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* $\texttt{num}_{\texttt{propagated}}$`= 0` : # groups of blocks that have been propagated from the node to its parent.

▶ *Block*

- *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* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: indices 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 $\texttt{b}\geq 0$ and $\texttt{x} \in \{\texttt{enq, deq}\}$)
- $\texttt{blocks[b].sum=blocks[b].sum}_{\texttt{enq}}\texttt{+blocks[b].sum}_{\texttt{deq}}$ (for $\texttt{b}\geq 0$)
- $\texttt{blocks[b].num}_{\texttt{x}}\texttt{=blocks[b].sum}_{\texttt{x}}\texttt{-blocks[b-1].sum}_{\texttt{x}}$
  (for b>0 and $\texttt{x} \in \{\emptyset, \texttt{enq, deq, enq-left, enq-right, deq-left, deq-right}\}$)

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, i>= IndexDeq(leaf.head, 1)
deqRest
    output= FindResponse(b, i)
    return output
end Dequeue

<int, int> FindResponse(int b, int i)
                                        ▷ Returns the the response to the D_{root,b,i}.
    if root.blocks[b-1].size + root.blocks[b].num_enq - i < 0 then
checkEmpty
        return null                     ▷ Check if the queue is empty.
    else
computeE
        e= i - root.blocks[b-1].size + root.blocks[b-1].sum_enq
                                        ▷ E_e(root) is the response.
findAnswer
        return root.GetENQ(root.DSearch(e, b))
    end if
end FindResponse
```

Algorithm *Node*

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

boolean REFRESH()
    h= head                                                        readHead
    <new, np_left, np_right>= CREATEBLOCK(h)                       keCreateBlock
        ▷ 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.   addOP
    else if blocks.tryAppend(new, h) then                          cas
        for each dir in {left, right} do                           okcas
            CAS(dir.super[np_dir], null, h)   ▷ Write would work too.   setSuper
            CAS(dir.num_propagated, np_dir, np_dir+1)              incNP
        end for
        CAS(head, h, h+1)                                          ncrementHead1
        return true
    else
        CAS(head, h, h+1)      ▷ Even if another process wins, help   computeLength
            to increase the head. The winner might have fallen sleep before increasing
            head.                                                  ncrementHead2
        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)
    newBlock.group= num_propagated                                 setGroup
    for each dir in {left, right} do
        index_last= dir.head-1                                     lastLine
        index_prev= blocks[i-1].end_dir                            prevLine
        newBlock.end_dir= index_last                               endDefLine
        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                                 setNP
        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  + 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
<int, int> DSEARCH(int e, int end)        ▷ Returns <b,i> if E_e(root) = E_i(root, b).
    start= end-1
    while root.blocks[start].sum_enq ≥ e do
        start= max(start-(end-start), 0)                           doubling
    end while
    b= root.BSearch(sum_enq, e, start, end)
    i= e- root.blocks[b-1].sum_enq                                 hComputei
    return <b,i>
end DSEARCH
```

Algorithm Node

```
⇝ Precondition: blocks[b].num_enq ≥ i ≥ 1
element GetEnq(int b, int i)                                    ▷ Returns the element of E_i(this, b).
    if this is leaf then
[tBaseCase]  return blocks[b].element
[ftOrRight]  else if i ≤ blocks[b].num_enq-left then            ▷ E_i(this, b) is in the left child of this node.
[tChildGet]  subBlock= left.BSearch(sum_enq, i+blocks[b-1].sum_enq-left, blocks[b-1].end_left+1, blocks[b].end_left)
        return left.GetEnq(subBlock, i)
    else
        i= i-blocks[b].num_enq-left
[tChildGet]  subBlock= right.BSearch(sum_enq, i+right.blocks[b-1].sum_enq-right, 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
[xBaseCase]  return <b, i>
    else
        dir= (parent.left==n)?  left:  right                    ▷ check if this node is a left or a right child
[puteSuper]  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 left then
[viousLeft]  i+= blocks[b-1].sum_enq-blocks[superBlock-1].sum_enq-left     ▷ consider the enqueues in the previous blocks from the left child
        end if
        if dir is right then
[iousRight]  i+= blocks[b-1].sum_enq-blocks[superBlock-1].sum_enq-right    ▷ consider the enqueues in the previous blocks from the right child
[foreRight]  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.
[pendStart]  blk.group= head
    blocks[head]= blk
[appendEnd]  head+=1
    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

TEST Fix the logical order of definitions (cyclic refrences).

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

Question A good notation for *the index of the* b?

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. A block in a node's blocklist 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`.

n::headInc **Lemma 2** (head Increment)**.** *Let* `R` *be an instance of* `Refresh` *on node* `n` *that reaches Line 313. After* `R` *terminates* `n.head` *is greater than h, the value read in line 310 of* `R`.

Proof. If Line 318 or 321 are successful then the claim holds, otherwise another process has incremented the head from `h` to `h+1`. ☐

dPosition **Invariant 3** (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`. We show the invariant still holds after these two changes.

