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* $\texttt{num}_{\texttt{propagated}}$= 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 the index of the superblock of some block in `blocks` whose `group` field is `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.
- *int* `size`= 1: #`blocks` in `blocks`.

▶ *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: <r_{deq}, b_{deq}>= leaf.INDEXDEQ(leaf.size, 1)
▷ r is the rank among all dequeues of the dequeue and b_{deq} is the index of the block in the root containing the dequeue.
215: <r_{enq}, $b_{r_{enq}}$>= FINDRESPONSE(r_{deq}, b_{deq})
▷ r_{enq} is the rank of the enqueue whose element is the response to the dequeue and b_{deq} is the index of the block of it in the blocklist. If the response is null then r_{deq} is -1.

deqRest

216: **if** r_{enq}==-1 **then**
217: output= null
218: root.blocks[b_{deq}].$num_{finished}$.inc() ▷ shared counter
219: **if** root.blocks[b_{deq}].$num_{finished}$==root.blocks[b_{deq}].num **then**
220: leaf.$last_{done}$= b_{deq}
221: **end if**
222: **else**
223: output= GETENQ(r_{enq}, $b_{r_{enq}}$) ▷ getting the reponse's element.
224: root.blocks[$b_{r_{enq}}$].$num_{finished}$.inc()
225: root.blocks[$b_{r_{enq}}$].$num_{finished}$.inc()
226: **if** root.blocks[b_{deq}].$num_{finished}$==root.blocks[b_{deq}].num **then**
227: leaf.$last_{done}$= b_{deq}
228: **else if** root.blocks[b_r].$num_{finished}$==root.blocks[b_r].num **then**
229: leaf.$last_{done}$= $b_{r_{enq}}$
230: **end if**
231: **end if**
232: **return** output
233: **end** DEQUEUE

234: *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.
235: **if** root.blocks[b-1].length + root.blocks[b].num_{enq} - i $<$ 0 **then**
236: **return** <-1,-->
237: **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.
238: r_{enq}= root.blocks[b-1].sum_{enq}- root.blocks[b-1].length + i
239: **return** <r_{enq}, root.blocks.get(enq, r_{enq}).order>
240: **end if**
241: **end** FINDRESPONSE

Algorithm *Node*

```
301: void PROPAGATE()
302:     if not this.REFRESH() then
303:         this.REFRESH()                          ▷ Lemma Double Refresh
304:     end if
305:     if this is not root then    ▷ To check a node is the root we can check
     its index is 0.
306:         this.parent.PROPAGATE()
307:     end if
308: end PROPAGATE

309: boolean REFRESH()
310:     h= size
311:     <new, np_left, np_right>= this.CREATEBLOCK(h)    ▷ np_left, np_right are
     the values read from the children's num_propagateds.
312:     if new.num==0 then return true          ▷ The block contains nothing.
313:     else if root.blocks.tryAppend(new) then
314:         for each dir in {left, right} do
315:             CAS(dir.super[np_dir], null, h+1)
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. It might fell sleep before increasing.
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 feild. 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 this.blocks[i]. Returns
     the created block as well as values read from each childnum_propagated field.
     The values are used for incrementing children's num_propagateds if the block
     was appended to this.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:         block_last= dir.blocks[index_last]
335:         block_prev= dir.blocks[index_prev]
336:                 ▷ newBlock includes dir.blocks[index_prev+1..index_last].
337:         n_dir= dir.num_propagated
338:         newBlock.end_dir= index_last
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: `n.blocks[b]` contains `i`th enqueue in the node.

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

⇝ Precondition: `b`th block of the node has propagated up to the root and `i`th dequeue in the node is in `blocks[b]`.

```
411: <int, int> IndexDeq(int b, int i)   ▷ Returns the order in the root of ith dequeue in the bth block of node n among dequeues.
412:     dir= (parent.left==n)?  left:  right                        ▷ check if a left or a right child
413:     superBlock= BSearch(parent, 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[].
414:     if dir is left then
415:         i+= parent.blocks[superBlock-1].sum_deq-right
416:     else
417:         i+= parent.blocks[superBlock-1].sum_deq + blocks[superBlock].sum_deq-left   ▷ consider dequeues from the right child
418:     end if
419:     return IndexDeq(this.parent, superBlock, i)
420: end Index
```

Algorithm Leaf

```
501: void Append(block blk)
502:     size+=1                                  ▷ Lines 503 to 502 are done by one process at time.
503:     blk.group= size                          ▷ Append is only called by the owener of the leaf.
504:     blocks[size]= blk
505:     parent.Propagate()
506: end Append

507: element GetEnq(int b, int i)
508:     return blocks[b].element
509: end GetEnq
```

Algorithm Root

```
601: <int, int> IndexDeq(int b, int i)
602:     return <i, b>
603: end IndexDeq
```

Algorithm BlockList

▷ : Supports two operations `blocks.tryAppend(Block b)`, `blocks[i]`. Initially empty, when `blocks.tryAppend(b)` returns true `b` is appended to the end of the list and `blocks[i]` returns ith block in the `blocks`. If some instance of `blocks.tryAppend(b)` returns `false` there is a concurrent instance of `blocks.tryAppend(b′)` which has returned `true`.`blocks[0]` contains an empty block with all fields equal to 0 and $\text{end}_{\texttt{left}}$, $\text{end}_{\texttt{right}}$ pointers to the first block of the corresponding children.

◇ *PBRT implementation*

```
701: boolean TryAppend(block blk)
702:     TODO
703: end TryAppend
```

◇ *Array implementation*

```
704: boolean TryAppend(block blk)
705:     TODO
706: end TryAppend
```

► *PRBTree[rootBlock]*

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.

```
801: void RBTAppend(block b)                ▷ adds block b to the root.blocks
802:     step= root.size
803:     if step%p^2==0 then  ▷ Help every often p^2 operations appended to the root. Used in lemma's using the size of the PBRT.
804:         Help()
805:         CollectGarbage()
806:     end if
807:     b.num_finished= 0
808:     return root.blocks.append(b, b.order)
809: end RBTAppend

810: void Help                              ▷ Helps pending operations
811:     for leaf l in leaves do ▷ if the tree is implemented with an array we can iterate over the second half of the array.
812:         last= l.size-1      ▷ l.blocks[last] can not be null because of lines 503-502.
813:         if l.blocks[last].element==null then      ▷ operation is dequeue
814:             goto 215 with these values <>          ▷ run Dequeue() for l.ops[last] after Propagate(). TODO
815:             l.responses[last]= response
816:         end if
817:     end for
818: end Help

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

824: Block FindMostRecentDone(b)
825:     for leaf l in leaves do
826:         max= Max(l.maxOld, max)
827:     end for
828:     return max                             ▷ This snapshot suffies.
829: end findYoungestOld

830: response FallBack(op i)                   ▷ really necessary?
831:     if root.blocks.get(num_enq), i is null then     ▷ this enqueue was already finished
832:         return this.leaf.response(block.order)
833:     end if
834: end FallBack
```