf has most recently finished the. A block is finished if all of its operations are finished. `enqueue(e)` is finished if `e` is returned by some `dequeue()` and `dequeue()` is finished when it computes its response. *put the definitions before the pseudocode*

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

- *int* `group` : the value read from $\texttt{num}_{\texttt{propagated}}$ when appending this block to the node.

▶ *LeafBlock* `extends` *Block*

- *Object* `element` : Each block in a leaf represents a single operation. For enqueue operations element is the input of the `enqueue` and for `dequeue` operations it is `null`.
- *Object* `response` : stores the response of the operation in the `LeafBlock`.
- *int* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: # enqueue, dequeue operations in the prefix for the block

▶ *InternalBlock* `extends` *Block*

- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: index of the last subblock of the block in the left and right child
- *int* $\texttt{sum}_{\texttt{enq-left}}$: # enqueue operations in the prefix for $\texttt{left.blocks[end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-left}}$: # dequeue operations in the prefix for $\texttt{left.blocks[end}_{\texttt{left}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: # enqueue operations in the prefix for $\texttt{right.blocks[end}_{\texttt{right}}\texttt{]}$
- *int* $\texttt{sum}_{\texttt{deq-right}}$: # dequeue operations in the prefix for $\texttt{right.blocks[end}_{\texttt{right}}\texttt{]}$

▶ *RootBlock* `extends` *InternalBlock*

- *int* `length` : length of the queue after performing all operations in the prefix for this block
- *counter* $\texttt{num}_{\texttt{finished}}$: number of finished operations in the block

Variable naming:

- $\texttt{b}_{\texttt{op}}$: index of the block containing operation`op`
- $\texttt{r}_{\texttt{op}}$: rank of operation `op` i.e. the ordering among the operations of its type according to linearization ordering

Abbreviations:

- $\texttt{blocks[b].sum}_{\texttt{x}}\texttt{=blocks[b].sum}_{\texttt{x-left}}\texttt{+blocks[b].sum}_{\texttt{x-right}}$ (for b≥0 and x ∈ {enq, deq})
- $\texttt{blocks[b].sum=blocks[b].sum}_{\texttt{enq}}\texttt{+blocks[b].sum}_{\texttt{deq}}$ (for b≥0)
- $\texttt{blocks[b].num}_{\texttt{x}}\texttt{=blocks[b].sum}_{\texttt{x}}\texttt{-blocks[b-1].sum}_{\texttt{x}}$

  (for b>0 and x ∈ {∅, enq, deq, enq-left, enq-right, deq-left, deq-right}, $\texttt{blocks[0].num}_{\texttt{x}}\texttt{=0}$)

Algorithm *Queue*

```
void Enqueue(Object e)    ▷ Creates a block with element e and appends it to the tree.
    block newBlock= new(LeafBlock)
    newBlock.element= e
    newBlock.sum_enq= leaf.blocks[leaf.size].sum_enq+1
    newBlock.sum_deq= leaf.blocks[leaf.size].sum_deq
    leaf.Append(newBlock)
end Enqueue

Object Dequeue()
    block newBlock= new(LeafBlock)    ▷ Creates a block with null value element, appends it to the tree, computes its order among operations, then computes and returns its response.
    newBlock.element= null
    newBlock.sum_enq= leaf.blocks[leaf.size].sum_enq
    newBlock.sum_deq= leaf.blocks[leaf.size].sum_deq+1
    leaf.Append(newBlock)
    return leaf.HelpDequeue()
end Dequeue

<int, int> FindResponse(int b, int i)    ▷ Computes the rank and index of the block in the root of the enqueue that is the response of the ith dequeue in the root's bth block. Returns <-1,--> if the queue is empty.
    if root.blocks[b-1].length + root.blocks[b].num_enq - i < 0 then
        return <-1,-->
    else
        ▷ We call the dequeues that return a value non-null dequeues. rth non-null dequeue returns the element of th rth enqueue. We can compute # non-null dequeues in the prefix for a block this way: #non-null dequeues= length - #enqueues. Note that the ith dequeue in the given block is not a non-null dequeue.
        r_enq= root.blocks[b-1].sum_enq- root.blocks[b-1].length + i
        return <root.BSearch(sum_enq, r_enq, root.FindMostRecentDone(), root.size), r_enq>
    end if
end FindResponse
```

