1 Pseudocode

Algorithm Tree Fields Description

◇ *Shared*

- A binary tree of Nodes with one leaf for each process. root is the root node.

◇ *Local*

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

◇ *Structures*

► *Node*

- **Node* left, right, parent : initialized when creating the tree.
- *BlockList*
- *int* head= 1: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.
- *int* $\text{num}_{\text{propagated}}$= 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.

► *Block*

- *int* group : the value read from $\text{num}_{\text{propagated}}$ 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* sum_{enq}, sum_{deq} : # enqueue, dequeue operations in the prefix for the block

► *InternalBlock* extends *Block*

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

► *RootBlock* extends *InternalBlock*

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

Abbreviations:

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

Algorithm *Queue*

```
void Enqueue(Object e)    ▷ Creates a block with element e and adds 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.Append(newBlock)
end Enqueue

Object Dequeue()    ▷ Creates a block with null value element, appends it to the tree, computes its order among operations, and returns its response.
    block newBlock= new(LeafBlock)
    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, i_deq>= IndexDeq(leaf.head, 1)
    <b_enq, i_enq>= FindResponse(b_deq, i_deq)        deqRest
    if i_enq==-1 then
        output= null
    else
        output= GetEnq(b_enq, i_enq)
    end if
    return output
end Dequeue

<int, int> FindResponse(int b_d, int i_d)    ▷ Returns <b_e,i_e>, if E_{root,b_e,i_e} is the response to the D_{root,b_d,i_d}. Returns <-1,--> if the queue is empty.
    if  root.blocks[b_d-1].size + root.blocks[b_d].num_enq - i_d < 0 then
        return <-1,-->
    else
        r_enq= i_d - root.blocks[b_d-1].size + root.blocks[b_d-1].sum_enq
        return root.DSearch(r_enq, b_d)
    end if
end FindResponse
```

Algorithm *Node*

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

boolean REFRESH()
readHead        h= head
makeCreateBlock <new, np_left, np_right>= CREATEBLOCK(h)    ▷ np_left, np_right are the values read from the children's num_propagated field.
addOP           if new.num==0 then return true    ▷ The block contains nothing.
cas             else if blocks.tryAppend(new, h) then
okcas               for each dir in {left, right} do
setSuper                CAS(dir.super[np_dir], null, h)    ▷ Write would work too.
incNP                   CAS(dir.num_propagated, np_dir, np_dir+1)
                    end for
incrementHead1      CAS(head, h, h+1)
                    return true
                else
                    CAS(head, h, h+1)    ▷ Even if another process wins, help to increase the head. The winner might have fallen sleep before increasing head.
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 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)
setGroup        newBlock.group= num_propagated
                for each dir in {left, right} do
lastLine            index_last= dir.head
prevLine            index_prev= blocks[i-1].end_dir
endDefLine          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].
setNP               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
computeLength       newBlock.size = max(root.blocks[i-1].size + newBlock.num_enq - newBlock.num_deq, 0)
                end if
                return <b, np_left, np_right>
end CREATEBLOCK
```

Algorithm Root

```
⇝ Precondition: root.blocks[end].sum_enq ≥ e ≥ 1
<int, int> DSEARCH(int e, int end)    ▷ Returns <b,i> if E_root,e = E_root,b,i.
    start= end-1
    while root.blocks[start].sum_enq≥e do
        start= max(start-(end-start), 0)
    end while
    b= root.BSearch(sum_enq, e, start, end)
    i= e- root.blocks[b-1].sum_enq
    return <b,i>
end DSEARCH
```

Algorithm Node

