# 1 Pseudocode

**Algorithm** Tree Fields Description

◇ *Shared*

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

◇ *Local*

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

◇ *Structures*

▶ *Node*

- **Node* `left, right, parent` : initialized when creating the tree.
- *BlockList* `blocks` implemented with an array.
- *int* `head`= 1: #`blocks` in `blocks`(-1). `blocks[0]` is a block with all fields equal to zero.
- *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`.

▶ *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. If the operation is `enqueue(x)` then `element=x`, otherwise `element=null`.
- *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* `size` : size of the queue after performing all operations in the prefix for this block

*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.head].sum_enq+1
    newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq
    leaf.head+=1
    leaf.Append(newBlock)
end Enqueue

<int, int> FindResponse(int b_d, int i_d)    ▷ If E(root, b_e, i_e) is the response to the D(root, b_d, i_d) returns <b_e,i_e>. Returns <-1,--> if the queue is empty.
    if root.blocks[b_d-1].size + root.blocks[b_d].num_enq - (i + root.blocks[b_d-1].sum_deq) < 0 then
        return <-1,-->
    else
        r_enq= i + root.blocks[b_d-1].sum_deq - (root.blocks[b_d-1].size - root.blocks[b_d-1].sum_enq + root.blocks[b_d-1].sum_deq)
                                        ▷ size-enqs+deqs=null deqs
        return root.DSearch(r_enq, b_d)
    end if
end FindResponse

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.head].sum_enq
    newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq+1
    leaf.Append(newBlock)
    <b_deq, r_deq>= IndexDeq(leaf.head, 1)    ▷ r_deq is the rank among the dequeues of the dequeue of the b_deq th block in the root containing.
    <b_enq, r_enq>= FindResponse(b_deq, r_deq)    ▷ E(root, b_enq, i_enq) is response to the D(root, b_deq, i_deq). If the response is null then
deqRest    r_enq is -1.
    if r_enq==-1 then
        output= null
    else
        output= GetEnq(b_enq, r_enq)    ▷ getting the reponse's element.
    end if
    return output
end Dequeue
```

**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()
readHead
    s= head
    <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
ncrementHead1
        CAS(head, s, s+1)
        return true
    else
        CAS(head, s, s+1)             ▷ Even if another process wins, help
            to increase the head. The winner might have fallen sleep before increasing
ncrementHead2
            head.
        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

⇝ Precondition: root.blocks[end].sum_enq ≥ r_enq
<int, int> DSearch(int i, int end)
            ▷ Searches for the ith enqueue of the given prefix sum of bth block
    in the root. Returns the index of the leftmost block in root.blocks whose
    sum_enq is ≥ i.
    start= b-1
    while root.blocks[start].sum_enq≥r_enq do
        start= start-(b-start)
    end while
    return root.BSearch(sum_enq, r_enq, start, b)
end DSearch

<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.head
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].
        np_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.size= max(root.blocks[i-1].size+ b.num_enq -
    b.num_deq, 0)
computeLength
    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** Leaf

```
void Append(block blk)                                    ▷ Append is only called by the owner of the leaf.
appendEnd
    head+=1
pendStart
    blk.group= head
    blocks[head]= blk
    parent.Propagate()
end Append
```

**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 $i$th 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 $\mathrm{end}_{\mathrm{left}}$, $\mathrm{end}_{\mathrm{right}}$ pointers to the first block of the corresponding children.