Algorithm *Node*

```
void PROPAGATE()
firstRefresh    if not REFRESH() then
secondRefresh       REFRESH()                              ▷ Lemma Double Refresh
    end if
    if this is not root then
        parent.PROPAGATE()
    end if
end PROPAGATE

boolean REFRESH()
readSize    s= size
    <new, np_left, np_right>= CREATEBLOCK(h)     ▷ np_left, np_right are the values read from the children's num_propagated feild.
    if new.num==0 then return true               ▷ The block contains nothing.
    else if blocks.tryAppend(new, s) then
okcas       for each dir in {left, right} do
            CAS(dir.super[np_dir], null, h+1)    ▷ Write would work too.
            CAS(dir.num_propagated, np_dir, np_dir+1)
        end for
        CAS(size, h, h+1)
        return true
    else
        CAS(size, h, h+1)                        ▷ Even if another process wins, help to increase the size. The winner might have fallen sleep before increasing size.
        return false
    end if
end REFRESH

⇝ Precondition: blocks[start..end] contains a block with field f ≥ i
int BSEARCH(field f, int i, int start, int end)
                                                 ▷ Does binary search for the value i of the given prefix sum field. Returns the index of the leftmost block in blocks[start..end] whose field f is ≥ i.
end BSEARCH

<Block, int, int> CREATEBLOCK(int i)
                                                 ▷ Creates a block to be inserted into as ith block in blocks. Returns the created block as well as values read from each child's num_propagated field. These values are used for incrementing the children's num_propagated field if the block was appended to blocks successfully.
    block newBlock= NEW(block)
    newBlock.group= num_propagated
    newBlock.order= i
    for each dir in {left, right} do
lastLine    index_last= dir.size
prevLine    index_prev= blocks[i-1].end_dir
        newBlock.end_dir= index_last
        block_last= dir.blocks[index_last]
        block_prev= dir.blocks[index_prev]
                                                 ▷ newBlock includes dir.blocks[index_prev+1..index_last].
        this_dir= dir.num_propagated
        newBlock.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq - block_prev.sum_enq
        newBlock.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq - block_prev.sum_deq
    end for
    if this is root then
        newBlock.length= max(root.blocks[i-1].length + b.num_enq - b.num_deq, 0)
    end if
    return <b, np_left, np_right>
end CREATEBLOCK
```

Algorithm Node

```
    ⇝ Precondition: blocks[b].num_enq ≥ i
element GetEnq(int b, int i)
    if this is leaf then
        return blocks[b].element
    else if i ≤ blocks[b].num_enq-left then                                  ▷ i exists in the left child of this node
        subBlock= left.BSearch(sum_enq, i, blocks[b-1].end_left+1, blocks[b].end_left)    ▷ Search range of left child's subblocks of blocks[b].
        return left.Get(i-left.blocks[subBlock-1].sum_enq, subBlock)
    else
        i= i-blocks[b].num_enq-left
        subBlock= right.BSearch(sum_enq, i, blocks[b-1].end_right+1, blocks[b].end_right)  ▷ Search range of right child's subblocks of blocks[b].
        return right.Get(i-right.blocks[subBlock-1].sum_enq, subBlock)
    end if
end GetEnq
```

```
    ⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num_enq ≥ i.
<int, int> IndexDeq(int b, int i)      ▷ Returns the rank of ith dequeue in the bth block of the node, among the dequeues in the root.
    if this is root then
        return <b, i>
    else
        dir= (parent.left==n)?  left:  right                                  ▷ check if a left or a right child
        superBlock= parent.BSearch(sum_deq-dir, i, super[blocks[b].group]-p, super[blocks[b].group]+p)
                                         ▷ superblock's group has at most p difference with the value stored in super[].
        if dir is right then
            i+= blocks[superBlock].sum_deq-left                               ▷ consider the dequeues from the right child
        end if
        return this.parent.IndexDeq(superBlock, i)
    end if
end Index
```

puteSuper

Algorithm Root

```
Block FindMostRecentDone
    for leaf l in leaves do
        max= Max(l.maxOld, max)
    end for
    return max                                                               ▷ This snapshot suffies.
end FindMostRecentDone
```

