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* counter= 0 # groups 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 n.counter 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* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: sum of # enqueue, dequeue operations in the leaf's blocks up to this one
- *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 + b[-1].$\texttt{sum}_{\texttt{enq-left}}$
- *int* $\texttt{sum}_{\texttt{deq-left}}$: #dequeues from the subblocks in the left child + b[-1].$\texttt{sum}_{\texttt{deq-left}}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: #enqueues from the subblocks in the right child + b[-1].$\texttt{sum}_{\texttt{enq-right}}$
- *int* $\texttt{sum}_{\texttt{enq-right}}$: #dequeues from the subblocks in the right child + b[-1].$\texttt{sum}_{\texttt{deq-right}}$
- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: index of the last subblock in the left and right child

▶ *RootBlock* extends *InternalBlock*

- *int* size : size of queue after this block's operations finish
- *int* $\texttt{sum}_{\texttt{non-null deq}}$: count of non-null dequeus up to this block
- *counter* $\texttt{num}_{\texttt{finished}}$: number of finished operations in the block
- *int* order : the index of the block in the node containing it. Useful in the root since in the PBRT we do not keep indices in another way.

▶ Conventions

- i : index of ith operation in the tree
- j : index of jth operation in a node
- $\texttt{b}_{\texttt{n}}$: index of the block containing the operation n based on the scope
- Also we are not going to refer to blocks directly, only with their indices. Except while constructing a new block.

Algorithm *Queue*

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

Object Dequeue()
    block newBlock= new(LeafBlock)    ▷ Creates a null element block, appends it to the tree, computes its order among operations, then computes its response index if it exists and returns the response's element.
    newBlock.element= null
    newBlock.sum_enq= tree[leaf].blocks[tree[leaf].head].sum_enq
    newBlock.sum_deq= tree[leaf].blocks[tree[leaf].head].sum_deq+1
    tree[leaf].Append(newBlock)
    <i, b_i>= tree[leaf].Index(tree[leaf].head, 1)    ▷ i is the rank among all dequeues of the dequeue in the root and b_i is the index of the block in the root containing it.
    index_response= ComputeDeqRes(i, b_i)    ▷ index_response is the index of the enqueue which is the response to the dequeue or -1 if the response is null.
deqRest
    if index_response==-1 then
        output= null
        tree[0].blocks[b_i].num_finished.inc()    ▷ shared counter
        if tree[0].blocks[b_i].num_finished==tree[0].blocks[b_i].num then    ▷ all the operations in the block containing the dequeue are finished.
            tree[leaf].last_done= b_i
        end if
    else
        output= Get(index_response)
        b_r= tree[0].blocks.get(enq, index_response).order    ▷ index of the block in the root that contains response enqueue.
        root.blocks[b_r].num_finished.inc()
        root.blocks[b_r].num_finished.inc()
        if root.blocks[b_r].num_finished==root.blocks[b_r].num then    ▷ become done
            tree[leaf].last_done= b_r
        else if root.blocks[b_r].num_finished==root.blocks[b_r].num then    ▷ root.blocks[b_r] comes after root.blocks[b_i].
            this.leaf.last_done= b_i
        end if
    end if
    return output
end Dequeue

