1 Pseudocode

Algorithm Fields description

Structure

◇ *Shared*

- *Tree* tree : A binary tree of Nodes. root is a pointer to the root node.

◇ *Local*

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

◇ *Structures*

► *Node*

- **Node* left, right, parent : initialized when creating the tree.
- *BlockList* blocks implemented with an array.
- *int* size= 1: #blocks in blocks.
- *int* $\text{num}_{\text{propagated}}$= 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.
- *int[]* super: super[i] stores an approximate index of the superblock of the blocks in blocks whose group field have value i.

► *Leaf* extends *NonRootNode*

- *int* $\text{last}_{\text{done}}$

  Stores the index of the block in the root such that the process that owns this leaf has most recently finished the. A block is finished if all of its operations are finished. enqueue(e) is finished if e is returned by some dequeue() and dequeue() is finished when it computes its response. *put the definitions before the pseudocode*

► *Block* ▷ For a block in a blocklist we define *the prefix for the block* to be the blocks in the BlockList up to and including the block. *put the definitions before the pseudocode*

- *int* group : the value read from $\text{num}_{\text{propagated}}$ when appending this block to the node.

► *LeafBlock* extends *Block*

- *Object* element : Each block in a leaf represents a single operation. For enqueue operations element is the input of the enqueue and for dequeue operations it is null.
- *Object* response : stores the response of the operation in the LeafBlock.
- *int* sum_{enq}, sum_{deq} : # enqueue, dequeue operations in the prefix for the block

► *InternalBlock* extends *Block*

- *int* end_{left}, $\text{end}_{\text{right}}$: index 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* length : length of the queue after performing all operations in the prefix for this block
- *counter* $\text{num}_{\text{finished}}$: number of finished operations in the block
- *int* order : the index of the block in the BlockList containing the block.

Variable naming:

- b_{op}: index of the block containing operationop
- r_{op}: rank of operation op i.e. the ordering among the operations of its type according to linearization ordering

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}, blocks[0].num_{x}=0)

Algorithm *Queue*

```
void Enqueue(Object e)    ▷ Creates a block with element e and appends it to the tree.
    block newBlock= new(LeafBlock)
    newBlock.element= e
    newBlock.sum_enq= leaf.blocks[leaf.size].sum_enq+1
    newBlock.sum_deq= leaf.blocks[leaf.size].sum_deq
    leaf.Append(newBlock)
end Enqueue

Object Dequeue()
    block newBlock= new(LeafBlock)    ▷ Creates a block with null value element, appends it to the tree, computes its order among operations, then computes and returns its response.
    newBlock.element= null
    newBlock.sum_enq= leaf.blocks[leaf.size].sum_enq
    newBlock.sum_deq= leaf.blocks[leaf.size].sum_deq+1
    leaf.Append(newBlock)
    return leaf.HelpDequeue()
end Dequeue

int, int FindResponse(int i, int b)    ▷ Computes the rank and index of the block in the root of the enqueue that is the response of the ith dequeue in the root's bth block. Returns <-1,--> if the queue is empty.
    if root.blocks[b-1].length + root.blocks[b].num_enq - i < 0 then
        return <-1,-->
    else
        ▷ We call the dequeues that return a value non-null dequeues. rth non-null dequeue returns the element of th rth enqueue. We can compute # non-null dequeues in the prefix for a block this way: #non-null dequeues= length - #enqueues. Note that the ith dequeue in the given block is not a non-null dequeue.
        r_enq= root.blocks[b-1].sum_enq- root.blocks[b-1].length + i
        return <root.blocks.get(enq, r_enq).order, r_enq>
    end if
end FindResponse
```