In the algorithm, some value is appended to `n.blocks[]` by writing into `n.blocks[head]` only in Line 313. Writing into `n.blocks[head]` preserves the invariant, since the claim does not talk about `n.blocks[head]`. The value of `n.head` is modified only in lines 318 and 321. Depending on whether the `TryAppend()` in Line 313 succeeded or not, we show that the claim holds after the increment of `n.head` in either case. If `n.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 whether 318 or 318 return true or false, after they finish we know that `n.head` has been incremented from the value read in Line 310 (Lemma 2). The failure case is also the same since it means some non-null value has been written into `n.blocks[head]` by some process. ☐

Explain More

dProgress **Lemma 4** (headProgress)**.** `n.head` *is non-decreasing over time. If* `n.blocks[i]`$\neq$`null` *and* `i.0` *then* `n.blocks[i].end`$_{\texttt{left}} \geq$ `n.blocks[i-1].end`$_{\texttt{left}}$ *and* `n.blocks[i].end`$_{\texttt{right}} \geq$ `n.blocks[i-1].end`$_{\texttt{right}}$.

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

Consider the block `b` written into `n.blocks[i]` by `TryAppend()` at Line 313. It is created by the `CreateBlock(i)` called at Line 311. Prior to this call to `CreateBlock(i)`, `n.head=i` at Line 310, so `n.blocks[i-1]` is already a non-null value `b′` by Invariant 3. Thus the `CreateBlock(i-1)` that creates `b′` terminates before `CreateBlock(i)` that creates `b` is invoked. The value written into `b.end`$_{\texttt{left}}$ at Line 333 of `CreateBlock(i)` was read from `n.left.head-1` at Line 331 of `CreateBlock(i)`. Similarly, the value in `n.blocks[i-1].end`$_{\texttt{left}}$ was read from `n.left.head-1` during the call to `CreateBlock(i-1)`. Since `n.left.head` is non-decreasing `b′.end`$_{\texttt{left}} \leq$ `b.end`$_{\texttt{left}}$. The proof for `end`$_{\texttt{right}}$ is similar. ☐

:subblock **Definition 5** (Subblock)**.** Block `b` is a *direct subblock* of `n.blocks[i]` if it is in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` $\cup$ `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]` . 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]`.

append **Corollary 6** (No Duplicates)**.** *If* `op` *is in* `n.blocks[i]` *then there is no* `j`$\neq$`i` *such that* `op`$\in$`ops(n.blocks[j])`.

Proof. Operation `op` is invoked only one time in an execution because every operations invoked is distinct. Since there is node `n` which `op` is in two different blocks of `n`, there is node $\mathtt{n}'$ that is the lowest height node in the tree that contains `op` in two of its blocks `b1,b2`. By Definition 5, `b1` and `b2` have distinct subblocks(not only direct subblocks) and since `op` is in only one leaf block, then it cannot be in both `b1` and `b2`. □

Definition 7 (Superblock)**.** Block `b` is *direct 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`.

`def::ops` **Definition 8** (Operations of a block)**.** A leaf block `b` in a leaf represents `enqueue(x)` if `b.element=x`$\neq$`null`. Else if `b.element=null` b represents a `dequeue()`. 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 `op`$\in$`ops(b)`.

Definition 9. A block `b` in `n.blocks` is *established* at time t if `n.head`> index of `b` at time t. $EST_{n,\ t}$ is the set of established blocks of node `n` at time t.

`head` **Observation 10.** *Once a block* `b` *is written in* `n.blocks[i]` *then* `n.blocks[i]` *never changes.*

Lemma 11. *Every block has at 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 $\mathtt{n.left.blocks[n.blocks[i-1].end_{left}..n.blocks[i].end_{left}]}$ and $\mathtt{n.right.blocks[n.blocks[i-1].end_{right}..n.blocks[i]}$ (Lines 331, 332). Since $\mathtt{end_{left},end_{right}}$ are non-decreasing ($\mathtt{n.blocks[i].end_{left|right}}$>$\mathtt{n.blocks[i-1].end_{left|right}}$), so the range of the subblocks of `n.blocks[i]` which is $\mathtt{(n.blocks[i-1].end_{dir}+1..n.blocks[i].end_{dir})}$ does not overlap with the range of the subblocks of `n.blocks[i-1]`. □

`shedOrder` **Lemma 12** (establishedOrder)**.** *If time* $t <$ *time* t', *then* $\mathtt{ops(EST_{n,\ t})}\subseteq\mathtt{ops(EST_{n,\ t'})}$.

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 grow. □

useless?

▶ Processes are numbered from 1 to p and leaves of the tree are assigned from left to right. We will show in Lemma blockSize 22 that there is at most one operation from each process in a given block.

ordering **Definition 13** (Ordering of operations inside the nodes)**.**

- The prefix of an operation op in the sequence of operations S is the sequence of operations strictly before op.
