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$\geq$0 and x $\in$ {enq, deq})
- $\texttt{blocks[b].sum=blocks[b].sum}_{\texttt{enq}}\texttt{+blocks[b].sum}_{\texttt{deq}}$ (for b$\geq$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 $\in$ {$\emptyset$, enq, deq, enq-left, enq-right, deq-left, deq-right}, $\texttt{blocks[0].num}_{\texttt{x}}$=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(s)    ▷ np_left, np_right are the values read from the children's num_propagated field.
    if new.num==0 then return true              ▷ The block contains nothing.
cas 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
incrementHead1  CAS(size, s, s+1)
        return true
    else
        CAS(size, s, s+1)                       ▷ Even if another process wins, help to increase the size. The winner might have fallen sleep before increasing size.
incrementHead2
        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
puteSuper
        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
```

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.
602:     size+=1                                                                         [appendEnd]
603:     blk.group= size                                                               [appendStart]
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_deqth 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
     b_deq is the index of that block of it in the blocklist. If the response is null then r_deq is -1.      [deqRest]
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

TEST As a temporary test I have changed the name of n.size to n.head 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).

TODO Fallback safety lemmas. Some parts are obsolete.

Questions When I write the lemmas since every claim in my mind is correct maybe I miss some fact that need proof or maybe I refer to some lemmas that are generally correct but not needed for the linearizability proof. Is lemma 7 necessary? Is lemma 13 induced trivially from lemma 8?

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

TEST How to merge notions of blocks and operations? block b $\sqsubseteq$ block c means b is subblock of c. block b $\in$ set B means b is in B. Merge these two to have shorter formulaes.

Definition 1 (Block)**.** A block is an object storing 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 *direct subblock* of `n.blocks[i]` if it is $\in$ `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`$_{right}$`]`. Block `b` is a subblock of a `n.blocks[i]` if it is a direct subblock of it or subblock of a direct subblock of it. Block b is direct superblock of block c if c is direct subblock of b.

For simplicity we say block `b` is propagated to node `n` or to a set of blocks `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.head` is greater than index of `b` at time t. Block `b` is in $EST_{n,\ t}$ if `b` is a subblock of `b′` in `n.blocks` such that `b′` is established at time t.

head **Observation 5.** *Once block b is written in* `n.blocks[i]` *then* `n.blocks[i]` *never changes.*

eProgress **Lemma 6** (headProgress)**.** `n.head` *is non-decreasing over time and* `n.blocks[i].end`$_{left,\ right}$ $\geq$ `n.blocks[i-1].end`$_{left,\ right}$.

Proof. Induced trivially from the pseudocode since `n.head` is only incremented. `n.blocks[i]. Also end`$_{left,\ right}$ are greater than or equal the previous values in the `CreateBlock(i)` code (Lines lastLine, prevLine ??). ☐

Lemma 7. *Every block has most one direct superblock.*

Proof. To show this we are going to refer to the way n.blocks[] is partitioned while propagating 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 ??). From we know that `end`$_{left}$`,end`$_{right}$ are non decreasing, so the range of the subblocks of n.blocks are disjoint. ☐

append **Corollary 8** (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])`.

dPosition **Invariant 9** (headPosition)**.** If the value of `n.head` is `h` then, `n.blocks[i]=null` for `i>h` and `n.blocks[i]`$\neq$`null` for `i<h`.

Proof. Some value is written into `n.blocks[head]` only in Line 313. It is obvious that writing into `n.blocks[head]` does not change the state of the claim of the lemma. The value of `n.head` is modified only in lines incrementHead incrementHead2 318, ||p321, we show in both cases of the successful cas 313 or not the claim holds after the increment lines of `n.head`. If h increments to h it is sufficient to show `n.blocks[h]`$\neq$`null` to claim the invariant still holds. In the first case the process applied a successful `TryAppend(new,h)` in line okcas 314, which means `n.blocks[h]` is not null anymore. Note that wether incrementHead1 318 returns true or false after Line `n.head` we know has been incremented from Line readSize 310. The failure case is also the same since it means some value is written into `n.blocks[head]` by some process. ☐

Lemma 10 (establishedOrder)**.** *If time $t <$ time t', then* $\texttt{ops(EST}_{\texttt{n, t}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t'}}\texttt{)}$.

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 grows. □

Lemma 11 (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* $\texttt{ops(EST}_{\texttt{n.left, t}_i}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_i}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_t}\texttt{)}$.