```
⇝ Precondition: blocks[b].num_enq≥i
element GetEnq(int b, int i)                                  ▷ Returns the element of E_{this,b,i}.
    if this is leaf then
        return blocks[b].element
    else if i ≤ blocks[b].num_enq-left then                   ▷ E_{this,b,i} is in the left child of this node.
        subBlock= left.BSearch(sum_enq, i+left.blocks[blocks[b].end_left-1].sum_enq, blocks[b-1].end_left+1, blocks[b].end_left)
        return left.GetEnq(subBlock, i)
    else
        i= i-blocks[b].num_enq-left
        subBlock= right.BSearch(sum_enq, i+right.blocks[blocks[b].end_right-1].sum_enq, blocks[b-1].end_right+1, blocks[b].end_right)
        return right.GetEnq(subBlock, i)
    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 <x, y> if D_{this,b,i} = D_{root,x,y}.
    if this is root then
        return <b, i>
    else
        dir= (parent.left==n)? left: right                    ▷ check if this node is a left or a right child
        superBlock= parent.BSearch(sum_deq-dir, i+blocks[b-1].sum_deq, 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].num_deq-left               ▷ consider the dequeues from the right child
        end if
        return this.parent.IndexDeq(superBlock, i)
    end if
end IndexDeq
```

Algorithm Leaf

```
void Append(block blk)                                        ▷ Append is only called by the owner of the leaf.
    head+=1
    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 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.

block[] blocks: array of blocks

int[] super: super[i] stores an approximate index of the superblock of the blocks in blocks whose group field have value i.

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

2 Proof of Linearizability

$\boxed{\text{TEST}}$ Fix the logical order of definitions (cyclic refrences).

$\boxed{\text{TEST}}$ Is it better to show $\mathtt{ops(EST_{n,\ t})}$ with $\mathtt{EST_{n,\ t}}$?

$\boxed{\text{Question}}$ A good notation for *the index of the* `b`?

$\boxed{\text{Question}}$ How to remove the notion of time? To say pre(n,i) contains n.blocks[0..i] instead of EST(n,t) which head=i at time t. Is it good? Furthermore, can we remove the notion of established blocks?

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 the index of the `b` is smaller than the index of the `b′`'s. 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`.

Invariant 2 (headPosition)**.** If the value of `n.head` is `h` then, `n.blocks[i]=null` for `i>h` and $\mathtt{n.blocks[i]}\neq\mathtt{null}$ 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 Line313 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 $\mathtt{n.blocks[h]}\neq\mathtt{null}$ 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 wether318 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 3 (headProgress)**.** `n.head` *is non-decreasing over time and* $\mathtt{n.blocks[i].end_{left}} \geq \mathtt{n.blocks[i-1].end_{left}}$ *and* $\mathtt{n.blocks[i].end_{right}}$ $\geq \mathtt{n.blocks[i-1].end_{right}}$.

Proof. The first claim follows trivially from the pseudocode since `n.head` is only incremented in the pseudocode in the lines 318 and 321 of the `Refresh()` .

Consider the block `b` written into `n.blocks[i]` by `TryAppend()` at Line 313. It is created by the `CreateBlock()` called at Line 311. Prior to this call to `CreateBlock()`, `n.head=i` at Line 310, so `n.blocks[i-1]` is already a non-null value `b′` by invariant 2. Thus the `CreateBlock(i-1)` that creates `b′` terminates before `CreateBlock(i)` that creates `b` is invoked. The value written into $\mathtt{b.end_{left}}$ at Line 333 of `CreateBlock()` was read from `n.left.head` at Line 331 of `CreateBlock`. Similarly, the value in $\mathtt{n.blocks[i-1].end_{left}}$ was read from `n.left.head`. Since `n.left.head` is non-decreasing $\mathtt{b'.end_{left}} \leq \mathtt{b.end_{left}}$. The proof for $\mathtt{end_{right}}$ is similar. ☐

Definition 4 (Subblock)**.** Block `b` is a *direct subblock* of `n.blocks[i]` if it is in $\mathtt{n.left.blocks[n.blocks[i-1].end_{left}+1..n.blocks[i].end_{left}]}$ $\cup\,\mathtt{n.right.blocks[n.blocks[i-1].end_{right}+1..n.blocks[i].end_{right}]}$ (The $\mathtt{end_{left}}$, $\mathtt{end_{right}}$ fields are written in 333 of `CreateBlock()`). Block `b` is a subblock of `n.blocks[i]` if `b` is a direct subblock of `n.blocks[i]` or a subblock of a direct subblock of `n.blocks[i]`.