- $E(n,b)$ is the sequence of enqueue operations in `ops(n.blocks[b])` defined recursively as follows. $E(leaf,b)$ is the single enqueue operation in `ops(leaf.blocks[b])` or an empty sequence if `leaf.blocks[b].num`$_{\texttt{enq}}$`=0`. If `n` is an internal node, then

$$\begin{aligned}E(n,b) =&E(n.left, n.blocks[b-1].end_{\text{left}}+1)\cdot E(n.left, n.blocks[b-1].end_{\text{left}}+2)\cdots E(n.left, n.blocks[b].end_{\text{left}})\cdot\\ &E(n.right, n.blocks[b-1].end_{\text{right}}+1)\cdot E(n.right, n.blocks[b-1].end_{\text{right}}+2)\cdots E(n.right, n.blocks[b].end_{\text{right}})\end{aligned}$$

- $E_i(n,b)$ is the ith enqueue in $E(n,b)$.
- The order of the enqueue operations in the node n is $E(n) = E(n,1)\cdot E(n,2)\cdot E(n,3)\cdots$
- $E_i(n)$ is the ith enqueue in $E(n)$.
- $D(n,b)$ is the sequence of dequeue operations in `ops(n.blocks[b])` defined recursively as follows. $D(leaf,b)$ is the single dequeue operation in `ops(leaf.blocks[b])` or an empty sequence if `leaf.blocks[b].num`$_{\texttt{deq}}$`=0`. If `n` is an internal node, then

$$\begin{aligned}D(n,b) =&D(n.left, n.blocks[b-1].end_{\text{left}}+1)\cdot D(n.left, n.blocks[b-1].end_{\text{left}}+2)\cdots D(n.left, n.blocks[b].end_{\text{left}})\cdot\\ &D(n.right, n.blocks[b-1].end_{\text{right}}+1)\cdot D(n.right, n.blocks[b-1].end_{\text{right}}+2)\cdots D(n.right, n.blocks[b].end_{\text{right}})\end{aligned}$$

- $D_i(n,b)$ is the ith enqueue in $D(n,b)$.
- The order of the dequeue operations in the node n: $D(n) = D(n,1)\cdot D(n,2)\cdot D(n,3)...$
- $D_i(n)$ is the ith dequeue in $D(n)$.

def::lin **Definition 14** (Linearization)**.** $L = E(root,1).D(root,1).E(root,2).D(root,2).E(root,3).D(root,3)...$

▶ In the non-root nodes, we only need ordering of enqueues and dequeues among the operations of their own type. Since `GetENQ()` only searches among enqueues and `IndexDEQ()` works with dequeues.

Lemma 15 (trueRefresh)**.** *Let t_i be the time an instance* `R` *of* `n.Refresh()` *is invoked and t_t be the time it terminates. If the* `TryAppend(new, s)` *of* `R` *returns* `true`*, then* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$.

Proof. Since `TryAppend` returns true a block `new` is written into `n.blocks[h]` in Line 313.

We show $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)}\subseteq\texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$. Let `h` be the value `n.Refresh()` reads from `n.head` at line 310, $\texttt{h}_{\texttt{left,i}}$ be the value of `n.left.head` at $\texttt{t}_\texttt{i}$ and $\texttt{h}_{\texttt{left,read}}$ be the value read from `n.left.head-1` at line 331. $\texttt{end}_{\texttt{left}}$ field of the block returned by `CreateBlock(i)` is $\texttt{h}_{\texttt{left,read}}$. By lines 332 and 331 the `new` block in `n.blocks[h]` contains $\texttt{n.left.blocks[n.blocks[h-1].end}_{\texttt{left}}\texttt{+1..h}_{\texttt{left,read}}\texttt{]}$. Since `left.head` is read after $\texttt{t}_\texttt{i}$ then $\texttt{h}_{\texttt{left,read}}\texttt{>h}_{\texttt{left,i}}$ which means $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)}\subseteq\texttt{ops(n.left.blocks [0..h}_{\texttt{left,read}}\texttt{])}$. After the successful `TryAppend` in line 313 we know all blocks in $\texttt{n.left.blocks[0..h}_{\texttt{left,read}}\texttt{-1}$ are subblocks of `n.blocks[0..h]` by the definition of subblock. At $\texttt{t}_\texttt{t}$ we have `n.head>h` by Lemma 4. So `n.blocks[1..h]` are in $\texttt{EST}_{\texttt{n,t}_\texttt{t}}$ by definition of `EST`. Note that after line 321 we are sure that the `head` is incremented by Lemma 2) which means `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. The proof for $\texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)}\subseteq\texttt{ops(EST}_{\texttt{n, t}_\texttt{t}}\texttt{)}$is the same. □

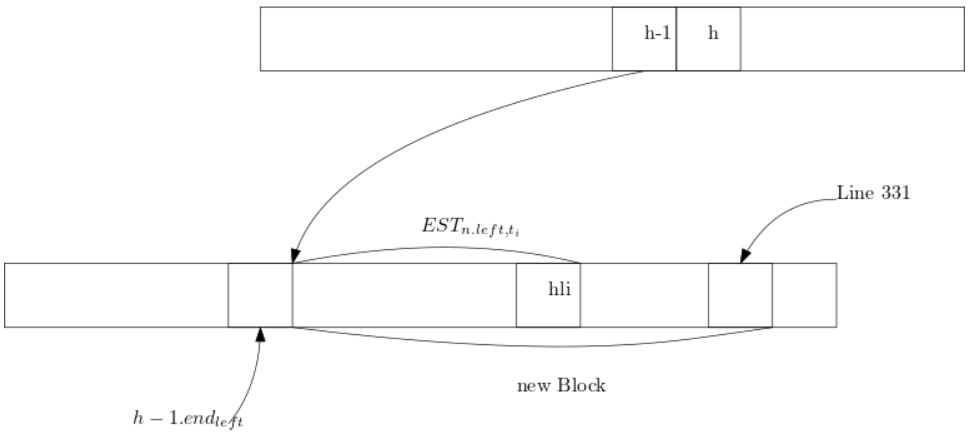


Figure 1: New established operations of the left child are in the new block.

