Algorithm Fields description

◇ *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`
- *int* `size=` 1: #`blocks` in `blocks`.
- *int* $\texttt{num}_{\texttt{propagated}}$= 0 : # groups of blocks that have been propagated from the node to its parent. Since it is incremented after propagating, it may be behind by 1.
- *int[]* `super`: `super[i]` stores an approximate index of the superblock of the blocks in `blocks` whose `group` field have value `i`.

▶ *Root* `extends` *Node*

- *PBRT* `blocks`

  `BlockList` is implemented with a persistent red-black tree.

▶ *NonRootNode* `extends` *Node*

- *Block[]* `blocks`

  `BlockList` is implemented with an array.

▶ *Leaf* `extends` *NonRootNode*

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

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

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

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

▶ *LeafBlock* `extends` *Block*

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

▶ *InternalBlock* `extends` *Block*

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

▶ *RootBlock* `extends` *InternalBlock*

- *int* `length` : length of the queue after performing all operations in the prefix for this block
- *counter* $\texttt{num}_{\texttt{finished}}$: number of finished operations in the block
- *int* `order` : the index of the block in the `BlockList` containing the block.

Variable naming:

- $\texttt{b}_{\texttt{op}}$: index of the block containing operation`op`
- $\texttt{r}_{\texttt{op}}$: rank of operation `op` i.e. the ordering among the operations of its type according to linearization ordering

Abbreviations:

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

  (for b>0 and x $\in$ {$\emptyset$, enq, deq, enq-left, enq-right, deq-left, deq-right}, $\texttt{blocks[0].num}_{\texttt{x}}\texttt{=0}$)

Algorithm *Queue*

201: *void* ENQUEUE(*Object* e) ▷ Creates a block with element e and appends it to the tree.

202: block newBlock= NEW(*LeafBlock*)

203: newBlock.element= e

204: newBlock.sum_{enq}= leaf.blocks[leaf.size].sum_{enq}+1

205: newBlock.sum_{deq}= leaf.blocks[leaf.size].sum_{deq}

206: leaf.APPEND(newBlock)

207: **end** ENQUEUE

208: *Object* DEQUEUE()

209: 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.

210: newBlock.element= null

211: newBlock.sum_{enq}= leaf.blocks[leaf.size].sum_{enq}

212: newBlock.sum_{deq}= leaf.blocks[leaf.size].sum_{deq}+1

213: leaf.APPEND(newBlock)

214: **return** leaf.HELPDEQUEUE()

215: **end** DEQUEUE

216: *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.

217: **if** root.blocks[b-1].length + root.blocks[b].num_{enq} - i $<$ 0 **then**

218: **return** <-1,-->

219: **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.

220: r_{enq}= root.blocks[b-1].sum_{enq}- root.blocks[b-1].length + i

221: **return** <root.blocks.get(enq, r_{enq}).order, r_{enq}>

222: **end if**

223: **end** FINDRESPONSE

Algorithm *Node*

```
301: void PROPAGATE()
302:     if not REFRESH() then
303:         REFRESH()                                   ▷ Lemma Double Refresh
304:     end if
305:     if this is not root then
306:         parent.PROPAGATE()
307:     end if
308: end PROPAGATE

309: boolean REFRESH()                                              lastLine
310:     s= size                                                    prevLine
311:     <new, np_left, np_right>= CREATEBLOCK(h)    ▷ np_left, np_right are the
     values read from the children's num_propagated feild.
312:     if new.num==0 then return true              ▷ The block contains nothing.
313:     else if blocks.tryAppend(new, s) then
314:         for each dir in {left, right} do                                  okcas
315:             CAS(dir.super[np_dir], null, h+1)          ▷ Write would work too.
316:             CAS(dir.num_propagated, np_dir, np_dir+1)
317:         end for
318:         CAS(size, h, h+1)
319:         return true
320:     else
321:         CAS(size, h, h+1)                          ▷ Even if another process
     wins, help to increase the size. The winner might have fallen sleep before
     increasing size.
322:         return false
323:     end if
324: end REFRESH

     ⇝ Precondition: blocks[start..end] contains a block with field f ≥ i
325: 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.
326: end BSEARCH

327: <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.
328:     block newBlock= NEW(block)
329:     newBlock.group= num_propagated
330:     newBlock.order= i
331:     for each dir in {left, right} do
332:         index_last= dir.size
333:         index_prev= blocks[i-1].end_dir
334:         newBlock.end_dir= index_last
335:         block_last= dir.blocks[index_last]
336:         block_prev= dir.blocks[index_prev]
337:                 ▷ newBlock includes dir.blocks[index_prev+1..index_last].
338:         this_dir= dir.num_propagated
339:         newBlock.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq
     - block_prev.sum_enq
340:         newBlock.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq
     - block_prev.sum_deq
341:     end for
342:     if this is root then
343:         newBlock.length= max(root.blocks[i-1].length + b.num_enq -
     b.num_deq, 0)
344:     end if
345:     return <b, np_left, np_right>
346: end CREATEBLOCK
```

