

- 1 Feilds
- 2 Queue
- 3 Search+Append
- 4 Propagate
- 5 Index+Get
- 6 MaxByProcess
- 7 Help
- 8 FreeMemory
- 9 PBRT

Algorithm Tree Fields Description

◇ *Shared*

- A binary tree of Nodes with one leaf for each process. root is the root node.
- *MaxbyProcess lastDequeuedFrom* Index of the most recent block in the root that has been dequeued from.

◇ *Local*

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

► *Node*

- **Node left, right, parent* : Initialized when creating the tree.
- *PBRT blocks* : Initially `blocks[0]` contains an empty block with all fields equal to 0.
- *int head= 1*: #blocks in `blocks`. `blocks[0]` is a block with all integer fields equal to zero.

► *Block*

- *int super* : approximate index of the superblock, read from `parent.head` when appending the block to the node

► *InternalBlock* extends *Block*

- *int end_{left}, end_{right}* : indices of the last subblock of the block in the left and right child
- *int sum_{enq-left}*: #enqueues in `left.blocks[1..endleft]`
- *int sum_{deq-left}*: #dequeues in `left.blocks[1..endleft]`
- *int sum_{enq-right}*: #enqueues in `right.blocks[1..endright]`
- *int sum_{deq-right}*: #dequeues in `right.blocks[1..endright]`

► *LeafBlock* extends *Block*

- *Object element* : Each block in a leaf represents a single operation. If the operation is `enqueue(x)` then `element=x`, otherwise `element=null`.
- *int sum_{enq}, sum_{deq}* : # enqueue, dequeue operations in this block and its previous blocks in the leaf
- *object response*

► *RootBlock* extends *InternalBlock*

- *int size* : size of the queue after performing all operations in this block and its previous blocks in the root
-

Algorithm Queue

```
1: void Enqueue(Object e)                                ▷ Creates a block with element e and adds it to the tree.
2:   block newBlock= new(LeafBlock)
3:   newBlock.element= e
4:   newBlock.sumenq= leaf.blocks[leaf.head].sumenq+1
5:   newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq
6:   leaf.Append(newBlock)
7: end Enqueue

▷ Creates a block with null value element, appends it to the tree and returns its response.

8: Object Dequeue()
9:   block newBlock= new(LeafBlock)
10:  newBlock.element= null
11:  newBlock.sumenq= leaf.blocks[leaf.head].sumenq
12:  newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq+1
13:  leaf.Append(newBlock)
14:  <b, i>= IndexDequeue(leaf.head, 1)
15:  output= FindResponse(b, i)
16:  return output
17: end Dequeue

▷ Returns the response to  $D_i(root, b)$ , the  $i$ th Dequeue in  $root.blocks[b]$ .

18: element FindResponse(int b, int i)
19:   if root.blocks[b-1].size + root.blocks[b].numenq - i < 0 then           ▷ Check if the queue is empty.
20:     lastDequeudFrom.update(b)
21:     return null
22:   else                               ▷ The response is  $E_e(root)$ , the  $e$ th Enqueue in the root.
23:     e= i + (root.blocks[b-1].sumenq-root.blocks[b-1].size)
24:     <x, y>= root.DoublingSearch(e, b)
25:     lastDequeudFrom.update(x)
26:     return root.GetEnqueue(x,y)
27:   end if
28: end FindResponse
```

Algorithm *Node*

↪ Precondition: `blocks[start..end]` contains a block with `sumenq` greater than or equal to `x`

▷ Update needed: search on RBT does not need start and end, we can search over whole the red-black tree..

```
26: int BinarySearch(int x)
27:   return min{j: blocks[j].sumenq ≥ x}
28: end BinarySearch
```

Algorithm *Root*

↪ Precondition: `root.blocks[end].sumenq ≥ e`

▷ Returns `<b,i>` such that $E_e(\text{root}) = E_i(\text{root}, b)$, i.e., the `e`th Enqueue in the `root` is the `i`th Enqueue within ▷ the `b`th block in the `root`.

```
37: <int, int> DoublingSearch(int e, int end) ▷ I think with garbage collection, doubling search is not needed any more
    and a binary search on root.blocks would be good enough.
38: end DoublingSearch
```

Algorithm *Leaf*

```
46: void Append(block B) ▷ Only called by the owner of the leaf.
47:   blocks.TryAppend(B, head)
48:   head= head+1
49:   parent.Propagate()
50: end Append
```

Algorithm *Node*

▷ *n*.Propagate propagates operations in **this**.children up to **this** when it terminates.

```
51: void Propagate()
52:   if not Refresh() then
53:     Refresh()
54:   end if
55:   if this is not root then
56:     parent.Propagate()
57:   end if
58: end Propagate
```

