algQ **Algorithm** Queue

◇ *Local*

- **Node* `leaf`: pointer the the process's leaf in the tree

◇ *Shared*

- *Tree* : A binary tree of `Nodes` is shared among the processes. It can be implemented with a 1 index based array of size p. Such that the root is index 1, the left child and the right child of a node with index i are indices 2i, 2i+1 in the array.

◇ *Structures*

▶ *Node*

- **Node* `left, right, parent`
- *Block[]* `blocks`: index 0 contains an empty block with all fields equal to 0 and en pointers to the first block of the corresponding children. `blocks[i]` returns the ith block stored. In the root node it is implemented with a persistent red-black tree and it is a big array in the other nodes.
- *int* `head= 1`: index of the first empty cell of `blocks`
- *int* `counter= 0`
- *int[]* `super`: `super[i]` stores the index of a superblock in parent that contains some block of this node whose `time` is field `i`

▶ *leaf* extends `Node`

- *int[]* `response`
  leaf.response[i] stores response of leaf.ops[i]
- *int* $\texttt{last}_{\texttt{done}}$
  Each process stores the index of the most recent block that the process has finished its last operation. An enqueue operation is finished if it has appended its element to the root and a dequeue operation is finished when it computes its response.

▶ *Block*

- *int* $\texttt{num}_{\texttt{enq-left}}$, $\texttt{sum}_{\texttt{enq-left}}$: #enqueues from subblocks in left child, prefix sum of $\texttt{num}_{\texttt{enq-left}}$
- *int* $\texttt{num}_{\texttt{deq-left}}$, $\texttt{sum}_{\texttt{deq-left}}$: #dequeues from subblocks in left child, prefix sum of $\texttt{num}_{\texttt{deq-left}}$
- *int* $\texttt{num}_{\texttt{enq-right}}$, $\texttt{sum}_{\texttt{enq-right}}$: #enqueues from subblocks in right child, prefix sum of $\texttt{num}_{\texttt{enq-right}}$
- *int* $\texttt{num}_{\texttt{deq-right}}$, $\texttt{sum}_{\texttt{deq-right}}$: #dequeues from subblocks in right child, prefix sum of $\texttt{num}_{\texttt{deq-right}}$
- *int* $\texttt{num}_{\texttt{enq}}$, $\texttt{num}_{\texttt{deq}}$: # enqueue, dequeue operations in the block
- *int* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: sum of # enqueue, dequeue operations in blocks up to this one
- *int* `num, sum` : total # operations in block, prefix sum of `num`
- *int* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: index of the last subblock in the left and right child
- *int* `group` : id of the group of blocks including this propagated together, more precisely the value read from the node `n`'s counter when propagating this block to the node `n`.
- *int* `order` : the index of the block in the node containing it

▶ *Leaf Block* 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`.

▶ *Root Block* extends *Block*

- *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
- *int* $\texttt{num}_{\texttt{done}}$: number of finished operations in the block

```
void Enqueue(Object e)
    block b= new(block)
    b.element= e
    b.num_enq=1
    b.sum_enq= this.blocks[leaf.head].sum_enq+1
    Append(b)
end Enqueue

Object Dequeue()
    block b= new(block)
    b.element= null
    b.num_deq=1
    b.sum_deq= this.blocks[leaf.head].sum_deq+1
    Append(b)
    <i, b_i>= Index(this.leaf, this.leaf.head, 1)    ▷ i is the order in the root among all dequeues, of the dequeue in the last block in the process's leaf. b_i is the block in the root containing it.
    index_response= ComputeDeqRes(i, b)    ▷ index_response is the index of the enqueue which is the response to the dequeue or -1.
    if index_response!=-1 then
        output= null
        b_i.num_done= b_i.num_done+1
        if b_r.num_done==b_r.num then    ▷ become old
            this.leaf.last_done= b_r
        end if
    else
        output= Get(res)
        b_r= root.blocks.get(enq, index_response)    ▷ block in the root contains response enqueue.
        b_i.num_done= b_i.num_done+1
        b_r.num_done= b_r.num_done+1
        if b_r.num_done==b_r.num then    ▷ become old
            this.leaf.last_done= b_r
        else if b_i.num_done==b_i.num then
            this.leaf.last_done= b_i
        end if
    end if
    return output
end Dequeue
```