Definition 5 (Superblock)**.** Block `b` is $\hat{\text{d}}$irect superblock of block `c` if c is a direct subblock of b. Block `b` is *superblock* of block `c` if `c` is a subblock of `b`.

Definition 6 (Operations of a block)**.** A leaf block `b` in a leaf represents operation `x` if $\mathtt{b.element{=}x}\neq\mathtt{null}$. The set of operations of block `b` are the operations in the subblocks of `b`. We denote the set of operations of block `b` by `ops(b)`.

We say block `b` is *propagated to node* `n` if `b` is in `n.blocks` or is a subblock of a block in `n.blocks`. We also say `b` contains `op` if $\mathtt{op}\in\mathtt{ops(b)}$.

Definition 7. A block b in n.blocks is *established* at time t if the last value written into n.head before t is greater than the index of b in n.blocks at time t. $EST_{n,\ t}$ is the set of established blocks of node n at time t.

head **Observation 8.** *Once a block* b *is written in* n.blocks[i] *then* n.blocks[i] *never changes.*

Lemma 9. *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$_{\texttt{left}}$..n.blocks[i].end$_{\texttt{left}}$] and n.right.blocks[n.blocks[i-1].end$_{\texttt{right}}$..n.blocks[i] (Lines lastLine, pr ??). Since end$_{\texttt{left}}$,end$_{\texttt{right}}$ are non-decreasing, so the range of the subblocks of n.blocks[i] which is (n.blocks[i-1].end$_{\texttt{dir}}$+1..n.blocks[i] does not overlap with the range of the subblocks of n.blocks[i-1]. ☐

append **Corollary 10** (No Duplicates)**.** *If* op *is in* n.blocks[i] *then there is no* j$\neq$i *such that* op$\in$ops(n.blocks[j]).

shedOrder **Lemma 11** (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 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 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.

:::headInc **Lemma 12** (head Increment). *If an n.Refresh instance reaches Line 313 instance and reads head=h (310) after it terminates head is greater than h.*

Proof. If Line 318 or 318 succeeded the claim holds, otherwise another process has incremented the head. □

ueRefresh **Lemma 13** (trueRefresh). *Let t_i be the time an instance of* `n.Refresh()` *is invoked and t_t be the time it terminates. Suppose the* `TryAppend(new, s)` *of the* `n.Refresh()` *returns* `true`*, then* $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_i}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_i}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_t})$.

Proof. From Lemma 11 we know that $\texttt{ops}(\texttt{EST}_{\texttt{n},\ t_i}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_t})$. So it remains to show the operations of $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_i}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_i})$ - $\texttt{ops}(\texttt{EST}_{\texttt{n},\ t_i})$, which we call *new operations*, are all in $\texttt{ops}(\texttt{EST}_{\texttt{n},\ t_t})$. If `TryAppend` returns true a block `new` is written into `n.blocks[h]` (Line 313). We show $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_i}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_t})$. The proof for the right child's claim is the same. Let `n.left.head` at $\texttt{t}_\texttt{i}$ be `hli`. Let `n.Refresh()` read `head` equal to h(Line 310). By the lines 332,331 the `new` block in `n.blocks[h]` contains `n.left.blocks[ n.blocks[h-1].end`$_{\texttt{left}}$`+1..left.head]`. Since `left.head` is read after $\texttt{t}_\texttt{i}$ then $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_i}) \subseteq$ `ops(n.left.blocks [0..left.head])`. By Lemma 11 `ops(n.left.blocks[0..n.blocks[h-1].end`$_{\texttt{left}}$`])` $\subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_i}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_t})$. Since after line 321 we are sure that the `head` is incremented (Lemma 12) and `n.head=h+1` at t_t so the new block is established at t_t and the new block contains the new operations which is what we wanted to show.

