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(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.
                  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
incrementHead2       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
nputeSuper
        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.

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 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
    blk.num_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 $\texttt{ops(EST}_{\texttt{n, t}}\texttt{)}$ with $\texttt{EST}_{\texttt{n, t}}$?

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 $\in$ $\texttt{n.left.blocks[n.blocks[i-1].end}_{\texttt{left}}\texttt{+1..n.blocks[i].end}_{\texttt{left}}\texttt{]}$ $\cup$ $\texttt{n.right.blocks[n.blocks[i-1].end}_{\texttt{right}}\texttt{+1..n.blocks[i].end}_{\texttt{right}}\texttt{]}$. 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** (Validity of `head`)**.** *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* $\texttt{n.blocks[i].end}_{\texttt{left, right}}$ *is non-decreasing over* `i`*.*

Proof. Induced trivially from the pseudocode since `n.head` is only incremented. `n.blocks[i].` Also $\texttt{end}_{\texttt{left, right}}$ 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 $\texttt{n.left.blocks[n.blocks[i-1].end}_{\texttt{left}}\texttt{..n.blocks[i].end}_{\texttt{left}}\texttt{]}$ and $\texttt{n.right.blocks[n.blocks[i-1].end}_{\texttt{right}}\texttt{..n.blocks[i].end}_{\texttt{righ}}$ (Lines lastLine, pr ??). From we know that $\texttt{end}_{\texttt{left}}\texttt{,end}_{\texttt{right}}$ are non decreasing, so the range of the subblocks of n.blocks are disjoint. □

dPosition **Lemma 8** (headPosition)**.** *If the value read in Line* prevLine *333(*`h=n.head`*) is h then,* `n.blocks[i]=null` *for* `i>h` *and* `n.blocks[i]`$\neq$`null` *for* `i<h`*.*

Proof. At the end of every `n.Refresh()` with a block with size greater than 0 returned by `CreateBlock()` `n.head` is incremented (Lines incrementHead1incrementHead2 318, 321). If a process went to sleep before incrementing the `established` (line incrementHead1 318), nothing can be appended to `n.blocks` until another process increments `n.establsihed` (Line incrementHead2 321). □

shedOrder **Lemma 9** (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. □

ueRefresh **Lemma 10** (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 9 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.

□

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 313 of r_1 or r_2 returns `true`, then the claim is held by Lemma 10. If not, then there is another successful instance of `n.Refresh()` r_2' which has did TryAppend successfully into `n.blocks[i+1]`. In figure 1 we see why the block it is appending contains established block in the n's children at t_i, since it create a block reading the head after t_1. □

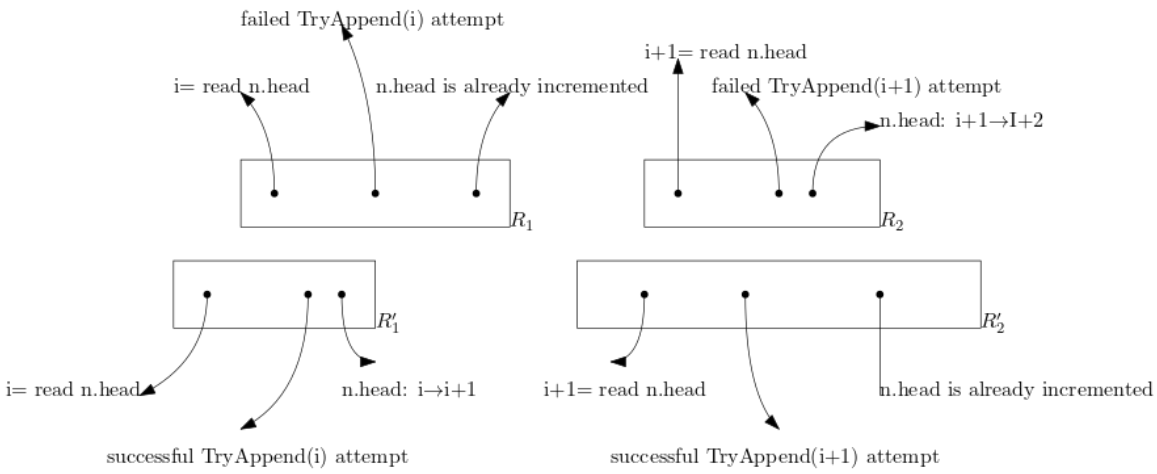

Figure 1: t_t is after R_2''s reads n.head after n.head is incremented which is after the succesufil TryAppend(i) which is after R_1's reading n.head that takes place at t_i.

lyRefresh **Corollary 12** (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 11. □

`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])`.

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∈ops(n.blocks[x]≤1)`.

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

search **Lemma 18** (Search Ranges)**.** *Preconditions of all invocation of* `BSearch` *are satisfied.*

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

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

Lemma 21 (Index correctness)**.** `n.Index(b,i)` *returns the rank in* $D(root)$ *of the ith Dequeue in* $D(n, b)$.

uperBlock **Lemma 22** (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 23** (Computing Queue's Head)**.** *Let S be the state of an empty queue if the operations in prefix in L of ith 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 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`.

rootRange **Lemma 25** (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 26 (Main)**.** *The queue implementation is linearizable.*

Lemma 27 (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 $O(\log^2 p)$.*