Algorithm Leaf

```
601: void Append(block blk)                                   ▷ Append is only called by the owner of the leaf.
appendEnd  602:     size+=1
pendStart  603:     blk.group= size
604:     blocks[size]= blk
605:     parent.Propagate()
606: end Append

607: Object HelpDequeue()
608:     <b_deq, r_deq>= IndexDeq(leaf.size, 1)      ▷ r is the rank among the dequeues of the dequeue of the b_deq th block in the root containing.
609:     <b_enq, r_enq>= FindResponse(b_deq, r_deq)  ▷ r_enq is the rank of the enqueue whose element is the response to the dequeue in the block containing it and
deqRest    b_deq is the index of that block of it in the blocklist. If the response is null then r_deq is -1.
610:     if r_enq==-1 then
611:         output= null
612:         root.blocks[b_deq].num_finished.inc()                                  ▷ shared counter
613:         if root.blocks[b_deq].num_finished==root.blocks[b_deq].num then
614:             last_done= b_deq
615:         end if
616:     else
617:         output= GetEnq(b_enq, r_enq)                                           ▷ getting the reponse's element.
618:         root.blocks[b_enq].num_finished.inc()
619:         root.blocks[b_enq].num_finished.inc()
620:         if root.blocks[b_deq].num_finished==root.blocks[b_deq].num then
621:             last_done= b_deq
622:         else if root.blocks[b_enq].num_finished==root.blocks[b_enq].num then
623:             last_done= b_enq
624:         end if
625:     end if
626:     return output
627: end Dequeue

628: void Help                                                                     ▷ Helps pending operations
629:     last= l.size-1        ▷ l.blocks[last] can not be null because size increases after appending, see lines 603-602.
630:     if l.blocks[last].element==null then                                      ▷ operation is dequeue
631:         l.blocks[last].response= l.HelpDequeue()
632:     end if
633: end Help
```

Algorithm BlockList

▷ : Supports two operations `blocks.tryAppend(Block b)`, `blocks[i]`. Initially empty, when `blocks.tryAppend(b, n)` returns true `b` is appended to `blocks[n]` and `blocks[i]` returns ith block in the `blocks`. If some instance of `blocks.tryAppend(b, n)` returns `false` there is a concurrent instance of `blocks.tryAppend(b′, n)` which has returned `true`.`blocks[0]` contains an empty block with all fields equal to 0 and $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$ pointers to the first block of the corresponding children.

◇ *root implementation*

boolean TryAppend(*block* blk, *int* n) ▷ adds block b to the root.blocks[n]
    if root.size%p^2==0 **then** ▷ Help every often p^2 operations appended to the root.
        for leaf l **in** tree leaves **do**
            l.Help()
        end for
    end if
    $\texttt{blk.num}_{\texttt{finished}}$= 0
    return CAS(blocks[n], null, blk)
end TryAppend

◇ *Array implementation*

blocks[] : array of blocks

boolean TryAppend(*block* blk, *int* n)
    return CAS(blocks[n], null, blk)
end TryAppend

Algorithm Yet to decide how to handle.

response FallBack(op i) ▷ *how to use as exception handling? by adding try catch in all the methods reading the root?*
    if root.blocks.get(num_{enq}), i is null **then** ▷ this enqueue was already finished
        return this.leaf.response(block.order)
    end if
end FallBack

2 Proof of Linearizability

$\boxed{\text{TEST}}$ As a temporary test I have changed the name of n.size to n.established here, other options are n.head and n.lastBLock but they might be confusing since we have used them before. Fix the logical order of definitions (cyclic refrences).

Definition 1 (Block)**.** A block is an object that stores some statistics described in Algorithm Queue. It implicitly shows a set of operations. The set of operations of block b are the operations in the leaf subblocks of b. We show the set of operations of block b, set of blocks B by $ops(b)$, $ops(B)$. We also say b contains op if $op \in ops(b)$.

Definition 2 (Order)**.** If `n.blocks[i]==b` we call `i` the *index* of block `b`. Block `b` is before block `b`′ in node `n` if and only if `b`'s index is smaller than `b`′'s.

Definition 3 (Subblock)**.** Block `b` is a subblock of `n.blocks[i]` if it is $\in$ `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` $\cup$ `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]` or is a subblock of a block in it.

For simplicity we say block `b` is propagated to node `n` or set `S` if `b` is in `n.blocks` or `S` or is a subblock of a block in `n.blocks` or `S`.

Definition 4. Block `b` in `n.blocks` is *Established* at time t if `n.established` is greater than index of `b` at time t. Block `b` in `n.blocks` is in $EST_{n,\ t}$ if `b` is a subblock of `b`′ in `n.blocks` such that `b`′ is established at time t.

`eProgress` **Lemma 5** (headProgress)**.** `n.established` *is non-decreasing over time.*

`dPosition` **Lemma 6** (headPosition)**.** *The value read in Line* prevLine *333(*`h=n.established`*) might be 1 bit behind the first empty block in the node.*

Proof. Because at the end of every `Refresh()` with block size greater than 0 (Lines 53,56) `n.head` is incremented. Maybe some process goes to sleep before incrementing the head, but after sleeping if `h` does not increase then `CAS` in Line 52 is going to be failed and nothing is going to be appended to `n.blocks`. □

`shedOrder` **Lemma 7** (establishedOrder)**.** *If time $t <$ time t', then* $\texttt{ops(EST}_{\texttt{n, t}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}'}\texttt{)}$.