□

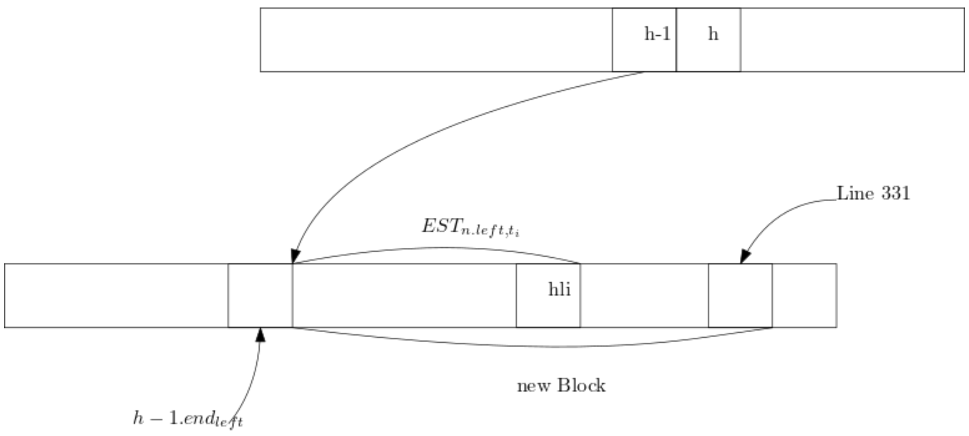


Figure 1: New established operations of the left child are in the new block.

ueRefresh **Lemma 14** (Precise True Refresh). *Let t_i be the time an instance of* `n.Refresh()` *read the head (Line 310) and t_t be the time its* `TryAppend(new, s)` *terminates with and returns* `true` *(Line 313). We have* $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_i}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_i}) \subseteq$ `ops(n.blocks)`.

Lemma 15 (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. We have* $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_1}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_1}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_2})$.

Proof.

If Line 313 of R_1 or R_2 returns `true`, then the claim is held by Lemma 13. Let R_1 read i and R_2 read $i+1$ from Line 310. 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 13 by the end of this instance $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_1}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_1})$ has been propagated.

Since R_2's `TryAppend()` returns `false` then there is another successful instance R_2' of `n.Refresh()` that has done `TryAppend()` successfully into `n.blocks[i+1]` before R_2 tries to append. 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. By Lemma 14 after R_2''s `CAS` we have $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_1}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_1}) \subseteq \texttt{ops}(\texttt{n.blocks})$. Also by Lemma 12 of R_2 head is more than $i+1$ after R_2ś 321 line, so the block appended by R_2' to `n` is established by then. To summarized t_1 is before R_2''s read head and R_2''s `CAS` is before R_2's termination. So $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_1}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_1}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_2})$. □

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

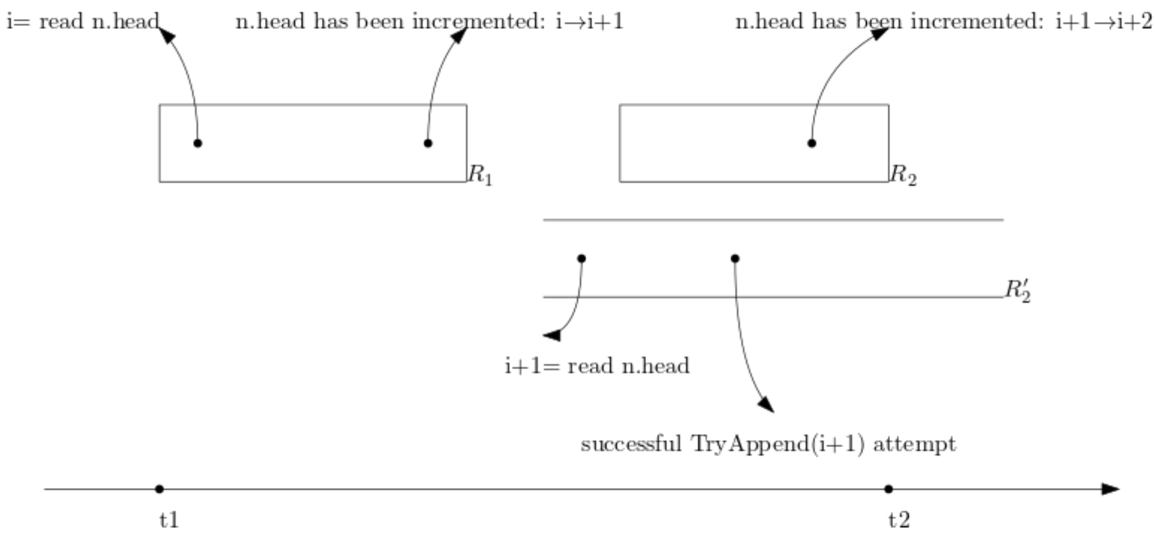


Figure 2: $t1 < r_1$ reading head $<$ incrementing `n.head` from i to $i+1 < R_2'$ reading head $<$ `TryAppend(i+1)` $<$ incrementing `n.head` from $i+1$ to $i+2 < t2$