int ComputeDeqRes(int i, int b)    ▷ Computes the response of the ith dequeue in the root's bth block. Returns the index of the the head of the queue or -1 if queue is empty.
    if root.blocks[b-1].size + root.blocks[b].num_enq - i < 0 then
        return -1
    else return root.blocks[b-1].sum_non-null deq + i
    end if
end ComputeDeqRes
```

Algorithm *Node*

void PROPAGATE
    if not this.REFRESH **then**
        this.REFRESH(this) ▷ Lemma Double Refresh
    end if
    if this **is not** root **then** ▷ To check a node is the root we can check its index is 0.
        this.parent().PROPAGATE()
    end if
end PROPAGATE

boolean REFRESH
    h= head
    c= counter lastLine
    <new, c_{left}, c_{right}>= CREATEBLOCK(n, h) ▷ c_{left}, c_{right} are the prevLine values read from n's children's counters.
    new.group= c
    if new.num==0 **then return** true ▷ The block contains nothing.
    else if root.blocks.tryAppend(new) **then** ▷ *how to put space in he first of the new line?*
okcas   **for each** dir **in** {left, right} **do**
            CAS(dir.super[c_{dir}], null, h+1) ▷ Superblock's Lemma
            CAS(dir.counter, c_{dir}, c_{dir}+1)
        end for
        CAS(head, h, h+1)
        return true
    else
        CAS(head, h, h+1) ▷ Even if another process wins, help to increase the head. It might fell sleep before increasing.
        return false
    end if
end REFRESH

⇝ Precondition: n.blocks[start..end] contains a block with field f $\geq$ i
int BSEARCH(*node* n, *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 n.blocks[start..end] whose *field* f is $\geq$ i.
end BSEARCH

<*Block, int, int*> CREATEBLOCK(*node* n, *int* i)
▷ Creates a block to insert into n.blocks[i]. Returns the created block as well as values read from each child counter field. The values are used for incrementing children's counters if the block was appended to n.blocks successfully. *Does it need help? I think no but in that case we do not have to pass these values to the calling line.*
    block b= NEW(*block*)
    if n **is** root **then**
        b= NEW(*root block*)
    end if
    b.order= i
    for each dir **in** {left, right} **do**
        $index_{last}$= n.dir.head
        $index_{prev}$= n.blocks[i-1].end_{dir}
        $block_{last}$= n.dir.blocks[$index_{last}$]
        $block_{prev}$= n.dir.blocks[$index_{prev}$] ▷ n.dir.blocks[$index_{prev}$..$index_{last}$] are merged to one block.
        c_{dir}= n.dir.counter
        b.end_{dir}= $index_{last}$
        b.$num_{enq-dir}$= $block_{last}$.sum_{enq} - $block_{prev}$.sum_{enq}
        b.$num_{deq-dir}$= $block_{last}$.sum_{deq} - $block_{prev}$.sum_{deq}
        b.$sum_{enq-dir}$= n.blocks[i-1].$sum_{enq-dir}$ + b.$num_{enq-dir}$
        b.$sum_{deq-dir}$= n.blocks[i-1].$sum_{deq-dir}$ + b.$num_{deq-dir}$
    end for
    b.num_{enq}= b.$num_{enq-left}$ + b.$num_{enq-right}$
    b.num_{deq}= b.$num_{deq-left}$ + b.$num_{deq-right}$
    b.num= b.num_{enq} + b.num_{deq}
    b.sum= n.blocks[i-1].sum + b.num
    if n **is** root **then**
        b.size= max(root.blocks[i-1].size + b.num_{enq} - b.num_{deq}, 0)
        b.$sum_{non-null\ deq}$= root.blocks[i-1].$sum_{non-null\ deq}$ + max(b.num_{deq} - root.blocks[i-1].size - b.num_{enq}, 0)
    end if
    return b, c_{left}, c_{right}
end CREATEBLOCK

Algorithm Node

```
    ⇝ Precondition: n.blocks[b] contains ≥i enqueues.
401: element GET(node n, int b, int i)        ▷ Returns the ith Enqueue in bth block of node n
402:     if n is leaf then return n.blocks[b].element
403:     else
404:         if i ≤ n.blocks[b].num_enq-left then        ▷ i exists in the left child of n
405:             subBlock= BSEARCH(n.left, sum_enq, i, n.blocks[b-1].end_left+1, n.blocks[b].end_left)
406:             return GET(n.left, subBlock, i-n.left.blocks[subBlock-1].sum_enq)
407:         else
408:             i= i-n.blocks[b].num_enq-left
409:             subBlock=BSEARCH(n.right, sum_enq, i, n.blocks[b-1].end_right+1, n.blocks[b].end_right)
410:             return GET(n.right, subBlock, i-n.right.blocks[subBlock-1].sum_enq)
411:         end if
412:     end if
413: end GET

    ⇝ Precondition: bth block of node n has propagated up to the root and ith dequeue resides in node n is in block b of node n.
414: <int, int> INDEX(node n, int b, int i)        ▷ Returns the order in the root of ith dequeue in the bth block of node n among dequeues.
415:     if n is root then return i, b
416:     else
417:         dir= (n.parent.left==n)? left: right        ▷ check n is a left or a right child
418:         superBlock= BSEARCH(n.parent, n.sum_deq-dir, i, super[n.blocks[b].group]-p, super[n.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+= n.parent.blocks[superBlock-1].sum_deq-right
421:         else
422:             i+= n.parent.blocks[superBlock-1].sum_deq + n.blocks[superBlock].sum_deq-left        ▷ consider dequeues from n's right child
423:         end if
424:         return INDEX(n.parent, superBlock, i)
425:     end if
426: end INDEX
```

Algorithm Leaf

```
                                                  appendStart appendEnd
     501: void APPEND(block b)
appendEnd   502:     head+=1                      ▷ Lines 503 to 502 are done by one process at time.
appendStart 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
```

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, sum_enq are
both strictly increasing we can use one of them for another.
void RBTAppend(block b)                          ▷ adds block b to the root.blocks
    step= root.head
    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.
        Help()
        CollectGarbage()
    end if
    b.num_finished= 0
    return root.blocks.append(b, b.order)
end RBTAppend

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

response FallBack(op i)                          ▷ really necessary?
    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
```