Algorithm Fields description

♢ *Shared*

- *Node** `tree[]` : A binary tree of Nodes such that `Tree[0]` is the root and the left child and the right child and the parent of `Tree[i]` are `Tree[2i+1]`, `Tree[2i+2]` and `Tree[i/2]`.

♢ *Local*

- *int* `leaf`: `tree[leaf]` is the process's leaf in the tree.

♢ *Structures*

▶ *Node*

- *BlockList* `blocks`: 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 `end` pointers to the first block of the corresponding children.
- *int* $\texttt{num}_{\texttt{propagated}}$= 0 : # groups of blocks have been propagated from the node. It may be behind its real value.
- *int[]* `super`: `super[i]` stores the index of the superblock of some block in `blocks` that its `group` field is `i`.

▶ *Root* `extends` *Node*

- *PBRT* `blocks`

  Implemented with a persistent red-black tree.

▶ *NonRootNode* `extends` *Node*

- *Block[]* `blocks`

  Implemented with an array and using CAS for appending to the head.
- *int* `head`= 1: #`blocks` in the `blocks`.

▶ *Leaf* `extends` *NonRootNode*

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

  Each process stores the index of the most recent block in the root that the process has finished the last operation of the block. `enqueue(e)` is finished if `e` is returned by some `dequeue()` and `dequeue()` is finished when it computes its response.

▶ *Block b* ▷ If `b` is `blocks[i](i!=0)` then `b[-1]` is `blocks[i-1]`.

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

▶ *LeafBlock* `extends` *Block*

- *Object* `element` : Each block in a leaf represents an operation. The `element` shows the operation's argument if it is an enqueue, and if it is a dequeue this value is `null`.
- *int* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: number of enqueue, dequeue operations in the leaf's containing this block till this block
- *Object* `response`

  response stores the response of the operation in the `LeafBlock`.

▶ *InternalBlock* `extends` *Block*

- *int* $\texttt{sum}_{\texttt{enq-left}}$: #enqueues from the subblocks in the left child + $\texttt{b[-1].sum}_{\texttt{enq-left}}$
- *int* $\texttt{sum}_{\texttt{deq-left}}$: #dequeues from the subblocks in the left child + $\texttt{b[-1].sum}_{\texttt{deq-left}}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: #enqueues from the subblocks in the right child + $\texttt{b[-1].sum}_{\texttt{enq-right}}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: #dequeues from the subblocks in the right child + $\texttt{b[-1].sum}_{\texttt{deq-right}}$
- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: index of the last subblock of the block in the left and right child

▶ *RootBlock* `extends` *InternalBlock*

- *int* `size` : size of the queue after this block's operations finish
- *counter* $\texttt{num}_{\texttt{finished}}$: number of finished operations in the block
- *int* `order` : the index of the block in the node containing the block. Useful in the root since in the PBRT we do not keep indices as key.

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}= tree[leaf].blocks[tree[leaf].head].sum_{enq}+1
205: newBlock.sum_{deq}= tree[leaf].blocks[tree[leaf].head].sum_{deq}
206: tree[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 its response; if it exists returns the response's element.
210: newBlock.element= null
211: newBlock.sum_{enq}= tree[leaf].blocks[tree[leaf].head].sum_{enq}
212: newBlock.sum_{deq}= tree[leaf].blocks[tree[leaf].head].sum_{deq}+1
213: tree[leaf].APPEND(newBlock)
214: <r_{deq}, $b_{r_{deq}}$>= tree[leaf].INDEX(tree[leaf].head, 1) ▷ r is the rank among all dequeues of the dequeue and $b_{r_{deq}}$ is the index of the block in the root containing the dequeue.
215: <r_{enq}, $b_{r_{enq}}$>= FINDRESPONSE(r_{deq}, $b_{r_{deq}}$) ▷ r_{deq} is the rank of the enqueue which is the response to the dequeue or -1 if the response is null.