this chain with more depth should be in the proof

Corollary 16 (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 15. □

Corollary 17. *After* `Append(blk)` *finishes* `ops(blk)`$\subseteq$`ops(root.blocks[x])` *for some* `x` *and only one* `x`*.*

Proof. Follows from Lemma 15, 10. □

blockSize **Lemma 18** (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 10. □

ocksBound **Lemma 19** (Subblocks Upperbound)**.** *Each block has at most p direct subblocks.*

Proof. It follows directly from Lemma 18 and the observation that each block contains at least one operation, induced from Line 312. □

ordering **Definition 20** (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 18 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 $\in$ `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 $\in$ `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: $E(n) = E(n, 1).E(n, 2).E(n, 3)...$
- $E_{n,i}$ is the ith 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 ith 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.*

getSearch **Lemma 21** (Get BSearch correctness)**.** *Preconditions of invocation of* `BSearch` *in* `GetENQ` *are satisfied i.e. if the direct subblock containg* $E_{n,b,i}$ *is computed in both cases in Lines* leftChildGet *405 and* rightChildGet *409.*

Proof. There are two cases that the subblock we're looking for is in the left child or the right child. If ith `enqueue` of `n.blocks[b]` was in the left child it would be in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` by definition def::subblock 4. `sum`$_{\texttt{enq}}$ is the sum of the number of enqueues in the prefix of a block. So $E_{n,b,i}$ is $E_{n.left,i+n.left.blocks[n.blocks[b].end_{left}-1]}$ which is $E_{n.left,b',i'}$ for some b' and i'. The parameters in leftChildGet 405 are correct so this `BSearch` returns the subblock containing $E_{n,b,i}$.

If the enqueue we were looking for was in the right child as there are `n.blocks[b].num`$_{\texttt{enq-left}}$ enqueues ahead of it in `n.blocks[b]` but not in `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`. So we need to search for `i-n.blocks[b].num`$_{\texttt{enq-left}}$`+` `n.right.blocks[n.blocks[b].end`$_{\texttt{right}}$`-1]` (Line rightChildGet 409). Other parameters are the same as the left child. □

get **Lemma 22** (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, the leaves of the tree, is pretty straight forward (Line getBaseCase 403). Since leaf blocks contain exactly one operation thus only `GetENQ(b,1)` can be called on a leaf block `b`. `l.GetENQ(b,1)` returns the operation stored in the bth block of leaf l. In the next paragraph we prove `n.GetENQ()` returns $E_{n,b,i}$, if `n.child.GetENQ()` works properly.

For non leaf nodes it is decided that the `i`th enqueue in block `b` of internal node `n` is in the `n.blocks[b]`'s subblocks in the left child of `n` or in the `n.blocks[b]`'s subblocks in the right child of `n` (line leftOrRight 404). From Definition ordering 20 we know operations in a block are ordered by their process id and since in leaves of the tree are ordered ascendingly by process id from left to right, thus operations from the left subblocks come before operations from the right subblocks in a block (See Figure figGet 3). Furthermore `b.num`$_{\texttt{enq-left}}$ stores the number of `enqueue()` operations from the `n.blocks[b]`'s subblocks in the left child of `n`. So if `i` is greater than `b.num`$_{\texttt{enq-left}}$ it means ith operation is propagated from the right child, otherwise we should search for the ith enqueue in the left child. By definition def::ops 6 and def::subblock 4 we need to search in subblocks of `n.blocks[b]` which their range is `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`$_{\texttt{right}}$`]`. We can compute the direct subblock containing the enqueue we are finding for with binary search by Lemma getSearch 21. Finaly `n.child.GetENQ()` is invoked on the subblock containing $E_{n,b,i}$ which returns $E_{n,b,i}$ by the hypothesis of the induction. □