Lemma 16 (Stronger 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* $\texttt{ops(EST}_{\texttt{n.left, t}_\texttt{i}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_\texttt{i}}\texttt{)} \subseteq \texttt{ops(n.blocks)}$.

Lemma 17 (Double Refresh)**.** *Consider two consecutive instances R_1, R_2 of* `Refresh()` *on* `n` *by process p. Let t_1 be the time R_1 is invoked and t_2 be the time R_2 terminated. If R_1 and R_2 both fail and return false, then 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 15. 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 15 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. Since R_2' creates the block after reading the value $i+1$ from `n.head`(Line 310) and R_1 reads the value i from `n.head` and the `head`'s value is increaing by Lemma 4 then $t_{R2'}\,{}_{310} > t_{R1}\,{}_{310} > t_1$ (See Figure 2). By Lemma 16 after R_2''s `CAS` we have $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_1}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_1}) \subseteq \texttt{ops(n.blocks)}$. Also by Lemma 2 on R_2 the value of `n.head` head is more than $i+1$ after R_2terminates, so the block appended by R_2' to `n` is established by then (`n.head>=i+2>i+1`). To summarize t_1 is before R_2''s `read n.head` and R_2''s successful `CAS` is before R_2's termination. So by Lemma 16 $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ t_1}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ t_1}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ t_2})$. □

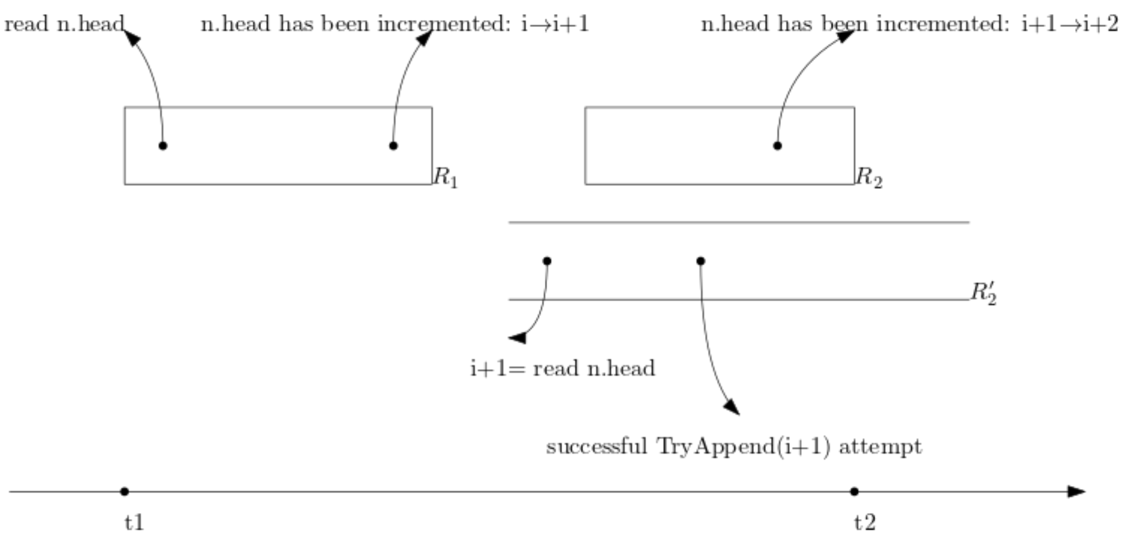


Figure 2: $t1 < r_1$ reading head $<$ incrementing `n.head` from i to $i+1 < R_2'$ reading head $<$ `TryAppend(i+1)` $<$ incrementing `n.head` from $i+1$ to $i+2 < t2$

this chain with more depth should be in the proof

Definition 18. $t_{before\ line}$ is the immediate time before running Line *line*. $t_{before\ line}$ is the immediate time after running Line *line*.