Algorithm Node

⇝ Precondition: `blocks[b].num`$_{\texttt{enq}}$$\geq$`i`

```
401: element GetEnq(int b, int i)
402:     if this is leaf then
403:         return blocks[b].element
404:     else if i ≤ blocks[b].num_enq-left then                    ▷ i exists in the left child of this node
405:         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].
406:         return left.Get(i-left.blocks[subBlock-1].sum_enq, subBlock)
407:     else
408:         i= i-blocks[b].num_enq-left
409:         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].
410:         return right.Get(i-right.blocks[subBlock-1].sum_enq, subBlock)
411:     end if
412: end GetEnq
```

⇝ Precondition: bth block of the node has propagated up to the root and `blocks[b].num`$_{\texttt{enq}}$$\geq$`i`.

```
413: <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.
414:     if this is root then
415:         return <b, i>
416:     else
417:         dir= (parent.left==n)?  left:  right                    ▷ check if a left or a right child
418:         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[].
419:         if dir is right then
420:             i+= blocks[superBlock].sum_deq-left                ▷ consider the dequeues from the right child
421:         end if
422:         return this.parent.IndexDeq(superBlock, i)
423:     end if
424: end Index
```

Algorithm Leaf

```
501: void APPEND(block blk)                                   ▷ Append is only called by the owner of the leaf.
appendEnd
502:     size+=1
pendStart
503:     blk.group= size
504:     blocks[size]= blk
505:     parent.PROPAGATE()
506: end APPEND

507: Object HELPDEQUEUE()
508:     <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.
509:     <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
deqRest      b_deq is the index of that block of it in the blocklist. If the response is null then r_deq is -1.
510:     if r_enq==-1 then
511:         output= null
512:         root.blocks[b_deq].num_finished.inc()                                   ▷ shared counter
513:         if root.blocks[b_deq].num_finished==root.blocks[b_deq].num then
514:             last_done= b_deq
515:         end if
516:     else
517:         output= GETENQ(b_enq, r_enq)                          ▷ getting the reponse's element.
518:         root.blocks[b_enq].num_finished.inc()
519:         root.blocks[b_enq].num_finished.inc()
520:         if root.blocks[b_deq].num_finished==root.blocks[b_deq].num then
521:             last_done= b_deq
522:         else if root.blocks[b_enq].num_finished==root.blocks[b_enq].num then
523:             last_done= b_enq
524:         end if
525:     end if
526:     return output
527: 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 end_{left}, $\text{end}_{\text{right}}$ 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, $\text{sum}_{\text{enq}}$are both strictly increasing we can use one of them for another.

`root`: pointer to the root of the PBRT

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

♢ *Array implementation*

`blocks[]`: array of blocks

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

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

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

814: Block FindMostRecentDone(b)
815:     for leaf l in leaves do
816:         max= Max(l.maxOld, max)
817:     end for
818:     return max                                         ▷ This snapshot suffies.
819: end FindMostRecentDone

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