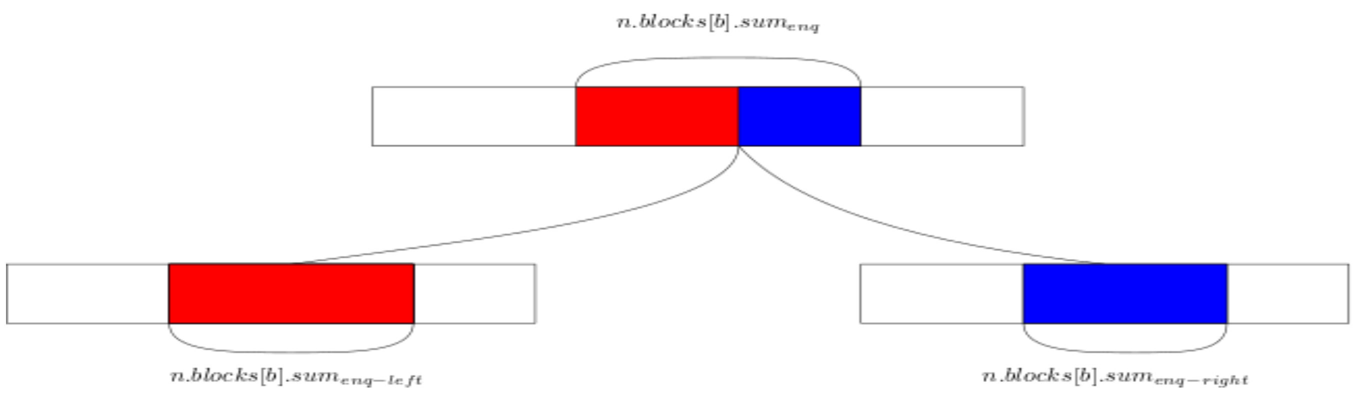


Figure 3: The number and ordering of the enqueues propagated from the left and the right child to `n.blocks[b]`. `n.blocks[b]` is colored in the tope node by the color of the subblocks propagted to it. Left subblocks in red come before the right subblocks in blue and `n.blocks[b].sum`$_{\texttt{enq}}$`=n.blocks[b].sum`$_{\texttt{enq-left}}$`+n.blocks[b].sum`$_{\texttt{enq-right}}$.

figGet

dsearch **Lemma 23** (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$\times$`root.blocks[b].size +root.blocks[end].size` . So the binary search takes Θ(log`root.blocks[b].size +root.blocks[end].size`)) steps. ☐

Lemma 24 (Index correctness)**.** `n.IndexDEQ(b,i)` *returns the rank in* $D(root)$ *of* $D_{n,b,i}$.

Proof. We will prove this by induction on the distance of `n` from the `root`. We can see the base case `root.IndexDEQ(b,i)` is trivial (Line 415). In the non-root nodes `n.IndexDEQ(b,i)` computes the superblock of the ith Dequeue in the bth block of `n` in `n.parent` by Lemma 25 (Line 418). After that the order in $D(n.parent, superblock)$ is computed and `index()` is called on `n.parent` recursively. Then if the operation was propagated from the right child the number of dequeues from the left child are added to it (Line 420), because the left child operations come before the right child operations (Definition 20). ☐

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? Make sure to show preconditions of all invocation of `BSearch` *are satisfied.*

uperBlock **Lemma 25** (Computing SuperBlock)**.** *After computing line 418 of* `n.IndexDEQ(b,i)`*,* `n.parent.blocks[superblock]` *contains* $D(n, b, i)$.