Algorithm *Node*

```
void Propagate()
    if not Refresh() then
        Refresh()                                  ▷ Lemma Double Refresh
    end if
    if this is not root then
        parent.Propagate()
    end if
end Propagate

boolean Refresh()                                  lastLine
    s= size                                        prevLine
    <new, np_left, np_right>= CreateBlock(h)       ▷ np_left, np_right are the values read from the children's num_propagated feild.
    if new.num==0 then return true                 ▷ The block contains nothing.
    else if blocks.tryAppend(new, s) then
okcas   for each dir in {left, right} do
            CAS(dir.super[np_dir], null, h+1)      ▷ Write would work too.
            CAS(dir.num_propagated, np_dir, np_dir+1)
        end for
        CAS(size, h, h+1)
        return true
    else
        CAS(size, h, h+1)                          ▷ Even if another process wins, help to increase the size. The winner might have fallen sleep before increasing size.
        return  false
    end if
end Refresh

⇝ Precondition: blocks[start..end] contains a block with field f ≥ i
int BSearch(field f, int i, int start, int end)
                                                   ▷ Does binary search for the value i of the given prefix sum field. Returns the index of the leftmost block in blocks[start..end] whose field f is ≥ i.
end BSearch

<Block, int, int> CreateBlock(int i)
                                                   ▷ Creates a block to be inserted into as ith block in blocks. Returns the created block as well as values read from each child's num_propagated field. These values are used for incrementing the children's num_propagated field if the block was appended to blocks successfully.
    block newBlock= new(block)
    newBlock.group= num_propagated
    newBlock.order= i
    for each dir in {left, right} do
        index_last= dir.size
        index_prev= blocks[i-1].end_dir
        newBlock.end_dir= index_last
        block_last= dir.blocks[index_last]
        block_prev= dir.blocks[index_prev]
                                                   ▷ newBlock includes dir.blocks[index_prev+1..index_last].
        this_dir= dir.num_propagated
        newBlock.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq - block_prev.sum_enq
        newBlock.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq - block_prev.sum_deq
    end for
    if this is root then
        newBlock.length= max(root.blocks[i-1].length + b.num_enq - b.num_deq, 0)
    end if
    return <b, np_left, np_right>
end CreateBlock
```

Algorithm Node

```
⇝ Precondition: blocks[b].num_enq ≥ i
element GetEnq(int b, int i)
    if this is leaf then
        return blocks[b].element
    else if i ≤ blocks[b].num_enq-left then                          ▷ i exists in the left child of this node
        subBlock= left.BSearch(sum_enq, i, blocks[b-1].end_left+1, blocks[b].end_left)   ▷ Search range of left child's subblocks of blocks[b].
        return left.Get(i-left.blocks[subBlock-1].sum_enq, subBlock)
    else
        i= i-blocks[b].num_enq-left
        subBlock= right.BSearch(sum_enq, i, blocks[b-1].end_right+1, blocks[b].end_right)   ▷ Search range of right child's subblocks of blocks[b].
        return right.Get(i-right.blocks[subBlock-1].sum_enq, subBlock)
    end if
end GetEnq

⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num_enq ≥ i.
<int, int> IndexDeq(int b, int i)     ▷ Returns the rank of ith dequeue in the bth block of the node, among the dequeues in the root.
    if this is root then
        return <b, i>
    else
        dir= (parent.left==n)?  left:  right                         ▷ check if a left or a right child
        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.
    size+=1                                              [appendEnd]
    blk.group= size                                      [appendStart]
    blocks[size]= blk
    parent.Propagate()
end Append

Object HelpDequeue()
    <b_deq, r_deq>= IndexDeq(leaf.size, 1)             ▷ r is the rank among the dequeues of the dequeue of the b_deqth block in the root containing.
    <r_enq, r_enq>= FindResponse(r_deq, b_deq)   ▷ r_enq is the rank of the enqueue whose element is the response to the dequeue in the block containing it and b_deq is the index of that block of it in the blocklist. If the response is null then r_deq is -1.   [deqRest]
    if r_enq==-1 then
        output= null
        root.blocks[b_deq].num_finished.inc()           ▷ shared counter
        if root.blocks[b_deq].num_finished==root.blocks[b_deq].num then
            last_done= b_deq
        end if
    else
        output= GetEnq(b_enq, r_enq)                     ▷ getting the reponse's element.
        root.blocks[b_enq].num_finished.inc()
        root.blocks[b_enq].num_finished.inc()
        if root.blocks[b_deq].num_finished==root.blocks[b_deq].num then
            last_done= b_deq
        else if root.blocks[b_enq].num_finished==root.blocks[b_enq].num then
            last_done= b_enq
        end if
    end if
    return output
end Dequeue
```

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.