Corollary 19. $\texttt{ops}(\texttt{EST}_{\texttt{n.left},\ \texttt{t}_{\texttt{before}\ 302}}) \cup \texttt{ops}(\texttt{EST}_{\texttt{n.right},\ \texttt{t}_{\texttt{before}\ 302}}) \subseteq \texttt{ops}(\texttt{EST}_{\texttt{n},\ \texttt{t}_{\texttt{after}\ 303}})$

Proof. If the first `Refresh()` in line 302 returns `true` then by Lemma 15 the claim holds. Also if first `Refresh()` failed and the second `Refresh()` succeeded the claim still holds by Lemma 15. Finally, if both failed the claim is satisfied by Lemma 17. □

Corollary 20 (Propagate Step)**.** *All operations in* `n`*'s children's established blocks before running line 302 of a* `Propagate` *routine are guaranteed to be in* `n`*'s established blocks after line 303.*

Proof. If 302 or 303 succeed, the claim is true by Lemma 15. Otherwise Lines 302 and 303 satisfy the preconditions of Lemma 17. □

Corollary 21. *After* `Append(blk)` *finishes* `ops(blk)`$\subseteq$`ops(root.blocks[x])` *for exactly one* `x`*.*

Proof. After `Append(blk)`'s termination, `blk` is in `root.blocks` since `blk` is established in the leaf it has been added to. By applying Lemma 20 inductively it is propagated up to the root. Finally Lemma 6 shows only one block in the root contains `blk`. □

blockSize **Lemma 22** (Block Size Upper Bound)**.** *Each block contains at most one operation of each processs.*

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process p in block b in node n. Without loss of generality op_1 is invoked earlier than op_2. A process cannot invoke more than one operations concurrently, so op_1 has to be finished before op_2. By Corollary 21, before appending op_2 to the tree op_1 exists in every node on the path from p's leaf to the root, because op_1's `Append` is finished before op_2's `Append` starts. So, there is some block b' before b in n containing op_1. Existence of op_1 in b and b' contradicts Lemma 6. □

ocksBound **Lemma 23** (Subblocks Upperbound)**.** *Each block has at most p direct subblocks.*

Proof. The claim follows directly from Lemma 22 and the observation that each block appended to the tree contains at least one operation, due to the test on Line 312. We can also see the blocks in the leaves have exactly one operation in the `Enqueue()` and `Dequeue()` routines. □

get

Lemma 24 (Get correctness)**.** *If* `n.blocks[b].num`$_{\texttt{enq}}$`≥i` *then* `n.GetENQ(b,i)` *returns the* `element` *enqueued by* $E_i(n,b)$.

Proof. We are going to prove this lemma by induction on the height of node `n`. For the base case `n` is a leaf. Leaf blocks each contain exactly one operation, so by the preconditions of `GetENQ()`, only `n.GetENQ(b,1)` can be called and `n.blocks[b]` contains an enqueue. At Line 403 `n.GetENQ(b,1)` returns the `element` of the `enqueue` operation stored in the bth block of leaf `n` .

For the induction step we prove `n.GetENQ(b,i)` returns $E_i(n,b)$, if `n.child.GetENQ(b,i)` returns $E_i(n.child,b)$. In Line 404 it is decided for the non-leaf nodes that the `i`th enqueue in `b`th block of internal node `n` is in the `n.blocks[b]`'s left child or right child subblocks. From Definition 13 of $E(n,b)$ we know enqueue operations in a block are ordered by their process id and since the leaves of the tree are ordered 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 3). Furthermore the `num`$_{\texttt{enq-left}}$ field in a block stores the number of `enqueue()` operations from the blocks's subblocks in the left child of `n`. So ith enqueue operation is propagated from the right child if `i` is greater than `b.num`$_{\texttt{enq-left}}$. otherwise we should search for the ith enqueue in the left child. By definition 8 and 5 we need to search in subblocks of `n.blocks[b]` from the range `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`$_{\texttt{right}}$`]`.

If the ith `enqueue` of `n.blocks[b]` is in the left child it would be ith enqueue in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` by Definition 5. Also we know there are $eb = n.blocks[b-1].sum_{enq-left}$ enqueues in the blocks before this range, so $E_i(n,b)$ is $E_{i+eb}(n.left)$ which is $E_{i'}(n.left,b')$ for some b' and i'. We can compute b' search for $i+eb$th enqueue in `n.left` and i' is `i+eb-n.left.blocks[`$b'-1$`].sum`$_{\texttt{enq}}$. The parameters in 405 are for searching $E_{i+eb}(n.left)$ in `n.left.block` in the expected range of blocks, so this `BSearch` returns the index of the subblock containing $E_i(n,b)$.

Else if the enqueue we are looking for is in the right child then 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.blocks[b-1].sum`$_{\texttt{enq-right}}$ (Line 409). Other parameters are assigned similar for the left child. So in both cases the direct subblock containing $E_i(n,b)$ is computed in Lines 405 and 409.