Proof. Lemmas 28,29,30,31. ☐

Lemma 26. *Value read for* `super[b.group]` *in line 418 is not null.*

Proof. Values $\texttt{np}_{\texttt{dir}}$ read in lines 337, `super` are set before incrementing in lines 315,316. So before incrementing $\texttt{num}_{\texttt{propagated}}$, $\texttt{super[num}_{\texttt{propagated}}\texttt{]}$ is set so it cannot be null while reading. ☐

Lemma 27. `super[]` *preserves order from child to parent; i.e. if in node* `n` *block* `b` *is before* `c` *then* `b.group` $\leq$ `c.group`

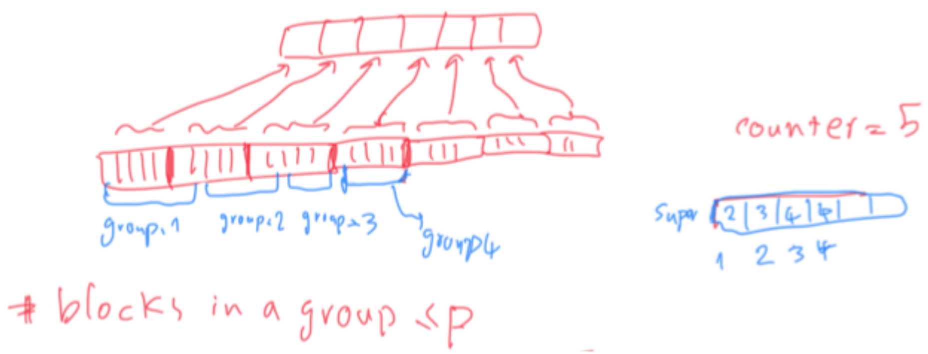
Proof. Line 329. Since $\texttt{num}_{\texttt{propagated}}$ is increasing. ☐

Lemma 28. *Let* `b, c` *be in node* `n`*, if* `b.group` $\leq$ `c.group` *then* `super[b.group]` $\leq$ `super[c.group]`

Proof. Line 315. ☐

Lemma 29. *The number of the* `blocks` *with* `group=i` *in a node is* $\leq p$.

Proof. For the sake of simplicity we assumed all the blocks are propagated from the left child. ☐

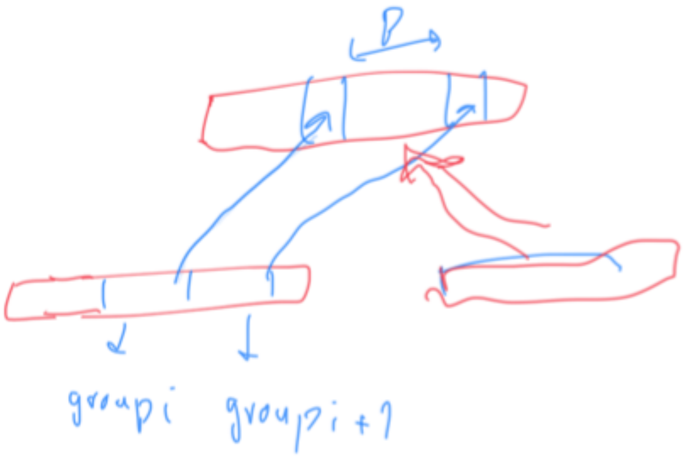


Lemma 30. `super[i+1]-super[i]`$\leq p$

Proof. 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. ☐

erCounter **Lemma 31** (super property)**.** *If* `super[i]` $\neq$ `null` *in node* `n`*, then* `super[i]` *is the index of the superblock of a block with* `time=i` *in* `n.parent.blocks`.

uperRange **Lemma 32.** *Superblock of* `b` *is within range* $\pm 2p$ *of the* `super[b.time]`.



Proof. `super[i]` is the index of the superblock of a block containing block b, followed by Lemma superCounter 31. `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 33. *Search in each level of* `IndexDeq()` *takes* $O(\log p)$ *steps.*

Proof. Show preconditions are satisfied and the range is p. □

We call the dequeues that return some 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= size - #enqueues. Note that the ith dequeue in the given block is not a non-null dequeue.

mputeHead **Lemma 34** (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 size of the queue if the operations in the prefix for the bth 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 342. If the size of a queue is greater than 0 then a `Dequeue()` would decrease the size of the queue, otherwise the size of the queue remains 0. Having size of the queue after the previous block and number of enqueues and dequeues in the block, Line computeLength 342 computes wether the queue becomes empty or the size of it.