Proof. Because blocks are only appended(not modified) with CAS to `n.blocks[n.head]` and `n.head` is non-decreasing. □

`eateBlock` **Lemma 8** (createBlock)**.** *If* `b` *is the block returned by* `n.CreateBlock(h, x)` *invoked at time* t*, then* $\texttt{ops(EST}_{\texttt{n.left, t}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}}\texttt{)}$ - $\texttt{ops(EST}_{\texttt{n, t}}\texttt{)} \subseteq \texttt{ops(b)}$.

Proof. We prove the claim for the left child. Blocks in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` are all the new established operations at time t by definition of Subblock. Line 70 is after t and since the head is only increasing (Lemma lem::headProgress ??) the lemma holds. See Figure fig::createBlock ??. The right child is the same. □

`ueRefresh` **Lemma 9** (trueRefresh)**.** *Let t_i be the time* `n.Refresh()` *is invoked and t_t be the time it is terminated. Suppose* `n.Refresh()`*'s* `TryAppend(new, s)` *returns* `true`*, then* $\texttt{ops(EST}_{\texttt{n.left, t}_i}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_i}\texttt{)} \in \texttt{ops(EST}_{\texttt{n, t}_t}\texttt{)}$.

Proof. By Lemma lem::createBlock 8 `new` contains `n`'s childrens' established blocks before Line 43 which is appended to `n.blocks` by `CAS` in Line 48. □

`seRefresh` **Lemma 10** (falseRefresh)**.** *If instance* `r` *of* `n.Refresh()` *reads value* `s` *on line* readSize *310 and then returns* `false`*, then there is another instance r' of* `n.Refresh()` *that has performed a successful* `TryAppend(new, s)`*. A* `TryAppend()` *is successful if its* `CAS` *is successful.*

Proof. If there is no other concurrent successful `Refresh(n)` then `Refresh(n)` would succeed in Line 48. So there is another `Refresh(n)`, that has to `CAS`ed successfully its block in `n.blocks[h]` after Line 43 of `Refresh(n)`. Otherwise the other `Refresh(n)` should have read `h'>h` instead of `h` for `n.head`(Line 52). □

`leRefresh` **Lemma 11** (Double Refresh)**.** *Consider two consecutive failed instances r_1,r_2 of* `n.Refresh()` *by some process. Let t_1 be the time r_1 is invoked and t_2 be the time r_2 terminated. After r_2's* `TryAppend` *we have* $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_2}\texttt{)}$.

Proof.

If Line 35 (first Refresh which we call R_1) returns `true`, the claim is held by Lemma 9. If not, then there is another successful instance of `Refresh()` R_1' by Lemma 10. R_1' may or may not have propagated some subblocks of `new` to `n`. It is obvious that the `new` constructed by the second Refresh in Line 36 contains the blocks in `new` by R_1 which R_1' did not contain, since `n.head` is only increasing (Maybe Lemma ??). If R_2 succeeds by Lemma 9 the claim holds. If not, it is deduced that `n.blocks[h]` was not `null` before R_2's `CAS`. Furthermore, `n.blocks[h]` was `null` before reading `h` by R_1. So there is a successful `Refresh()` after the read of `h` in R_1 and before the `CAS` of R_2. This `Refresh()` contains all the new established operations before Line 35 (Maybe Lemma ??) and by Lemma 9 our claim holds. See Figure ??. □

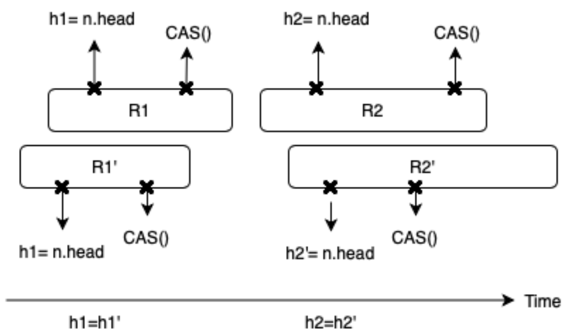

Figure 1: R_2''s `CAS` is executed after `h1=n.head`.

lyRefresh **Corollary 12** (Propagate Step). *All operations in* `n`*'s children's established blocks before line 302 are guaranteed to be in* `n`*'s blocks after line 303.*

