algQ

Algorithm Queue

◇ *Local*

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

◇ *Shared*

- *Tree to complete, how?*

◇ *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 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* maxOld

  Index of the youngest old block in the root that this process has seen yet.

▶ *Block*

- *int* $\text{num}_{\text{enq-left}}$, $\text{sum}_{\text{enq-left}}$: #enqueues from subblocks in left child, prefix sum of $\text{num}_{\text{enq-left}}$
- *int* $\text{num}_{\text{deq-left}}$, $\text{sum}_{\text{deq-left}}$: #dequeues from subblocks in left child, prefix sum of $\text{num}_{\text{deq-left}}$
- *int* $\text{num}_{\text{enq-right}}$, $\text{sum}_{\text{enq-right}}$: #enqueues from subblocks in right child, prefix sum of $\text{num}_{\text{enq-right}}$
- *int* $\text{num}_{\text{deq-right}}$, $\text{sum}_{\text{deq-right}}$: #dequeues from subblocks in right child, prefix sum of $\text{num}_{\text{deq-right}}$
- *int* num_{enq}, num_{deq} : # enqueue, dequeue operations in the block
- *int* sum_{enq}, sum_{deq} : sum of # enqueue, dequeue operations in blocks up to this one
- *int* num, sum : total # operations in block, prefix sum of num
- *int* end_{left}, $\text{end}_{\text{right}}$: 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 of the node's counter when propagating this block.
- *int* order : the index of the block in the node containing it

▶ *Leaf Block* extends *Block*

- *Object* element Each block in a leaf also represents an operation. The element shows the operations argument if it is an enqueue, and if it is a dequeue the value is null.

▶ *Root Block* extends *Block*

- *int* size : size of queue after this block's operations finish
- *int* $\text{sum}_{\text{non-null deq}}$: count of non-null dequeus up to this block
- *int* age : number of finished operations in the block

```
1:  void ENQUEUE(Object e)
2:      block b= NEW(block)
3:      b.element= e
4:      b.sum_enq=1
5:      APPEND(b)
6:  end ENQUEUE

7:  Object DEQUEUE()
8:      block b= NEW(block)
9:      b.element= null
10:     b.sum_deq=1
11:     APPEND(b)
12:     <i, b>= INDEX(l_pid, b.order, 1)
13:     res= COMPUTEHEAD(i, b)        ▷ Index of the enqueue whose argument
        deqRest should be returned
14:     return GET(res)
15:     b_i= b                        ▷ block in the root contains the invocation of dequeue
16:     b_r= root.blocks.get(sum_enq==i)        ▷ block in the root contains the
        invocation of dequeue
17:     b_i.age= b_i.age+1
18:     b_r.age= b_r.age+1
19:     if b_i==b_i.num or b_r==b.num then                    ▷ become old
20:         this.leaf.maxOld= Max(b_i*(b_i-b_i.num), b_r*(b_r-b_r.num),
        this.leaf.maxold)
21:     end if
22: end DEQUEUE

23: int COMPUTEHEAD(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.
24:     if root.blocks[b-1].size + root.blocks[b].num_enq - i < 0 then
25:         return -1
26:     else return root.blocks[b-1].sum_non-null deq + i
27:     end if
28: end COMPUTEHEAD

29: void APPEND(block b)
30:     b.group= this.leaf.head
31:     l_pid.blocks[this.leaf.head]= b
32:     this.leaf.head+=1
33:     PROPAGATE(this.leaf.parent)
34: end APPEND
```

```
34: void PROPAGATE(node n)
35:     if not REFRESH(n) then
36:         REFRESH(n)
37:     end if
38:     if n.parent is not null then
39:         PROPAGATE(n.parent)
40:     end if
41: end PROPAGATE

42: boolean REFRESH(node n)
43:     h= n.head
44:     c= n.counter
45:     <new, c_left, c_right>= CREATEBLOCK(n, h)
46:     new.group= c
47:     if new.num==0 then return true
48:     else if n is root then
49:         if root.blocks.append(new) then
50:             goto okcas
51:         end if
52:     else if CAS(n.blocks[h], null, new) then
53: okcas:
54:         for each dir in {left, right} do
55:             CAS(n.dir.super[c_dir], null, h+1)
56:             CAS(n.dir.counter, c_dir, c_dir+1)
57:         end for
58:         CAS(n.head, h, h+1)
59:         return true
60:     else
61:         CAS(n.head, h, h+1)
62:         return false
63:     end if
64: end REFRESH

65: element GET(int i)                          ▷ Returns ith Enqueue.
66:     if i is null then
67:         return null
68:     end if
69:     res= root.blocks.get(sum_enq==i).order
70:     return GET(root, res, i-root.blocks[res-1].sum_enq)
71: end GET

    ⇝ Precondition: n.blocks[start..end] contains a block with field f ≥ i
72: 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.
73: end BSEARCH

74: <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.
75:     block b= NEW(block)
76:     b.order= i
77:     for each dir in {left, right} do
78: lastLine:
79:         lastIndex= n.dir.head
80: prevLine:
81:         prevIndex= n.blocks[i-1].end_dir
82:         lastBlock= n.dir.blocks[lastIndex]
83:         prevBlock= n.dir.blocks[prevIndex]
84:         c_dir= n.dir.counter
85:         b.end_dir= lastIndex
86:         b.num_enq-dir= lastBlock.sum_enq - prevBlock.sum_enq
87:         b.num_deq-dir= lastBlock.sum_deq - prevBlock.sum_deq
88:         b.sum_enq-dir= n.blocks[i-1].sum_enq-dir + b.num_enq-dir
89:         b.sum_deq-dir= n.blocks[i-1].sum_deq-dir + b.num_deq-dir
90:     end for
91:     b.num_enq= b.num_enq-left + b.num_enq-right
92:     b.num_deq= b.num_deq-left + b.num_deq-right
93:     b.num= b.num_enq + b.num_deq
94:     b.sum= n.blocks[i-1].sum + b.num
95:     if n.parent is null then
96:         Cast(b, RootBlock)                  ▷ cast block to a root block
97:         b.size= max(root.blocks[i-1].size + b.num_enq - b.num_deq, 0)
98:         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)
99:     end if
100:    return b, c_left, c_right
101: end CREATEBLOCK
```