```
blocks[]:   array of blocks
boolean TryAppend(block blk, int n)
    return CAS(blocks[n], null, blk)
end TryAppend
```

## 2 Proof of Linearizability

TEST As a temporary test I have changed the name of n.headto 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, as described in Algorithm Queue. It implicitly represents a set of operations. 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 2** (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. The set of operations of block $b$ are the operations in the leaf sub blocks of $b$. We show the set of operations of block by `ops(b)`. We also say $b$ contains $op$ if $op \in ops(b)$.

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 3.** A block `b` in `n.blocks` is *established* at time $t$ if `n.head` is greater than the index of `b` in `n.blocks` 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 4.** *Once a block* `b` *is written in* `n.blocks[i]` *then* `n.blocks[i]` *never changes.*

dProgress **Lemma 5** (headProgress)**.** `n.head` *is non-decreasing over time and* `n.blocks[i].end`$_{left}$ $\geq$ `n.blocks[i-1].end`$_{left}$,`n.blocks[i].end`$_{right}$ $\geq$ `n.blocks[i-1].end`$_{right}$.

*Proof.* The first claim follows trivially from the pseudocode since `n.head` is only incremented. Also when `n.blocks[i]` is created its `end`$_{left}$`,` `end`$_{right}$ are greater than or equal to the values in `n.blocks[i-1]`. Since `blocks[i-1].end`$_{dir}$`<dir.head= blocks[i].end`$_{dir}$ (Lines lastLine, prevLine ??). $\square$

**Lemma 6.** *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 blocks up to `n.parent`. `n.CreateBlock(i)` merges the blocks in `n.left.blocks[n.blocks[i-1].end`$_{left}$`..n.blocks[i].end`$_{left}$`]` and `n.right.blocks[n.blocks[i-1].end`$_{right}$`..n.blocks[i]` (Lines lastLine, pr ??). Since `end`$_{left}$`,end`$_{right}$ are non-decreasing, so the range of the subblocks of `n.blocks[i]` which is `(n.blocks[i-1].end`$_{dir}$`+1..n.blocks[i]` does not overlap with the range of the subblocks of `n.blocks[i-1]`. $\square$

append **Corollary 7** (No Duplicates)**.** *If* `op` *is in* `n.blocks[i]` *then there is no* `j`$\neq$`i` *such that* `op`$\in$`ops(n.blocks[j])`.

dPosition **Invariant 8** (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.* The invariant is true initially since 1 is assigned to `n.head` and `n.blocks[x]` is null for every `x`. The truth of the invariant may be affected by writing into `n.blocks` or incrementing `n.head`.

Some value is written into `n.blocks[head]` only in Line 313. It is obvious that writing into `n.blocks[head]` presereves the invariant. The value of `n.head` is modified only in lines 318], 321. Depending on wether the `TryAppend()`in Line 313 succeeded or not we show that the claim holds after the increment lines of `n.head` in either case. If `head` is incremented to $h$ it is sufficient to show `n.blocks[h]`$\neq$`null` to prove the invariant still holds. In the first case the process applied a successful `TryAppend(new,h)` in line 314, which means `n.blocks[h]` is not null anymore. Note that wether 318 returns true or false after Line `n.head` we know has been incremented from Line 310. The failure case is also the same since it means some value is written into `n.blocks[head]` by some process. ☐

*Explain More*

**Lemma 9** (establishedOrder). *If time $t <$ time $t'$, then* `ops(EST`$_{\texttt{n, t}}$`)`$\subseteq$ `ops(EST`$_{\texttt{n, t}'}$`)`.

*Proof.* Blocks are only appended (not modified) with CAS to `n.blocks[n.head]` and `n.head` is non-decreasing, so the set of operations in established blocks of a node can only grows. ☐

`CreateBlock()` aggregates the blocks in the children that are not already established in the parent into one block. If a `Refresh()` procedure returns true it means it has appended the block created by `CreateBlock()` into the parent node's sequence. So suppose two `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 another successful `Refresh()` that assuredly has read block before the mentioned line `35`. After all it means if any of the `Refresh()` attempts were successful the claim is true, and also if both fail the mentioned claim still holds.