▷ Creates a block containing new operations of **this**.children, and then tries to append it to **this**.

```
59: boolean Refresh()
60:   h= head
61:   for each dir in {left, right} do
62:     hdir= dir.head
63:     if dir.blocks[hdir] != null then
64:       dir.Advance(hdir)
65:     end if
66:   end for
67:   new= CreateBlock(h)
68:   if new.num==0 then return true
69:   end if
70:   result= blocks.TryAppend(new, h)
71:   this.Advance(h)
72:   return result
73: end Refresh
```

Algorithm *Node*

```
74: void Advance(int h)                                ▷ Sets blocks[h].super and increments head from h to h+1.
75:     hp = parent.head
76:     blocks[h].super.CAS(null, hp)
77:     head.CAS(h, h+1)
78: end Advance

79: Block CreateBlock(int i)                            ▷ Creates and returns the block to be installed in blocks[i].
80:     block new = new(InternalBlock)
81:     for each dir in {left, right} do
82:         indexprev = blocks[i-1].enddir
83:         new.enddir = dir.head-1                      ▷ new contains dir.blocks[blocks[i-1].enddir..dir.head-1].
84:         blockprev = dir.blocks[indexprev]
85:         blocklast = dir.blocks[new.enddir]
86:         new.sumenq-dir = blocks[i-1].sumenq-dir + blocklast.sumenq - blockprev.sumenq
87:         new.sumdeq-dir = blocks[i-1].sumdeq-dir + blocklast.sumdeq - blockprev.sumdeq
88:     end for
89:     if this is root then
90:         new.type = InternalBlock-->RootBlock
91:         new.size = max(root.blocks[i-1].size + new.numenq - new.numdeq, 0)
92:     end if
93:     return new
94: end CreateBlock

95: int GetLastDequeuedFrom                            ▷ Returns the index that is safe to remove the blocks before that in the node.
96:     x = lastDequeuedFrom.Get()-1
97:     n = root
98:     while n!=this do
99:         dir = left (if this is in left subtree of n): otherwise dir=right
100:         x = n.blocks[x].enddir
101:     end while
102: end GetLastDequeuedFrom
```

Algorithm Node

~~ Precondition: $\text{blocks}[b].\text{num}_{\text{enq}} \geq i \geq 1$

```
95: element GetEnqueue(int b, int i)                                ▷ Returns the element of  $E_i(\text{this}, b)$ .
96:   if this is leaf then
97:     return blocks[b].element
98:   else if i <= blocks[b].numenq-left then                        ▷  $E_i(\text{this}, b)$  is in the left child of this node.
99:     subblockIndex= left.BinarySearch(i+blocks[b-1].sumenq-left, blocks[b-1].endleft+1,
                                     blocks[b].endleft)           ▷ start and end values are not needed anymore?
100:    return left.GetEnqueue(subblockIndex, i)
101:   else
102:     i= i-blocks[b].numenq-left
103:     subblockIndex= right.BinarySearch(i+blocks[b-1].sumenq-right, blocks[b-1].endright+1,
                                     blocks[b].endright)           ▷ start and end values are not needed anymore?
104:    return right.GetEnqueue(subblockIndex, i)
105:   end if
106: end GetEnqueue
```

~~ Precondition: bth block of the node has propagated up to the root and $\text{blocks}[b].\text{num}_{\text{deq}} \geq i$.

```
107: <int, int> IndexDequeue(int b, int i)    ▷ Update needed: return null when superblock in the root was not found.
108:   if this is root then
109:     return <b, i>
110:   else
111:     dir= (parent.left==n ? left: right)
112:     superblockIndex= parent.blocks[blocks[b].super].sumdeq-dir > blocks[b].sumdeq ?
                                     blocks[b].super: blocks[b].super+1           ▷ Preconditions might be not met.
113:     if dir is left then
114:       i+= blocks[b-1].sumdeq-parent.blocks[superblockIndex-1].sumdeq-left
115:     else
116:       i+= blocks[b-1].sumdeq-parent.blocks[superblockIndex-1].sumdeq-right
117:       i+= parent.blocks[superblockIndex].numdeq-left
118:     end if
119:     return this.parent.IndexDequeue(superblockIndex, i)
120:   end if
121: end IndexDequeue
```

Algorithm *MaxByProcess*

```
122: int[p] lastDequeuedbyProcess

123: int Get
124:   return max(lastDequeuedbyProcess)
125: end Get

126: Update(int b)
127:   if lastDequeuedbyProcess[pid]<b then
128:     lastDequeuedbyProcess[pid]=b
129:   end if
130: end Update
```

