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`.

▶ *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* $\texttt{num}_{\texttt{finished}}$: 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.

```
34: void Enqueue(Object e)  ▷ Creates a block with element e and appends it to the tree.
35:     block b= new(leaf block)
36:     b.element= e
37:     b.num_enq=1
38:     b.sum_enq= this.leaf.blocks[this.leaf.head].sum_enq+1
39:     Append(b)
40: end Enqueue

41: Object Dequeue()
42:     block b= new(leaf block)  ▷ 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.   appendStart
43:     b.element= null   appendEnd
44:     b.num_deq=1
45:     b.sum_deq= this.leaf.blocks[this.leaf.head].sum_deq+1  ▷ this is the current running process
46:     Append(b)
47:     <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 index of the block in the root containing it. Since only one invocation of dequeue is running by this process at one time we are allowed to used this.leaf.head safely.
48:     index_response= ComputeDeqRes(i, b)  ▷ index_response is the index of the enqueue which is the response to the dequeue or -1 if the response is null.   deqRest
49:     if index_response!=-1 then
50:         output= null
51:         b_deq=root.blocks[b_i]
52:         b_deq.num_finished.inc()  ▷ shared counter
53:         if b_deq.num_finished==b_deq.num then  ▷ all the operations in the block containing the dequeue are finished.
54:             this.leaf.last_done= b_i
55:         end if
56:     else
57:         output= Get(index_response)
58:         b_r= root.blocks.get(enq, index_response).order  ▷ index of the block in the root contains response enqueue.
59:         b_enq=root.blocks[b_r]
60:         b_enq.num_finished.inc()
61:         b_deq.num_finished.inc()
62:         if b_enq.num_finished==b_enq.num then  ▷ become done
63:             this.leaf.last_done= b_r
64:         else if b_deq.num_finished==b_deq.num then  ▷ b_deq comes after b_enq.
65:             this.leaf.last_done= b_i  ▷ this.leaf.last_done is an increasing value.
66:         end if
67:     end if
68:     return output
69: end Dequeue

70: void Append(block b)
71:     b.group= this.leaf.head  ▷ Only this block is propagated from the leaf by itself.
72:     this.leaf.blocks[this.leaf.head]= b
73:     this.leaf.head+=1  ▷ Lines 71 to 73 are done by one process at time.
74:     Propagate(this.leaf.parent)
75: end Append

76: void Propagate(node n)
77:     if not Refresh(n) then
78:         Refresh(n)  ▷ Lemma Double Refresh
79:     end if
80:     if n is not root then  ▷ To check anode is the root we can check its index if the tree is implemented by an array or check if n.parent is not null.
81:         Propagate(n.parent)
82:     end if
83: end Propagate

84: element Get(int i)  ▷ Returns ith Enqueue.
85:     res= root.blocks.get(enq, i).order
86:     return Get(root, res, i-root.blocks[res-1].sum_enq)
87: end Get

88: 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.
89:     if root.blocks[b-1].size + root.blocks[b].num_enq - i < 0 then
90:         return -1
91:     else return root.blocks[b-1].sum_non-null deq + i
92:     end if
93: end ComputeDeqRes
```

```
boolean REFRESH(node n)
    h= n.head
    c= n.counter
    <new, c_left, c_right>= CREATEBLOCK(n, h)        ▷ c_left, c_right are the values read from n's children's counters.
    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        ▷ how to put 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)        ▷ Superblock's Lemma
            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)        ▷ 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 ≥ 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 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
lastLine    index_last= n.dir.head
prevLine    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
```

```
⇝ 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 of ith dequeue in the bth block of node n among dequeues.
    if n is root then return i, b
    else
        dir= (n.parent.left==n)?  left:  right                                          ▷ check n is a left or a right child
        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[].
        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          ▷ consider dequeues from n's right child
        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. 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 appendStart appendEnd 71-73.
        if l.blocks[last].element==null then     ▷ operation is dequeue
            goto deqRest 48 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
```