deqRest

216: **if** r_{enq}==-1 **then**
217: output= null
218: tree[0].blocks[$b_{r_{deq}}$].$num_{finished}$.inc() ▷ shared counter
219: **if** tree[0].blocks[$b_{r_{deq}}$].$num_{finished}$==tree[0].blocks[$b_{r_{deq}}$].num **then** ▷ all the operations in the block containing the dequeue are finished.
220: tree[leaf].$last_{done}$= $b_{r_{deq}}$
221: **end if**
222: **else**
223: output= GET(index$_{response}$)
224: root.blocks[$b_{r_{enq}}$].$num_{finished}$.inc()
225: root.blocks[$b_{r_{enq}}$].$num_{finished}$.inc()
226: **if** root.blocks[b_r].$num_{finished}$==root.blocks[b_r].num **then**
227: tree[leaf].$last_{done}$= $b_{r_{enq}}$
228: **else if** root.blocks[$b_{r_{deq}}$].$num_{finished}$==root.blocks[$b_{r_{deq}}$].num **then** ▷ root.blocks[b_r] comes after root.blocks[b_i].
229: tree[leaf].$last_{done}$= $b_{r_{deq}}$
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 ¡-1,0¿ if the queue is empty.
235: **if** root.blocks[b-1].size + root.blocks[b].num_{enq} - i < 0 **then**
236: **return** -1, 0
237: **else**
238: res= root.blocks[b-1].sum_{deq}- root.blocks[b-1].size + i
239: **return** <res, tree[0].blocks.get(enq, res).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= head
311:     <new, n_left, n_right>= this.CREATEBLOCK(h)    ▷ n_left, n_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[n_dir], null, h+1)
316:             CAS(dir.num_propagated, n_dir, n_dir+1)
317:         end for
318:         CAS(head, h, h+1)
319:         return true
320:     else
321:         CAS(head, h, h+1)          ▷ Even if another process wins, help to
     increase the head. 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= n.dir.head
333:         index_prev= n.blocks[i-1].end_dir
334:         block_last= n.dir.blocks[index_last]
335:         block_prev= n.dir.blocks[index_prev]
336:               ▷ newBlock includes n.dir.blocks[index_prev+1..index_last].
337:         n_dir= n.dir.num_propagated
338:         newBlock.end_dir= index_last
339:         newBlock.num_enq-dir= block_last.sum_enq - block_prev.sum_enq
340:         newBlock.num_deq-dir= block_last.sum_deq - block_prev.sum_deq
341:         newBlock.sum_enq-dir= n.blocks[i-1].sum_enq-dir + b.num_enq-dir
342:         newBlock.sum_deq-dir= n.blocks[i-1].sum_deq-dir + b.num_deq-dir
343:     end for
344:     if n is root then
345:         newBlock.size= max(root.blocks[i-1].size + b.num_enq -
     b.num_deq, 0)
346:     end if
347:     return b, c_left, c_right
348: end CREATEBLOCK
```

lastLine

prevLine

okcas

Algorithm Node

⇝ Precondition: `n.blocks[b]` contains `i`th enqueue in the node.

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

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

```
414: <int, int> INDEX(int b, int i)   ▷ Returns the order in the root of ith dequeue in the bth block of node n among dequeues.
415:     if this is root then return <i, b>
416:     else
417:         dir= (this.parent().left==n)?  left:  right                ▷ check n is a left or a right child
418:         superBlock= BSEARCH(this.parent(), this.sum_deq-dir, i, super[this.blocks[b].group]-p, super[this.blocks[b].group]+p)   ▷ superblock's
```

group has at most p difference with the value stored in `super[]`.

```
419:         if dir is left then
420:             i+= this.parent().blocks[superBlock-1].sum_deq-right
421:         else
422:             i+= this.parent.blocks[superBlock-1].sum_deq + this.blocks[superBlock].sum_deq-left   ▷ consider dequeues from the right child
423:         end if
424:         return INDEX(this.parent(), superBlock, i)
425:     end if
426: end INDEX
```

Algorithm Leaf

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

appendEnd

pendStart

Algorithm Root

```
601: element GET(int i)                                   ▷ Returns ith Enqueue.
602:     res= root.blocks.get(enq, i).order
603:     return GET(root, res, i-root.blocks[res-1].sum_enq)
604: end GET
```

▶ *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.

```
701: void RBTAppend(block b)                    ▷ adds block b to the root.blocks
702:     step= root.head
703:     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.
704:         Help()
705:         CollectGarbage()
706:     end if
707:     b.num_finished= 0
708:     return root.blocks.append(b, b.order)
709: end RBTAppend

710: void Help                                  ▷ Helps pending operations
711:     for leaf l in leaves do ▷ if the tree is implemented with an array we
         can iterate over the second half of the array.
712:         last= l.head-1     ▷ l.blocks[last] can not be null because of
         lines 503-502.
713:         if l.blocks[last].element==null then     ▷ operation is dequeue
714:             goto 215 with these values <>        ▷ run Dequeue() for
         l.ops[last] after Propagate(). TODO
715:             l.responses[last]= response
716:         end if
717:     end for
718: end Help

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

724: Block FindMostRecentDone(b)
725:     for leaf l in leaves do
726:         max= Max(l.maxOld, max)
727:     end for
728:     return max                                 ▷ This snapshot suffies.
729: end findYoungestOld

730: response FallBack(op i)                       ▷ really necessary?
731:     if root.blocks.get(num_enq), i is null then     ▷ this enqueue was already
         finished
732:         return this.leaf.response(block.order)
733:     end if
734: end FallBack
```