Algorithm *Tree*

```
131: int Help
132:   for each process P
133:     h=P.leaf.head
134:     if P.leaf.blocks[h].num_deq==1 and P.leaf.IndexDequeue(h,1)!=null then
135:       <b, i>= IndexDequeue(h, 1)
136:       output= FindResponse(b, i)
137:       P.leaf.blocks[h].response= output
138:     end if
139:   end for
140: end Help
```

Algorithm *Node*

```
141: FreeMemory(int b)
142:   if not leaf then
143:     left.FreeMemory(blocks[i].end_left-1)
144:     right.FreeMemory(blocks[i].end_right-1)
145:   end if
146:   blocks= blocks.splitGreater(i)
147: end FreeMemory
```

▷ I think CAS is not needed.

Algorithm *PBRT*

```
PBRT prbt
nodes store <key, sumenq-> block
[i] -> GetByBlock(i)

141: GetByBlock(int i)
142:   return rbt.get(i)
143:   if not found then
144:     return written response
145:   end if
146: end GetByBlock

147: TryAppend(block B, int i)
    Tries to append B with key i to the red-black tree.
148:   if i%p2=0 then
149:     Help()
150:     root.FreeMemory()(lastDequeuedFrom.Get()-1)
151:     garbageCollectRound=floor(root.head/p2)
152:   end if
153: end FreeMemory
```

10 Description

In our original algorithm an **Enqueue** or a **Dequeue** remains in the **blocks** array in the tree nodes even after they terminate. This makes the space used by the algorithm factor of the number of operations of invoked on the queue. In this section here we want to free the memory allocated by the operations that are no longer needed and make the space used polynomial of $p + q$.

One way of handling garbage collection is to reallocate the space taken by each operation right after it is not needed anymore, but it might cost too much to do that. So we garbage collect by batching. If we do the garbage collection every p^2 block appended to the root, its cost is amortized over p^2 blocks which is $O(p^3)$ and $\Omega(p^2)$.

In our design a process attempts to do **Garbage Collect** every p^2 block appended to the root. **Garbage Collect** corresponded to the kp^2 root block is called the k th round. To know if it is k th round or not we can use an if condition on the value of **root.head**. But the problem is that a process might miss a round not because it is on idle but because it reads **root.head** not to be kp^2 . We handle that by checking with a window.

Lemma 1. *The number of the blocks in the **blocks** is $O(p^2 + q)$.*

Lemma 2. *When $\frac{\text{root.head}}{p^2}$ increments from r to $r + 1$, at least one operation has executed Lines 48,49.*

An **Enqueue** operation cannot be removed after it is terminated since later it might be dequeued, however it can be removed from the tree after it has been computed to be the response of a **Dequeue**. A **Dequeue** operation can be removed from the tree safely after it has computed its response **Enqueue**. Note that it is not needed for the **Dequeue** to be terminated but only it is sufficient if it has computed its response. We say a block in the **root** is *finished* if all of its operations can be removed.

Lemma 3. *There are at most $p^2 + q$ not finished blocks in the **root.blocks** at a time.*

The good property in a fifo queue is that if i th **Enqueue** gets dequeued then it means 1 to $i - 1$ th enqueue operation have also dequeued. This gives us the idea that if a block is finished all the blocks before it are also finished. However if an operation in a block goes to sleep for a long time then the block remains not finished. Since there are at most p idle operations we can help them and then remove all the finished blocks safely.

Lemma 4. *If all current operations are helped, then there is a block in the root that all of its previous blocks are finished.*

The idea above leads us to a poly-log data structure that supports throwing away all the values smaller than an index. Red-black trees do this for us. We can create a shared red-black tree just creating a new path for the operation and then using CAS to change the root of the tree. See [this] for more.

Observation 5. *PBRT supports....*

Lemma 6. *If we replace the arrays we used to implement `blocks` with red-black trees the amortized complexity of the algorithm would be $PolyLog(p, q)$. And also the algorithm is correct.*

We can help a `Dequeue` by computing its response and writing it down. If the process in future failed to execute, it can read the helped value written down.

Lemma 7. *The `response` written is correct.*

But how can we know which blocks in each node are finished or not? We can keep track of the last finished block.

Lemma 8. *If all current operations are helped, then the blocks before the newest block that some `Enqueue` has been dequeued from is safe to remove. If the most current `Dequeue` returned `null` then all the blocks before the block containing the `Dequeue` can be removed.*

There is a shared array among processes which they write the last block dequeued from in it.

Lemma 9. *`lastDequeuedFrom`- index of the last finished block in the root is $O(p)$.*

Lemma 10. *After `FreeMemory`, the space taken by each node is $O(p^2 + q)$*