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 coresponding children. blocks[i] returns the ith block stored.
- *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

```
void Enqueue(Object e)
    block b= new(block)
    b.element= e
    b.$sum_{enq}$=1
    Append(b)
end Enqueue

Object Dequeue()
    block b= new(block)
    b.element= null
    b.$sum_{deq}$=1
    Append(b)
    <i, b>= Index($l_{pid}$, b.order, 1)
    res= ComputeHead(i, b)        ▷ Index of the enqueue whose argument should be returned
deqRest
    return Get(res)
    $b_i$= b        ▷ block in the root contains the invocation of dequeue
    $b_r$= root.blocks.get($sum_{enq}$==i)        ▷ block in the root contains the invocation of dequeue
    $b_i$.age= $b_i$.age+1
    $b_r$.age= $b_r$.age+1
    if $b_i$ or $b_r$ become old then update maxOld
    end if
end Dequeue

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.
    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 ComputeHead

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
```

```
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 53
51:         end if
52:     else if CAS(n.blocks[h], null, new) then
okcas 53:       for each dir in {left, right} do
54:             CAS(n.dir.super[c$_{dir}$], null, h+1)
55:             CAS(n.dir.counter, c$_{dir}$, c$_{dir}$+1)
56:         end for
57:         CAS(n.head, h, h+1)
58:         return true
59:     else
60:         CAS(n.head, h, h+1)
61:         return  false
62:     end if
63: end Refresh

64: element Get(int i)                                  ▷ Returns $i$th Enqueue.
65:     if i  is null then
66:         return null
67:     end if
68:     res= root.blocks.get(sum$_{enq}$==i).order
69:     return Get(root, res, i-root.blocks[res-1].sum$_{enq}$)
70: end Get

    ⇝ Precondition: n.blocks[start..end] contains a block with field f ≥ i
71: 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.
72: end BSearch
```

```
73: <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.
74:     block b= new(block)
75:     b.order= i
76:     for each dir in {left, right} do
lastLine 77:    lastIndex= n.dir.head
prevLine 78:    prevIndex= n.blocks[i-1].end$_{dir}$
79:         lastBlock= n.dir.blocks[lastIndex]
80:         prevBlock= n.dir.blocks[prevIndex]
81:         c$_{dir}$= n.dir.counter
82:         b.end$_{dir}$= lastIndex
83:         b.num$_{enq-dir}$= lastBlock.sum$_{enq}$ - prevBlock.sum$_{enq}$
84:         b.num$_{deq-dir}$= lastBlock.sum$_{deq}$ - prevBlock.sum$_{deq}$
85:         b.sum$_{enq-dir}$= n.blocks[i-1].sum$_{enq-dir}$ + b.num$_{enq-dir}$
86:         b.sum$_{deq-dir}$= n.blocks[i-1].sum$_{deq-dir}$ + b.num$_{deq-dir}$
87:     end for
88:     b.num$_{enq}$= b.num$_{enq-left}$ + b.num$_{enq-right}$
89:     b.num$_{deq}$= b.num$_{deq-left}$ + b.num$_{deq-right}$
90:     b.num= b.num$_{enq}$ + b.num$_{deq}$
91:     b.sum= n.blocks[i-1].sum + b.num
92:     if n.parent is null then
93:         b.size= max(root.blocks[i-1].size + b.num$_{enq}$ - b.num$_{deq}$, 0)
94:         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)
95:     end if
96:     return b, c$_{left}$, c$_{right}$
97: 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.ops[b]
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 deqRest 13 with      ▷ run Dequeue() for l.ops[last] after
    Propagate()
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:     return      read all maxOld values among leaves and decide the
    largest one                   ▷ There is no need to do a sasDK;Lnapgshot.
28: end FINDYOUNGESTOLD

29: response FALLBACK(op i)
30:     if operation i in leaf l cannot find its desired RootBlock then
31:         return l.response[i]
32:     end if
33: end FALLBACK
```