Finally, `n.child.GetENQ()` is invoked on the subblock containing $E_i(n,b)$ which returns $E_i(n,b)$ by the hypothesis of the induction. ☐

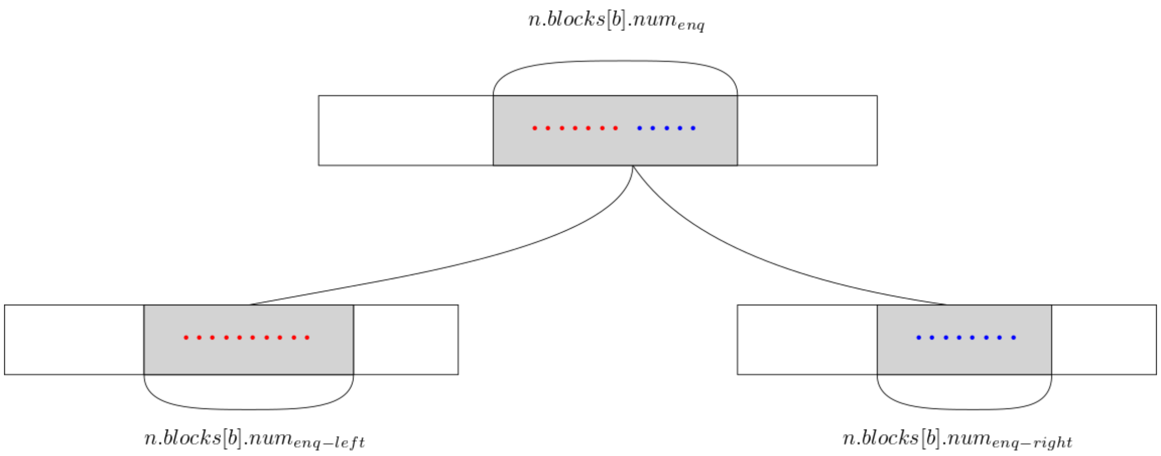


Figure 3: The number and ordering of the enqueue operations propagated from the left and the right child to `n.blocks[b]`. Enqueue operations from the left subblocks (colored red), are ordered before the enqueue operations from the right child (colored blue).

figGet

dsearch

Lemma 25 (DSearch correctness)**.** *Assume* `root.blocks[end].sum`$_{\texttt{enq}}$$\geq$`e` *and* $E_e(root)$*'s* `element` *is the response to some* `Dequeue()` *operation in* `root.blocks[end]`*.* `DSearch(e, end)` *returns* `<b, i>` *such that* $E_i(root, b) = E_e(root)$*.*

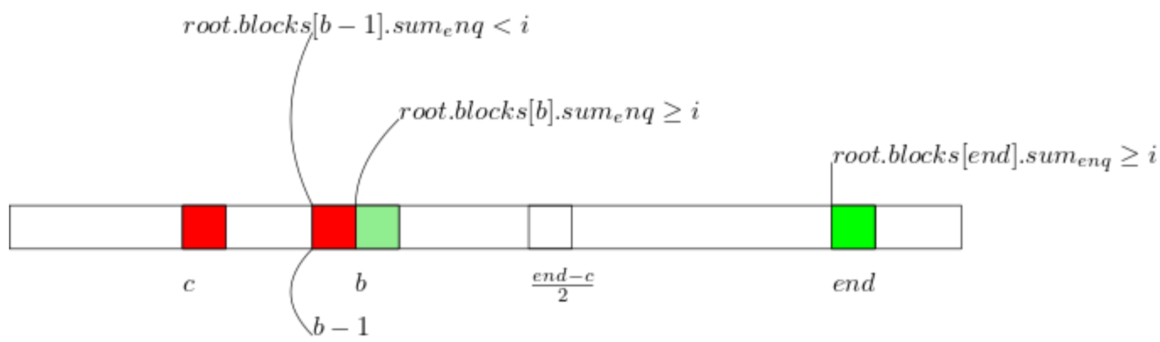
Proof. It is trivial to see that the doubling search from `root.blocks[end]` to `root.blocks[0]` will find $E_e(root)$ eventually. Because `root.blocks[].sum`$_{\texttt{enq}}$ is an increasing value from 0 to some value greater than `e`. So there is a `b` that `root.blocks[b].sum`$_{\texttt{enq}}$$> e$ but `root.blocks[b-1].sum`$_{\texttt{enq}}$$< e$.

First we show `end-b`$\leq$ $2\times$`root.blocks[b].size +root.blocks[end].size +1`. From line 312, we know that size of the every block in the tree is greater than 0. So each block in `root.blocks[b..end]` contains at least one `Enqueue` or at least one `Dequeue`. Suppose there were more than `root.blocks[b].size Dequeue`s in `root.blocks[b+1..end-1]`. Then the queue would become empty at some point after `blocks[b]`'s last operations and before `root.blocks[end]`'s first operation. Which means the response to to a `Dequeue` in `root.blocks[end]` could not be in $E(n, b)$. Furthermore since the size of the queue would become `root.blocks[end].size` after the `root.blocks[end]`, there cannot be more than `root.blocks[b].size + root.blocks[end].size` `Enqueue`s. Because there can be at most `root.blocks[b].size Dequeue`s and the final size is `root.blocks[end].size` . Overall there can be at most $2\times$`root.blocks[b].size + root.blocks[end].size` operations in `root.blocks[b+1..end-1]` and since each block size is ≥ 1 thus there are at most $2\times$`root.blocks[b].size + root.blocks[end].size` blocks in between `root.blocks[b]` and `root.blocks[end]`. So `end-b`$\leq$ $2\times$`root.blocks[b].size +root.blocks[end].size +1`. See Figure ??.