◇ *PBRT implementation*

A persistant red-black tree supporting `append(b, key)`,`get(key=i)`,`split(j)`. `append(b, key) returns true in case successful. Since order,` $\texttt{sum}_{\texttt{enq}}$`are both strictly increasing we can use one of them for another.`

`root`: pointer to the root of the PBRT

```
boolean TryAppend(block blk, int n)                ▷ adds block b to the root.blocks[n]
    if root.size%p^2==0 then              ▷ Help every often p^2 operations appended to the root.
        Help()
        CollectGarbage()
    end if
    blk.num_finished= 0
    *oldRoot= &root.blocks.root
    *newRoot= root.blocks.Append(blk).root
    return CAS(root, oldRoot, newRoot)
end TryAppend
```

◇ *Array implementation*

`blocks[]`: array of blocks

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

```
void Help                                    ▷ Helps pending operations
    for leaf l in leaves do
        last= l.size-1    ▷ l.blocks[last] can not be null because size increases after appending, see lines 503-502.
        if l.blocks[last].element==null then      ▷ operation is dequeue
            l.blocks[last].response= l.HelpDequeue()
        end if
    end for
end Help

void CollectGarbage              ▷ Collects the root blocks that are done.
    s=FindMostRecentDone(Root.Blocks.root)    ▷ Lemma: If block b is done after helping then all blocks before b are done as well.
    t1,t2= RBT.split(order, s)
    RBTRoot.CAS(t2.root)
end CollectGarbage
```

```
Block FindMostRecentDone(b)
    for leaf l in leaves do
        max= Max(l.maxOld, max)
    end for
    return max                               ▷ This snapshot suffies.
end FindMostRecentDone

response FallBack(op i) ▷ how to use as exception handling? by adding try catch in all the methods reading the root?
    if root.blocks.get(num_enq), i is null then      ▷ this enqueue was already finished
        return this.leaf.response(block.order)
    end if
end FallBack
```

2 Proof of Linearizability

Definition 1. If `n.blocks[i]==b` we call `i` the *index* of block `b` in node `n`. Block `b` is before block `b'` in node `n` if and only if `b`'s index is smaller than `b'`'s. Block `b` is propagated to node `n` or set `S` if `b` is in `n.blocks` or `S` or is a subblock of a block in `n.blocks` or `S`.

Definition 2. Block `b` in node `n` is in the set *Established(n, t)* if `n.size` is greater than `b`'s index at time t.

ePrgress **Lemma 3** (sizeProgress)**.** `n.size` *is non-decreasing over time.*

dPosition **Lemma 4** (headPosition)**.** *The value read in Line* prevLine *333(*`h=n.size`*) might be 1 bit behind the first empty block in the node.*

shedOrder **Lemma 5** (establishedOrder)**.** *If time* $t <$ *time* t', *then* $Established(n, t) \subseteq Established(n, t')$.

eateBlock **Lemma 6** (createBlock)**.** *Suppose* ***n.CreateBlock(h, x)*** *is invoked at time* t. *The blocks propagated to* $Established(n.left, t)$ *and* $Established(n.right, t)$ *that are not propagated to* $Established(n, t)$*, are subblock of the block returned by* ***CreateBlock(n, h, x)****.*

ueRefresh **Lemma 7** (trueRefresh)**.** *Suppose* ***Refresh(n)****'s* ***CAS(n.blocks[h], null, new)*** *returns* ***true****. Let* t *be the time* ***Refresh(n)*** *is invoked, blocks propgatated to* $Established(n.left, t)$ *and* $Established(n.right, t)$ *are propagated to in* $Established(n, t)$ *after* ***CAS(n.blocks[h], null, new)****.*

seRefresh **Lemma 8** (falseRefresh)**.** *If instance* r *of* ***Refresh(n)*** *returns* ***false****, then there is another successful instance* r' *of* ***Refresh(n)*** *that has performed a successful* ***CAS(n.blocks[h], null, new)****(Line 49) after Line 43(*`h= n.head`*) of* r*.*

leRefresh **Lemma 9** (Double Refresh)**.** *Consider two consecutive instances* r_1,r_2 *of* ***Refresh(n)*** *by the same process (Lines 35,36). Let be the time before* r_1 *invoked. After* r_2*'s* ***CAS*** *all the blocks propagated to* $Established(n.left, t)$ *and* $Established(n.right, t)$ *are in* $Established(n, t)$*.*