**Lemma 10** (trueRefresh). *Let $t_i$ be the time a* `n.Refresh()` *operation is invoked and $t_t$ be the time it terminates. Suppose* `n.Refresh()`*'s* `TryAppend(new, s)` *returns* `true`*, then* `ops(EST`$_{\texttt{n.left, t}_i}$`)` $\cup$ `ops(EST`$_{\texttt{n.right, t}_i}$`)` $\subseteq$ `ops(EST`$_{\texttt{n, t}_t}$`)`.

*Proof.* From Lemma 9 we know that `ops(EST`$_{\texttt{n, t}_i}$`)`$\subseteq$`ops(EST`$_{\texttt{n, t}_t}$`)`. So it remains to show the operations of `ops(EST`$_{\texttt{n.left, t}_i}$`)` $\cup$ `ops(EST`$_{\texttt{n.right, t}_i}$`) - ops(EST`$_{\texttt{n, t}_i}$`)`, which we call new operations, are all in `ops(EST`$_{\texttt{n, t}_t}$`)`. If `TryAppend` returns true a block is appended to `n`. From the code of the `CreateBlock()` this block includes the established blocks in n's children at $t_i$. Since the head in `CreateBlock()` is read after $t_i$. So the new operations are in the block appended to $n$.

☐

*Mention this new block is establish in $t_t$. last sentece is not complete*

**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. W e have* `ops(EST`$_{\texttt{n.left, t}_1}$`)` $\cup$ `ops(EST`$_{\texttt{n.right, t}_1}$`)` $\subseteq$ `ops(EST`$_{\texttt{n, t}_2}$`)`.

*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 $R_2'$ of `n.Refresh()` that has done `TryAppend()` successfully into `n.blocks[i+1]`. If $R_2$ reads some value greater than $i+1$ in Line 310 it means a successful instance of `Refresh()` started after Line 310 of $R_1$ and finished its Line 318 or 321 before 310 of $R_2$, from Lemma 10 by the end of this instance `ops(EST`$_{\texttt{n.left, t}_1}$`)` $\cup$ `ops(EST`$_{\texttt{n.right, t}_1}$`)` has been propagated. In Figure 1 we see why the block $R_2'$ is appending contains established block in the n's children at $t_1$, since it create a block reading the head after $t_1$. ☐

*last sentence need more detail and should be earlier. define i and tell why R2prime exists*

*this chain with more depth should be in the proof*

**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. ☐

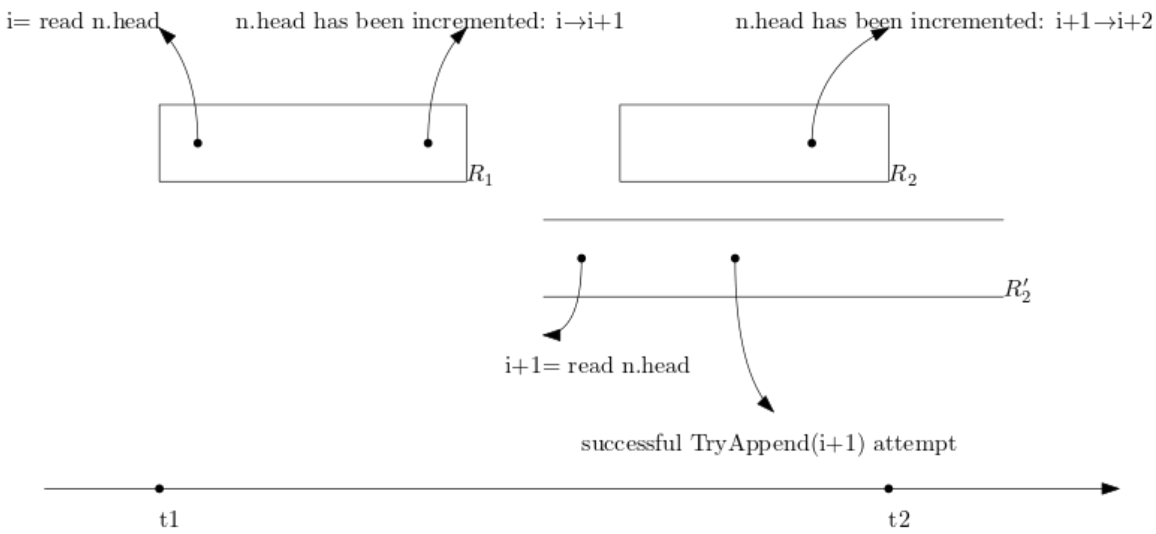


Figure 1: $t1 < r_1$ reading head $<$ incrementing `n.head` from $i$ to $i+1 < R_2'$ reading head $<$ `TryAppend(i+1)` $<$ incrementing `n.head` from $i+1$ to $i+2 < t2$

**Corollary 13.** *After* `Append(blk)` *finishes* `ops(blk)`⊆`ops(root.blocks[x])` *for some* `x` *and only one* `x`.

*Proof.* Follows from Lemma doubleRefresh append 11, 7. ☐

blockSize **Lemma 14** (Block Size Upper Bound)**.** *Each block contains at most one operation from each processs.*

*Proof.* By proof of contradiction, assume there are more than one operation from process $p$ in block $b$ in node $n$. A process cannot invoke more than one operations concurrently. From $p$ 's operations in $b$, let $op_1$ be the first operation invoked and $op_2$ be the second one. Note that it is terminated before $op_2$ started. So before appending $op_2$ to the tree $op_1$ exists in every node from the path of $p$'s leaf to the root. So there is some block $b'$ before $b$ in $n$ containing $op_1$. $op_1$ existing in $b$ an $b'$ contradicts with append 7. ☐

ocksBound **Lemma 15** (Subblocks Upperbound)**.** *Each block has at most $p$ direct subblocks.*

*Proof.* It follows directly from Lemma blockSize 14 and the observation that each block contains at least one operation, induced from Line addOP ??. ☐

ordering **Definition 16** (Ordering of operations inside the nodes)**.** ► Note that processes are numbered from 1 to $p$, left to right in the leaves of the tree and from Lemma blockSize 14 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 $i$th 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 $i$th enqueue in $D(n, b)$.
- Order of the enqueue operations in $n$: $E(n) = E(n, 1).E(n, 2).E(n, 3)...$
- $E(n, i)$ is the $i$th enqueue in $E(n)$.
- Order of the dequeue operations in $n$: $D(n) = D(n, 1).D(n, 2).D(n, 3)...$
- $D(n, i)$ is the $i$th dequeue in $D(n)$.
- 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 17** (Get correctness)**.** *If* `n.blocks[b].num`$_{\texttt{enq}}\geq$`i` *then* `n.GetENQ(b,i)` *returns* $E(n,b,i)$.