Proof. From Lemma 10 we know that $\texttt{ops(EST}_{\texttt{n, t}_i}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_t}\texttt{)}$. So it remains to show $\texttt{ops(EST}_{\texttt{n.left, t}_i}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_i}\texttt{)}$ - $\texttt{ops(EST}_{\texttt{n, t}_i}\texttt{)}$ which we call new operations $\subseteq \texttt{ops(EST}_{\texttt{n, t}_t}\texttt{)}$. If `TryAppend`returns true a block is appended to n. From the code of the `CreatBlock` this block includes the establsihed blocks in n's children at t_i. Since the head in createblock is read after t_i. So the new operations are inthe block appended to n.

□

Lemma 12 (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 313 of R_1 or R_2 returns `true`, then the claim is held by Lemma 11. If not, then there is another successful instance of `n.Refresh()` R_2' which has did TryAppend successfully into `n.blocks[i+1]`. Note that if R_2 reads some value greater than $i+1$ in Line 310 it means there is a succesful `Refresh()` instance started after Line 310 of R_1 and finished its Line 318 or 321 before 310 of R_2, form Lemma 11 by finish of this instance $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)}$ has been propagated. In the figure 1 we see why the block R_2' is appending contains established block in the n's children at t_i, since it create a block reading the head after t_1. □

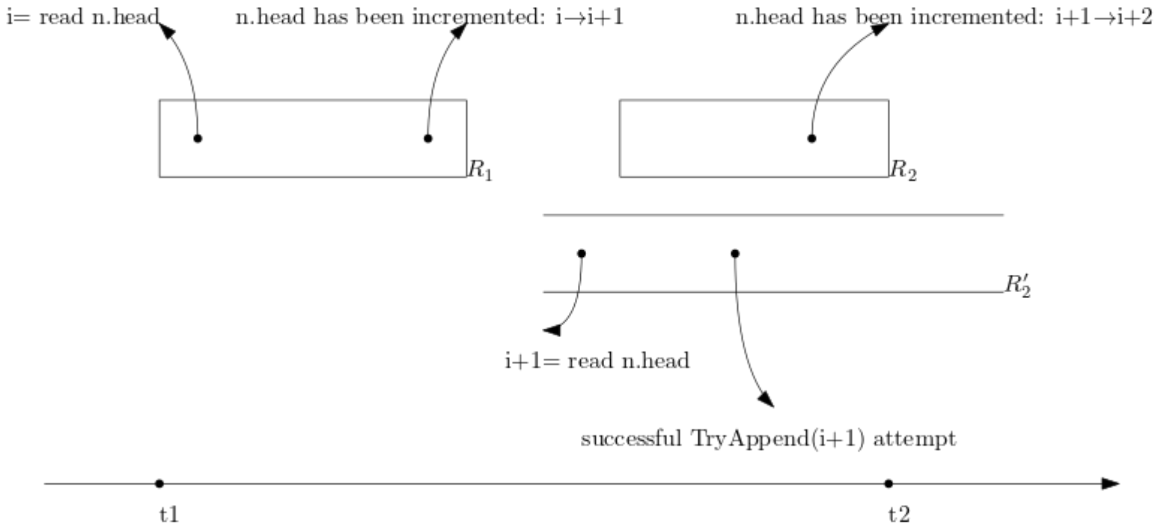

Figure 1: $t1 < r_1$ reading head $<$ incrementing `n.head` from i to $i+1 < R_2'$ reading head $<$ `TryAppend(i+1)` $<$ incrementing `n.head` from $i+1$ to $i+2 < t2$

Corollary 13 (Propagate Step)**.** *All operations in* `n`*'s children's established blocks before line 302 are guaranteed to be in* `n`*'s established blocks after line 303.*

Proof. Lines 302 and 303 satisfy the preconditions of Lemma 12. □

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

Proof. Follows from Lemma 12, 8. □

`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 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 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.

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

Proof. Assume there are two operations op_1, op_2 from process p in block b. A process cannot invoke two operations concurrently. WLOG assume `leaf of p.Append(block containing op`$_1$`)` has to be finished before `leaf of p.Append(block containing op`$_2$`)` starts. So before appending op_2 to the tree op_1 exists in every node. So there is a node that has two blocks containing op_1 since if not then b cannot exist. This contradicts with 8. □

Is something wrong with this proof? It is more complicated than I thought.

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

Proof. It follows directly from Lemma 15 and the observation that each block contains at least one operation, induced from Line ??. □

ordering **Definition 17** (Ordering of operations inside the nodes)**.** ▶ Note that from Lemma 15 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 ∈ `ops(n.blocks[b])` ordered by their process id.
- $E(n, b, i)$ is the ith enqueue in $E(n, b)$.