Proof. Suppose block `b` with index `i` is in the the left child of `n` before the line 35. By Lemma 23 it follows that `n.left.head` is greater than `i`. `Refresh()` calls `CreateBlock()` and creates a block from blocks between `n.blocks[n.head].end`$_{\texttt{left}}$ and `n.left.head` in the left child, which contains `b` as well. First it tries to append it in `n.blocks.head` and if it was succsuful it continues recursively. If not it tries again, and if the second call of `Refresh()` in Line 36 fails. It means there is another `Refresh` which has reah its `i` after the Line 35, so it contains `b` as well. □

`CreateBlock()` reads blocks in the children that do not exist in the parent and aggregates them 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 `Refresh`es 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 a 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.

append **Lemma 13** (No Duplicates). *If* `op` *is appended to* `n.blocks[i]` *then after that there is no* `j>i` *such that* `op`$\in$`ops(n.blocks[j])`.

Proof. `Append(op)` adds `op` to `l`$_{\texttt{p}}$`.blocks`(Line ??) and `Propagate()` recursively propagates `op` up to the root. By lemma 12 we know that operation `op` propagates from child to parent at each level. □

Corollary 14. *After* `Append(blk)` *finishes* `ops(blk)`$\subseteq$`ops(root.blocks[x])` *for some* `x` *and only one* `x`.

blockSize **Lemma 15** (Block Size Upper Bound). *Each block contains at most one operation from each processs* (∀ `process p:#operations of p`$\in$`ops(n.blocks[x]`$\leq$`1)`.

ocksBound **Lemma 16** (Subblocks Upperbound). *Each block has at most p subblocks.*

ordering **Definition 17** (Ordering of operations inside the nodes)**.** ▶ Note that from Lemma blockSize 15 we know there is at most one operation from each process in a given block.

- We call operations before *op* in the sequence of operations S, prefix of the *op*.
- $E(n, i)$ is the sequence of enqueue operations $\in$ `ops(n.blocks[i])` ordered by their process id.
- $D(n, i)$ is the sequence of dequeue operations $\in$ `ops(n.blocks[i])` ordered by their process id.
- Order of the enqueue operations in n: $E(n) = E(n,1).E(n,2).E(n,3)...$
- Order of the dequeue operations in n: $D(n) = D(n,1).D(n,2).D(n,3)...$
- Linearization: $L = E(root,1).D(root,1).E(root,2).D(root,2).E(root,3).D(root,3)...$

Note that in the non-root nodes we only order enqueues and dequeues among the operations of their own type. Since `GetENQ()` *only works on enqueues and* `IndexDEQ()` *works on deqeueus.*

Theorem 18. *The queue implementation is linearizable.*

get **Lemma 19** (Get corretness)**.** `n.GetENQ(b,i)` *returns the ith Enqueue in* $E(n, b)$*.*

Lemma 20 (Index correctness)**.** `n.Index(b,i)` *returns the rank in the* $D(root)$ *of ith Dequeue in* $D(n, b)$*.*

uperBlock **Lemma 21** (Computing SuperBlock)**.** *After computing line* computeSuper *418 of* `n.IndexDEQ(b,i)`, `superblock` *contains the* `i`*th dequeue in the* `b`*th block of the node* `n`*.*

mputeHead **Lemma 22** (Computing Queue's Head)**.** *Let S be the state of an empty queue if the operations in prefix of ith dequeue in* $D(root, b)$ *are applied on it.* `FindResponse()` *returns the index of the enqueue that is the head in S. If the queue is empty in S it returns* `<-1,-->`*.*

head **Lemma 23** (Validity of `head`)**.** *If block b is written in* `n.blocks[i]` *then* `n.blocks[i]` *is going to remain b (not overwritten in future).*

erCounter **Lemma 24** (Validity of `super` and `counter`)**.** *If* `super[i]` $\neq$ `null`*, then* `super[i]` *in node* `n` *is the index of the superblock of a block with* `time=i±p`*.*

search **Lemma 25** (Search Ranges)**.** *Preconditions of all invocation of* `BSearch` *are satisfied.*

help **Lemma 26** (help)**.** *After that* `TryAppend()` *who is helping finishes, prefix for the blocks of* `root.blocks[root.FindMostRecentDone` *are done.*

rootRange **Lemma 27** (Root search range)**.** `root.size-root.FindMostRecentDone()` *is* $O(p^2 + q)$*, which p is # processes and q is the length of the queue.*

TODO Fallback safety lemmas.