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

```
84: element GET(node n, int b, int i)                          ▷ Returns the ith Enqueue in bth block of node n
85:     if n is leaf then return n.blocks[b].element
86:     else
87:         if i ≤ n.blocks[b].num_enq-left then                          ▷ i exists in the left child of n
88:             subBlock= BSEARCH(n.left, sum_enq, i, n.blocks[b-1].end_left+1, n.blocks[b].end_left)
89:             return GET(n.left, subBlock, i-n.left.blocks[subBlock-1].sum_enq)
90:         else
91:             i= i-n.blocks[b].num_enq-left
92:             subBlock=BSEARCH(n.right, sum_enq, i, n.blocks[b-1].end_right+1, n.blocks[b].end_right)
93:             return GET(n.right, subBlock, i-n.right.blocks[subBlock-1].sum_enq)
94:         end if
95:     end if
96: 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.

```
97: <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
98:     if n is root then return root.blocks.get(order==b-1).sum_deq+i, b
99:     else
100:        dir= (n.parent.left==n)?  left:  right
101:        superBlock= BSEARCH(n.parent, n.sum_deq-dir, i, super[n.blocks[b].group]-p, super[n.blocks[b].group]+p)
102:        if dir is left then
103:            i+= n.parent.blocks[superBlock-1].sum_deq-right
104:        else
105:            i+= n.parent.blocks[superBlock-1].sum_deq + n.blocks[superBlock].sum_deq-left
106:        end if
107:        return INDEX(n.parent, superBlock, i)
108:    end if
109: 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.

```
1:  void RBTAPPEND(block b)                          ▷ adds block b to the root.blocks
2:      step= root.head
3:      if step%p^2==0 then
4:          Help()
5:          CollectGarbage()
6:      end if
7:      b.age= 0
8:      return root.blocks.append(b, b.order)
9:  end RBTAPPEND

10: void HELP                                        ▷ Helps pending operations
11:     for leaf l in leaves do                      ▷ how to iterate over them?
12:         last= l.head-1
13:         if l.blocks[last] is not null then
14:             if l.blocks[last].element==null then ▷ operation is dequeue
15:                 goto deqRest13 with these values <>   ▷ run Dequeue() for l.ops[last] after Propagate(). TODO
16:                 l.responses[last]= response
17:             end if
18:         end if
19:     end for
20: end HELP

21: void COLLECTGRABAGE                              ▷ Collects the old root blocks.
22:     l=FindYoungestOld(Root.Blocks.root)
23:     t1,t2= RBT.split(l)
24:     RBTRoot.CAS(t2.root)
25: end COLLECTGRABAGE

26: Block FINDYOUNGESTOLD(b)
27:     for leaf l in leaves do
28:         max= Max(l.maxOld, max)
29:     end for
30:     return max                                   ▷ This snapshot suffies.
31: end FINDYOUNGESTOLD

32: response FALLBACK(op i)
33:     if a dequeue cannot find the root block then
34:         return this.leaf.response(block.order)
35:     end if
36: end FALLBACK
```