- $D(n, b)$ is the sequence of dequeue operations ∈ `ops(n.blocks[b])` ordered by their process id.
- $D(n, b, i)$ is the ith enqueue in $D(n, b)$.
- 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 searches among enqueues and* `IndexDEQ()` *works on dequeues.*

get **Lemma 18** (Get correctness)**.** `n.GetENQ(b,i)` *returns the ith Enqueue in* $E(n, b)$.

Proof. It is easy to see that `leaf.GetENQ(b,1)` returns the operation stored in the bth block of leaf l. In Line 404 it is decided that the `i`th enqueue in block `b` of internal node `n`resides in the left child or the right child of `n`. Then `n.child.GetENQ( block containing, order in the block)` is invoked. This recursive procedure ends in a leaf node. After that Lines 88, 92 search the proper subblocks of `b`. The correctness of the invocation parameters is induced from the definition of ops(block), subblock. □

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

Definition 19. An enqueue operation is *finished* if its argument is returned by some process. A dequeue operation is `finished` if it returns `null` or some value. Block `b` is *done* if all operations in `ops(b)` are finished.

Problem: we increment the $\texttt{num}_{\texttt{finished}}$ *before returning and after the computing response. How to articulate the sentence above in a not confusing correct way?*

help **Lemma 20** (help)**.** *After that* `TryAppend()` *who is helping finishes, prefix for the blocks of* `root.blocks[root.FindMostRecentDone]` *are done.*

uperBlock **Lemma 21** (Computing SuperBlock)**.** *After computing line* computeSuper *418 of* `n.IndexDEQ(b,i)`, `n.parent.blocks[superblock]` *contains* $D(n, b, i)$.

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

▶ Values $\mathtt{c_{dir}}$ read in lines 23, `super` are set before incrementing in lines 26,27.

2. `super[]` preserves order from child to parent; if in a child block `b` is before `c` then `b.group` $\leq$ `c.group` and `super[b.group]` $\leq$ `super[c.group]`

▶ Follows from the order of lines 60, 48, 49.

3. `super[i+1]-super[i]`$\leq p$

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

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

▶ `super[i]` is the index of the superblock of a block containing block b, followed by Lemma superCounter 25. It is trivial to see that `n.super` and `n.b.counter` are increasing. `super(b)` is the real superblock of b. `super(t]` is the index of the superblock of the last block with time `t`. If `b.time` is `t` we have:

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

□

Lemma 22 (Index correctness)**.** `n.IndexDEQ(b,i)` *returns the rank in* $D(root)$ *of* $D(n, b, i)$.

Proof. We can see the base case `root.IndexDEQ(b,i)` is trivial. `n.IndexDEQ(b,i)` computes the superblock of the ith Dequeue in the bth block of `n` in `n.parent`(Lemma superBlock 21). Then the order in $D(n.parent, superblock)$ is computed and `index()` is called on `n.parent` recursively. It is easy to see why the second is correct. Correctness of computing superblock comes from Lemma superBlock 21. □

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?

search **Lemma 23** (Search Ranges)**.** *Preconditions of all invocation of* `BSearch` *are satisfied.*

Proof. Line 83: `Get(i)` is called if the result of a dequeue is not null. The search is among all blocks in the root.

Line 88: This search tries to find the ith enqueue, knowing that it is in the left child. Search is done over the left subblocks. The start and end of the range are followed by definition. Line 92 is the same.

Line 101: Here, the goal is to find the superblock. We know the distance between answer and the `super[i]` is at most p, since at most p processes could die.

□

mputeHead **Lemma 24** (Computing Queue's Head)**.** *Let* S *be the state of an empty queue if the operations in prefix in* L *of* i*th dequeue in* $D(root, b)$ *are applied on it.* `FindResponse()` *returns the index in* $E(root, b)$ *of the enqueue that is the head in* S*. If the queue is empty in* S *it returns* `<-1,-->`.

erCounter **Lemma 25** (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`$\pm$`p`.

rootRange **Lemma 26** (Root search range). `root.size-root.FindMostRecentDone()` *is* $O(p^2 + q)$*, which* p *is # processes and* q *is the length of the queue.*

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

Lemma 28 (Time analysis). `n.GetEnq(b,i)`, `n.Index(b,i)` *take* $O(\log^2 p)$ *steps. Search in the root may take* $O(\log Q + p^2)$ *steps. Helping is done every* p^2 *block appended to the root and takes* $p \times \log^2 p$ *steps. Amortized time consumed for helping by each process is* $O(\log^2 p)$.