
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.BinarySearch(e)
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 *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:   if h%p2=0 then
62:     Help()
63:     root.FreeMemory() (lastDequeuedFrom.Get()-1)
64:   end if
65:   for each dir in {left, right} do
66:     hdir= dir.head
67:     if dir.blocks[hdir] != null then
68:       dir.Advance(hdir)
69:     end if
70:   end for
71:   new= CreateBlock(h)
72:   if new.num==0 then return true
73:   end if
74:   result= blocks.TryAppend(new, h)
75:   this.Advance(h)
76:   return result
77: 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].sumeqleft, 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 $\text{blocks}[b].\text{num_deg} \geq i$.

```
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)
```

[illegible]

```
113:         if dir is left then
```

```
114:         i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-left
```

```
115:         else
```

```
116:         i+= blocks[b-1].sum_deg-parent.blocks[superblockIndex-1].sum_deg-right
```

```
117:         i+= parent.blocks[superblockIndex].num_deg-left
```

```
118:         end if
```

```
119:     return this.parent.IndexDeque(superblockIndex, i)
```

```
120:    end if
```

```
121: end IndexDeque
```

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
```

1 Description

In our 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. Here in this section, we want to add a mechanism to free the memory allocated by the operations that are no longer needed and make the overall space used polynomial of $p + q$.

It is a common way to do garbage collection in batches. If we garbage collect the unnecessary blocks in the nodes every p^2 block appended to the root, the garbage collection cost is amortized over p^2 blocks which is $O(p^3)$ and $\Omega(p^2)$ operations.

In our design a process attempts to collect the garbage, when it is going to append a block in the kp^2 position in **root.blocks** (see Line ??). Every p^2 block appended to the root, one **GarbageCollect** terminates because **root.head** cannot advance until a **GarbageCollect** garbage collect is done (see Lines).

Lemma 1. *Size of **root.blocks** after **GarbageCollect** is $O(p^2 + q)$.*

Lemma 2. *Total number of the blocks in the tree is $O(p^3 + pq)$.*

Enqueue operation e can be removed from the tree after termination of **Dequeue** d where $Resp(d) = e$ or $e.element$ has been computed to be $RESP(d)$. It is safe to remove **Dequeue** d from the tree after the d is terminated or $Resp(d).element$ has been computed. We can remove a block after the told conditions are satisfied for all of its operations. A **Dequeue** may go to sleep for a long time and prevent the **block** to be removed. In that situation other processes can help the **Dequeue** by computing its response and writing it down somewhere. After writing down the response of the **Dequeue** it is safe to remove. Because if the **Dequeue** failed to compute its response it can read the response written (computing the **Dequeue** index in the **root** or getting the response **Enqueue**).

Definition 3. A block is *finished* if all of its **Enqueues** have been computed to be the response of some **Dequeue** and all of its **Dequeues** responses have been computed.

Corollary 4. *If a block is finished, then all of its subblocks are also finished.*

Lemma 5. *It is safe to remove a finished block from the tree.*

If the i th **Enqueue** gets dequeued in a FIFO queue, it means the first $i - 1$ **Enqueue** operations have been already dequeued. This gives us the idea that if a block is finished then all the blocks before it are also finished. If an operation in a block goes to sleep for a long time then other processes help the operation so

the block get finished. Note that when an **Enqueue** operation in a block is not finished the block cannot be finished until some **Dequeue** dequeues that **Enqueue**. Since there are at most p idle operations, we can help them before garbage collection and then remove all the finished blocks safely.

Lemma 6. *If all current operations are helped, then there is a block in the root that all of its previous blocks are finished. That block is the most recent block that has been dequeued from.*

The idea above leads us to a poly-log data structure that supports throwing away all the blocks with keys smaller than an index. Red-black trees do this for us. **Get(i)**, **Append()** and **Split(i)** are logarithmic in block trees. 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 7. *PBRT supports poly-log operations*

Lemma 8. *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 9. *The response written is correct.*

But how can we know which blocks in each node are finished or not?

Lemma 10. *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 11. *$GetLastDequeuedFrom(n)$ — index of the last finished block in the node n is $O(p)$.*

Lemma 12. *After **FreeMemory**, the space taken by each node of the tree is $O(p^2 + q)$. The total space in the tree is $PolyLog(p + q)$*

Lemma 13. *The amortized step per process for **GarbageCollect** is $O()$.*

Lemma 14. *Algorithm is wait-free and linearizable.*