deqRest

```
int ComputeDeqRes(int i, int b)    ▷ Computes head of the queue when ith dequeue in bth block occurs. The dequeue should return the argument of the head enqueue.
    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

void Append(block b)
    b.group= this.leaf.head
    l_pid.blocks[this.leaf.head]= b
    this.leaf.head+=1
    Propagate(this.leaf.parent)
end Append

void Propagate(node n)
    if not Refresh(n) then
        Refresh(n)
    end if
    if n.parent is not null then
        Propagate(n.parent)
    end if
end Propagate

boolean Refresh(node n)
    h= n.head
    c= n.counter
    <new, c_left, c_right>= CreateBlock(n, h)
    new.group= c
    if new.num==0 then return true                ▷ The block contains nothing.
    else if (n is root and root.blocks.append(new)) or
(n is not root and CAS(n.blocks[h], null, new)) then    ▷ space in he first of the new line?
okcas   for each dir in {left, right} do
            CAS(n.dir.super[c_dir], null, h+1)
            CAS(n.dir.counter, c_dir, c_dir+1)
        end for
        CAS(n.head, h, h+1)
        return true
    else
        CAS(n.head, h, h+1)
        return  false
    end if
end Refresh

element Get(int i)                                ▷ Returns ith Enqueue.
    if i is null then
        return null
    end if
    res= root.blocks.get(enq, i).order
    return Get(root, res, i-root.blocks[res-1].sum_enq)
end Get

⇝ Precondition: n.blocks[start..end] contains a block with field f ≥ 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 ≥ 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 feild.
    block b= new(block)
    if n is root then
        block b= new(root block)
    end if
    b.order= i
    for each dir in {left, right} do
lastLine    lastIndex= n.dir.head
prevLine    prevIndex= n.blocks[i-1].end_dir
        lastBlock= n.dir.blocks[lastIndex]
        prevBlock= n.dir.blocks[prevIndex]
        c_dir= n.dir.counter
        b.end_dir= lastIndex
        b.num_enq-dir= lastBlock.sum_enq - prevBlock.sum_enq
        b.num_deq-dir= lastBlock.sum_deq - prevBlock.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.parent is null 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
```

⇝ Precondition: n.blocks[b] contains ≥i enqueues.

```
element Get(node n, int b, int i)                    ▷ Returns the ith Enqueue in bth block of node n
    if n is leaf then return n.blocks[b].element
    else
        if i ≤ n.blocks[b].num_enq-left then                          ▷ i exists in the left child of n
            subBlock= BSearch(n.left, sum_enq, i, n.blocks[b-1].end_left+1, n.blocks[b].end_left)
            return Get(n.left, subBlock, i-n.left.blocks[subBlock-1].sum_enq)
        else
            i= i-n.blocks[b].num_enq-left
            subBlock=BSearch(n.right, sum_enq, i, n.blocks[b-1].end_right+1, n.blocks[b].end_right)
            return Get(n.right, subBlock, i-n.right.blocks[subBlock-1].sum_enq)
        end if
    end if
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.

```
<int, int> Index(node n, int b, int i)      ▷ Returns the order in the root among dequeus, of ith dequeue in bth block of node n.
    if n is root then return root.blocks.get(order==b-1).sum_deq+i, b
    else
        dir= (n.parent.left==n)?  left:  right
        superBlock= BSearch(n.parent, n.sum_deq-dir, i, super[n.blocks[b].group]-p, super[n.blocks[b].group]+p)
        if dir is left then
            i+= n.parent.blocks[superBlock-1].sum_deq-right
        else
            i+= n.parent.blocks[superBlock-1].sum_deq + n.blocks[superBlock].sum_deq-left
        end if
        return Index(n.parent, superBlock, i)
    end if
end Index
```

► *PRBTree[rootBlock]*

A persistant red-black tree supporting append(b, key),get(key=i),split(j).

append(b, key) returns true in case successful.

```
void RBTAppend(block b)                          ▷ adds block b to the root.blocks
    step= root.head
    if step%p^2==0 then
        Help()
        CollectGarbage()
    end if
    b.age= 0
    return root.blocks.append(b, b.order)
end RBTAppend

void Help                                        ▷ Helps pending operations
    for leaf l in leaves do                      ▷ how to iterate over them?
        last= l.head-1
        if l.blocks[last] is not null then
            if l.blocks[last].element==null then     ▷ operation is dequeue
                goto deqRest 15 with these values <>     ▷ run Dequeue() for l.ops[last] after Propagate(). TODO
                l.responses[last]= response
            end if
        end if
    end for
end Help

void CollectGarbage                              ▷ Collects the old root blocks.
    l=FindYoungestOld(Root.Blocks.root)
    t1,t2= RBT.split(l)
    RBTRoot.CAS(t2.root)
end CollectGarbage

Block FindYoungestOld(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 a dequeue cannot find the root block then
        return this.leaf.response(block.order)
    end if
end FallBack
```