*Proof.* We are going to prove this lemma by induction on the height of the tree. The base case for the leaves of the tree is pretty straight forward. Since leaf blocks contain exactly one operation then only `GetENQ(b,1)` can be called on leaves. `leaf.GetENQ(b,1)` returns the operation stored in the $b$th block of leaf $l$. For non leaf nodes 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`. By definition of $E(n,b)$ operations from the left child come before the operations of the right child. Having `sum`$_{\texttt{enq}}$, the prefix sum of the number of enqueues we can compute the direct subblock containing the enqueue we are finding for with binary search. Then `n.child.GetENQ( block containing, order in the block)` is invoked which returns the correct operation by the hypothesis of the induction. ☐

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

dsearch **Lemma 18** (DSearch correctness)**.** *If* `root.blocks[end].num`$_{\texttt{enq}}\geq$`i` *and* $E(root,i)$ *is the response to some* `Dequeue()` *in* `root.blocks[end]` *then* `DSearch(i, end)` *returns* `b` *such that* `root.blocks[b]` *contains* $E(root,b,i)$ *in* $\Theta(\log($`root.blocks[b].size+root.blocks[end].size`$)$ *steps.*

*Proof.* First we show `end-b`$\leq$`root.blocks[b].size+root.blocks[end].size`. We know each block size is greater than 0. So every block in `root.blocks[b..end]` contains at least one `Enqueue()` or one `Dequeue()`. There cannot be more than `root.blocks[b].size` `Dequeue()`s in `root.blocks[b+1..end-1]`, since the queue would become empty after bth block end before end and $E(n,i)$ could not be the response to to some `DEQ` in `end`. And since the lentgh of the queue would become `root.blocks[end].size` in the end so there cannot be more than `root.blocks[end].size` Dequeus in `root.blocks[b..end]`. Cause if there was more then the end's lenght would become more than `root.blocks[end].size`.

Now that we know there are at most `root.blocks[b].size+root.blocks[end].size` distance between end and b then with doubling search in log`root.blocks[b].size+root.blocks[end].size` steps we reach a block `c` that the `c.sum`$_{\texttt{enq}}$ is less than i and the distance between c and end is not more than 2×`root.blocks[b].size+root.blocks[end].size`. So the binary search takes $\Theta($log`root.blocks[b].size+root.blocks[end].size`$))$ steps. ☐

uperBlock **Lemma 19** (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 `c`$_{\texttt{dir}}$ 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 23. 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:

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

□

We call the dequeues that return some value *non-null dequeues*. $r$th non-null dequeue returns the element of th $r$th enqueue. We can compute # non-null dequeues in the prefix for a block this way: #non-null dequeues= size- #enqueues. Note that the $i$th dequeue in the given block is not a non-null dequeue.

**Lemma 20** (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 $i$th Dequeue in the $b$th block of `n` in `n.parent`(Lemma superBlock 19). 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 19.size □

*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 21** (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 22** (Computing Queue's Head block). *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* `(b, i)` *which* $E(root, b, i)$ *is the the head of the queue in* $S$. *If the queue is empty in* $S$ *then it returns* `<-1,-->`.

*Proof.* The sizeof the queue if the operations in the prefix for the $b$th block in the root are applied with the order of $L$ is stored in the `root.blocks[b].size`. It is computed while creating the block in Line computeLength 350. If the sizeof a queue is greater than 0 then a `Dequeue()` would decrease the sizeof the queue, otherwise the sizeof the queue remains 0. Having sizeof the queue after the previous block and number of enqueues and dequeues in the block, Line computeLength 350 computes wether the queue becomes empty or the sizeof it. □

HOW? How to prove mathematicaly that `ax(root.blocks[i-1].size+` $\texttt{b.num}_{\texttt{enq}}$ `-` $\texttt{b.num}_{\texttt{deq}}$`, 0)` is the sizeof the queue after the block. I can only explain it here.

erCounter **Lemma 23** (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 24** (Root search range). `root.size-root.FindMostRecentDone()` *is* $O(p^2 + q)$, *which* $p$ *is # processes and* $q$ *is the sizeof the queue.*

**Theorem 25** (Main). *The queue implementation is linearizable.*

**Lemma 26** (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* $O(\log^2 p)$.