Now that we know there are at most `root.blocks[b].size +root.blocks[end].size` blocks in between `root.blocks[b]` and `root.blocks[end]` then with doubling search in $\Theta\big(\log($`root.blocks[b].size +root.blocks[end].size` $)\big)$ steps we reach `start=c` that the `root.blocks[c].sum`$_{\texttt{enq}}$ is less than `e` and `end-c` is not more than $2\times$`root.blocks[b].size +root.blocks[end].size` . Beause otherwise, then `(end-c)/2` satisfied the `root.blocks[(end-c)/2].sum`$_{\texttt{enq}}$`<e`. In line 804 the differnece between `end` and `start` is doubled. See Figure 4.

After computing `b`, the value `i` is computed via the definition of `sum`$_{\texttt{enq}}$ in constant time (Line 807). So the routine non constant part is the binary search which takes $\Theta(\log$`root.blocks[b].size +root.blocks[end].size` $))$ steps from the first paragraph.

$\square$



:doubling

Figure 4: Distance relations between b, c, end

Lemma 26. *Let n.propagates be the number of groups of blocks that have been propagated from node* `n` *to its parent (successful* `n.parent.Refresh()`*). We have* $\texttt{num}_{\texttt{propagated}} \leq$ *n.propagates* $\leq \texttt{num}_{\texttt{propagated}}$`+p`*.* `p` *is the number of processes.*

Proof. $\texttt{num}_{\texttt{propagated}}$ is incremented after propagating (Line incNP 316). Since maybe some process falls sleep before incrementing $\texttt{num}_{\texttt{propagated}}$ it may be behind by `p`. □

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 setGroup 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]`

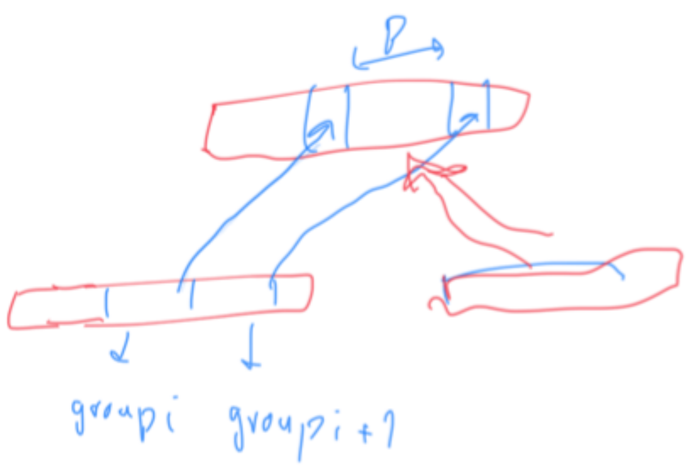
Proof. Line setSuper 315. □

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.group]`*.*

Proof. `super[i]` is the index of the superblock of a block containing block b, followed by Lemma 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. □

Lemma 34 (Computing SuperBlock)**.** *For the* `superblock` *value computed in line 418 of* `n.IndexDEQ(b,i)` *we have* `n.parent.blocks[superblock]` *contains* $D_{n,b,i}$*.*

Proof. First we show the value read for `super[b.group]` in line 418 is not null. 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. Then by Lemma 32if we search in the range p, we can find the superblock.

□

Lemma 35 (Index correctness)**.** *If* $\texttt{n.blocks[b].num}_{\texttt{deq}}\geq\texttt{i}$ *then* `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 where `n` is root 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 34 (Line 418). After that the order in $D(n.parent, superblock)$ is computed. Note that by Lemma 22 in each block there is at most one operation from each process and operations of one type are ordered based on the order in the subblocks (See Figure 5). Finally `index()` is called on `n.parent` recursively and it returns the correct response from induction hypothesis. If the operation was propagated from the right child the number of dequeues from the left child are added to it (Line ??), because the left child operations come before the right child operations (Definition 13). □

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

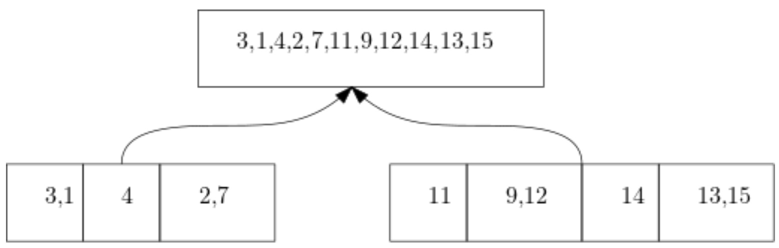


Figure 5: Relation of ordering of operations of a block from its subblocks

Definition 36. Assume the operations in L are applied on an empty queue. If element of `enqueue e` is the response to `dequeue d` then we say `R(d)=e`. If `d` 's response id `null` (queue is empty) then `R(d)=null`.

Definition 37. In an execution on a queue, the dequeue operations that return some value are called *non-null dequeues.*

Observation 38. *In a sequential execution on a queue, kth non-null dequeue returns the* `element` *of kth enqueue.*

Lemma 39. `root.blocks[b].size` *is 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.*

Proof. need to say? :: 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. By definition 13 enqueue operations come before dequeue operations in a block in L.