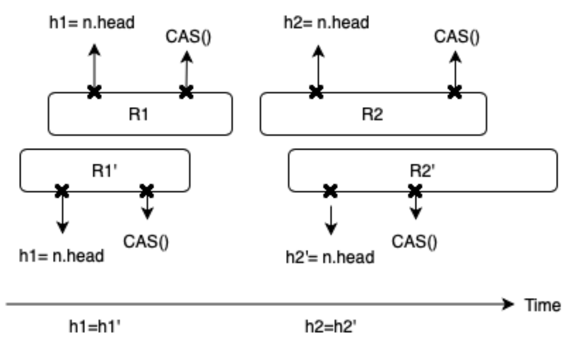

Figure 1: R_2''s `CAS` is executed after `h1=n.head`.

Block `new` is created of new established subblocks of children of `n`(Lemma lem::createBlock 6, Line 46). If `CAS` in Line 48 succeeds then by Lemma lem::trueRefres 7 new established blocks will be in `n`.

lyRefresh **Lemma 10** (Double Refresh)**.** *All operations in* ***n****'s children's blocks before line* ***35*** *are guaranteed to be in* ***n****'s blocks after Line* ***37****.*

`CreateBlock()` reads blocks in the children that do not exist in the parent and aggregates them into one block. If a `Refresh()` procedure returns true it means it has appended the block created by `CreateBlock()` into the parent node's sequence. So suppose two `Refresh`es fail. Since the first `Refresh()` was not successful, it means another CAS operation by a `Refresh`, concurrent to the first `Refresh()`, was successful before the second `Refresh()`. So it means the second failed `Refresh` is concurrent with a successful `Refresh()` that assuredly has read block before the mentioned line `35`. After all it means if any of the `Refresh()` attempts were successful the claim is true, and also if both fail the mentioned claim still holds.

append **Lemma 11** (Append)**.** *When* ***Append(op)*** *is finished,* ***op*** *appears exactly once in a block of* ***root.blocks****.*

blockSize **Lemma 12** (Block Size Upper Bound)**.** *Each block in a node contains at most one operation from each processs.*

ocksBound **Lemma 13** (Subblocks Upperbound)**.** *Each block in a node has at most* p *subblocks.*

ordering **Definition 14** (Ordering of operations inside a node)**.** ▶ Note that from Lemma blockSize 12 we know there is at most one operation from each process in a given block.

- $E(n, i)$ is the sequence of enqueue operations that are member of `n.blocks[i]` ordered by process id.
- $D(n, i)$ is the sequence of dequeue operations that are member of `n.blocks[i]` ordered by process id.
- $D(n) = D(n, 1).D(n, 2).D(n, 3)...$
- $L = E(root, 1).D(root, 1).E(root, 2).D(root, 2).E(root, 3).D(root, 3)...$

Theorem 15. *The queue implementation is linearizable.*

get **Lemma 16** (Get)**.** *`Get(n,b,i)` returns ith Enqueue in $E(n, b)$.*

Lemma 17 (Index)**.** *Index(n,b,i) returns the rank in the $D(root)$ of ith Dequeue in $D(n, b)$.*

uperBlock **Lemma 18** (Computing SuperBlock)**.** *If `Index(n,b,i)` performs line 101, then `superblock` contains `i`th Dequeue in `b`th block of node `n`.*

mputeHead **Lemma 19** (Computing Queue's Head)**.** *Let Q be state of the queue if the operations before ith Dequeue in $L(root)$ are applied on the Queue sequentially and X be the head of Q. If Q is empty `ComputeHead(i,b)` returns -1, otherwise returns index in $E(root, b)$ of X.*

head **Lemma 20** (Validity of `head`)**.** *No two blocks are written in the same index in `n.blocks`.*

erCounter **Lemma 21** (Validity of `super` and `counter`)**.** *If `super[i]` $\neq$ `null`, then `super[i]` in node `n` is the superblock of a block with `time=i`.*

search **Lemma 22** (Search Ranges)**.** *Preconditions of all invocation of `BSearch` are satisfied.*