$$\texttt{r}_{\texttt{enq}}\texttt{= (i}_{\texttt{d}}\texttt{ + root.blocks[b}_{\texttt{d}}\texttt{-1].sum}_{\texttt{deq}}\texttt{) - (root.blocks[b}_{\texttt{d}}\texttt{-1].size - root.blocks[b}_{\texttt{d}}\texttt{-1].sum}_{\texttt{enq}}\texttt{ + root.blocks[b}_{\texttt{d}}\texttt{-1].sum}_{\texttt{deq}}\texttt{)}$$

□

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

Theorem 35 (Main)**.** *The queue implementation is linearizable.*

Proof. We choose L in Definition 20 to be linearization ordering of operations and prove if we linearize operations as L the queue works consistently. □

Lemma 36 (satisfiability)**.** *L can be a linearization ordering.*

Proof. To show this we need to say if in an execution, $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts then $\texttt{op}_1$ is linearized before $\texttt{op}_2$. If $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts it means $\texttt{op}_1\texttt{.Append()}$ is terminated before $\texttt{op}_2\texttt{.Append()}$ starts. From Lemma 10 $\texttt{op}_1$ is in `root.blocks` before $\texttt{op}_2$ propagates so $\texttt{op}_1$ is linearized before $\texttt{op}_2$ by Definition 20.

Once some operations are aggregated in one block they will be propagated together up to the root and we can linearize them in any order among themselves. Furthermore in L we arbitrary choose the order to be by process id, since it makes computations in the blocks faster . □

Lemma 37 (correctness)**.** *If operations are applied as L on a sequential queue, the sequence of the responses would be the same as our algorithm.*

Proof. Old parts to review We show that the ordering L stored in the root, satisfies the properties of a linearizable ordering.

1. If op_1 ends before op_2 begins in E, then op_1 comes before op_2 in T.

   ► This is followed by Lemma 10. The time op_1 ends it is in root, before op_2, by Definition 20 op_1 is before op_2.

2. Responses to operations in E are same as they would be if done sequentially in order of L.

   ► Enqueue operations do not have any response so it does no matter how they are ordered. It remains to prove Dequeue d returns the correct response according to the linearization order. By Lemma 34 it is deduced that the head of the queue at time of the linearization of d is computed properly. If the Queue is not empty by Lemma 22 we know that the returning response is the computed index element.

□

Lemma 38 (Amortized time analysis)**.** `Enqueue()` *and* `Dequeue` *take* $O(\log^2 p + q)$ *steps (amortized anlysis), which* p *is the number of processes and* q *is the size of the queue at the time of invocation.*

Proof. `Enqueue(x)` consists of creating a `block(x)` and appending it to the tree. The first part takes constant time. To propagate `x` to the root the algorithm tries two `Refresh`es in each node of the path from the leaf to the root (Lines 302, 303). Each `Refresh` takes $O(1)$ steps since creating a block is done in constant time and does $O(1)$ `CAS`es. Since the height of the tree is $O(\log p)$, `Enqueue(x)` takes $O(\log p)$ steps.

A `Dequeue()` creates a block with null value element, appends it to the tree, computes its order among operations, and returns the response. The first two part is similar to an `Enqueue` operation. To compute the order there are some constant steps and `IndexDeq` is called. `IndexDeq` does a search with range p in each level (Lemma 32) which takes $O(log^2 p)$ in the tree. In the `FindResponse()` routine `DSearch()` in the root takes $\Theta(\log($`root.blocks[b].size +root.blocks[end].size` $)$ by Lemma 23, which is $O(\log$ size of the queue when `enqueue` is invoked$) + \log$ size of the queue when `dequeue` is invoked). Each search in `GetEnq()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma 19), so `GetEnq()` takes $O(\log^2 p)$ steps.

If we split `DSearch` time cost between the corresponding `Enqueue`, `Dequeue`, in amortized we have `Enqueue` takes $O(\log p + q)$ and `Dequeue` takes $O(\log^2 p + q)$ steps. ∎