We prove the claim by induction on `b`. Base case `b=0` is trivial since the queue is initialy empty and `root.blocks[0].size=0`. For `b=i` we are going to use the hypothesis for `b=i-1`. If there are more than `root.blocks[i-1].size+ root.blocks[i].sum`$_{\texttt{enq}}$ dequeue operations in `root.blocks[i]` then the queue would become empty after `root.blocks[i]`. Otherwise we can compute the size of the queue after bth block using with this equality `root.blocks[b].size= root.blocks[b-1].size+ root.blocks[b].sum`$_{\texttt{enq}}$`- root.blocks[b].sum`$_{\texttt{deq}}$ (Line 342). See Table 1 for an example of running some blocks of operations on an empty queue. ☐

Lemma 40 (Duality of #non-null dequeues and `block.size`)**.** *If the operations are applied with the order of L, the number of non-null dequeues in the prefix for a block* `b` *is* `b.sum`$_{\texttt{enq}}$`-b.size`

Proof. There are `b.sum`$_{\texttt{enq}}$ enqueue operations in the prefix for `b`, then the size of the queue after the prefix for `b` is `#enqs - #non-null dequeues` in the prefix for `b`, by Observation 35. So `#non-null dequeues` is `b.sum`$_{\texttt{enq}}$`-b.size` . The correctness of the `block.size` field is shown in Lemma 39. ☐

Lemma 41. `R(D`$_{\texttt{root,b,i}}$`)` *is* `null` *iff* `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0`.

Lemma 42 (Computing Response)**.** `FindResponse(b,i)` *returns* `R(D`$_{\texttt{root,b,i}}$`).element`.

Proof. First note that by Definition 13 the linearization ordering of operations will not change as new operations come so instead of talking about the linearization of operations before the $E_i(root, b)$ we talk about what if the whole operation in the linearization are applied on a queue.

$D_{root,b,i}$ is $D_{root,root.blocks[b-1].sum_{deq}+i}$ from the definition 13 and sum_{enq}. $D_{root,b,i}$ returns `null` if `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0` by Lemma 41 (Line 220). Otherwise if it is d'th non-null dequeue in L it returns d'th enqueue by Observation 38 . By Lemma 40 there are `root.blocks[b-1].sum`$_{\texttt{enq}}$ `- root.blocks[b-1].size` non-null dequeue operations before prefix for `root.blocks[b-1]`. Note that the dequeues in `root.blocks[b]` before the ith dequeue are non-null dequeues. So the response is $E_{i-root.blocks[b-1].size+root.blocks[b-1].sum_{\text{deq}}}(root)$ (Line 222). See figure 6.

After computing `e` we can find `b,i` such that $E_i(root, b) = E_e(root)$ using `DSearch` and then find its `element` using `GetEnq` (Line 223). ☐

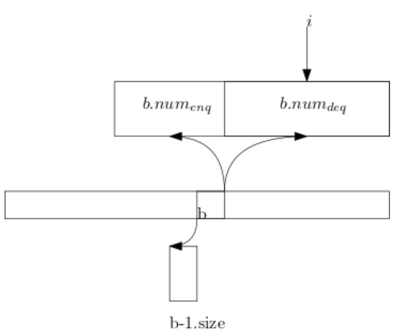


Figure 6: The position of $E_i(root, b)$.

	DEQ()	ENQ(5), ENQ(2), ENQ(1), DEQ()	ENQ(3), DEQ()	ENQ(4), DEQ(), DEQ(), DEQ(), DEQ()
#enqueues	0	3	1	1
#dequeues	1	1	1	4
#non-null dequeues	0	1	2	5
size	0	2	2	0

qhistory Table 1: An example of root blocks fields. Blocks are from left to right and operations in the blocks are also from the left to right.

Theorem 43 (Main)**.** *The queue implementation is linearizable.*

Proof. We choose L in Definition ordering 13 to be linearization ordering of operations and prove if we linearize operations as L the queue works consistently. □

Lemma 44 (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 append 6 $\texttt{op}_1$ is in `root.blocks` before $\texttt{op}_2$ propagates so $\texttt{op}_1$ is linearized before $\texttt{op}_2$ by Definition ordering 13.

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 45 (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 append 6. The time op_1 ends it is in root, before op_2, by Definition ordering 13 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 computeHead 42 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 get 24 we know that the returning response is the computed index element.

□

Lemma 46 (Amortized time analysis)**.** `Enqueue()` *and* `Dequeue()`*, each take* $O(\log^2 p + \log q)$ *steps in amortized analysis. Where* p *is the number of processes and* q *is the size of the queue at the time of invocation of operation.*

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). We can see from the code that each `Refresh` takes constant number of steps since creating a block is done in constant time and does $O(1)$ `CAS`es. Since the height of the tree is $\Theta(\log p)$, `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 enqueue operations, and returns the response. The first two part is similar to an `Enqueue` operation. To compute the order of a `dqueue` in $D(n)$ 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 25, 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 23), 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. □

Lemma 47 (CASes invoked)**.** *An* `Enqueue()` *or* `Dequeue()` *operation, does at most* $4 \log p$ `CAS` *operations.*

Proof. In each height of the tree at most 2 times `Refresh()` is invoked and every `Refresh()` has 2 `CAS`es, one in Line 313 and one in Lines 318 or 321. □