**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 deqeueued 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* $\mathrm{end}_{\mathrm{left}}$, $\mathrm{end}_{\mathrm{right}}$ : indices of the last subblock of the block in the left and right child
- *int* $\mathrm{sum}_{\mathrm{enq\text{-}left}}$: #enqueues in left.blocks[1..$\mathrm{end}_{\mathrm{left}}$]
- *int* $\mathrm{sum}_{\mathrm{deq\text{-}left}}$: #dequeues in left.blocks[1..$\mathrm{end}_{\mathrm{left}}$]
- *int* $\mathrm{sum}_{\mathrm{enq\text{-}right}}$: #enqueues in right.blocks[1..$\mathrm{end}_{\mathrm{right}}$]
- *int* $\mathrm{sum}_{\mathrm{deq\text{-}right}}$: #dequeues in right.blocks[1..$\mathrm{end}_{\mathrm{right}}$]

▶ *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* $\mathrm{sum}_{\mathrm{enq}}$, $\mathrm{sum}_{\mathrm{deq}}$ : # 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*

```
void Enqueue(Object e)                      ▷ Creates a block with element e and adds it to the tree.
    block newBlock= new(LeafBlock)
    newBlock.element= e
    newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq+1
    newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq
    leaf.Append(newBlock)
end Enqueue

    ▷ Creates a block with null value element, appends it to the tree and returns its response.
Object Dequeue()
    block newBlock= new(LeafBlock)
    newBlock.element= null
    newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq
    newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq+1
    leaf.Append(newBlock)
    <b, i>= IndexDequeue(leaf.head, 1)
    output= FindResponse(b, i)
    return output
end Dequeue

    ▷ Returns the response to D_i(root, b), the ith Dequeue in root.blocks[b].
element FindResponse(int b, int i)
    if root.blocks[b-1].size  + root.blocks[b].num_enq - i < 0 then      ▷ Check if the queue is empty.
         lastDequeudFrom.update(b)
        return null
    else                                    ▷ The response is E_e(root), the eth Enqueue in the root.
        e= i + (root.blocks[b-1].sum_enq-root.blocks[b-1].size)
        <x, y>= root.BinarySearch(e)
        lastDequeudFrom.update(x)
        return root.GetEnqueue(x,y)
    end if
end FindResponse
```

**Algorithm** *Node*

```
⇝ Precondition: blocks[start..end] contains a block with sum_enq 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..
int BinarySearch(int x)
    return min{j: blocks[j].sum_enq≥x}
end BinarySearch
```

**Algorithm** *Leaf*

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

**Algorithm** *Node*

```
▷ n.Propagate propagates operations in this.children up to this when it terminates.
void Propagate()
    if not Refresh() then
        Refresh()
    end if
    if this is not root then
        parent.Propagate()
    end if
end Propagate

▷ Creates a block containing new operations of this.children, and then tries to append it to this.
boolean Refresh()
    h= head
    if h%p^2=0 then
        Help()
        root.FreeMemory()(lastDequeuedFrom.Get()-1)
    end if
        for each dir in {left, right} do
            h_dir= dir.head
            if dir.blocks[h_dir]!=null then
                dir.Advance(h_dir)
            end if
        end for
        new= CreateBlock(h)
        if new.num==0 then return true
        end if
        result= blocks.TryAppend(new, h)
        this.Advance(h)
        return  result
    end Refresh
```

**Algorithm** *Node*

```
void Advance(int h)                    ▷ Sets blocks[h].super and increments head from h to h+1.
    h_p= parent.head
    blocks[h].super.CAS(null, h_p)
    head.CAS(h, h+1)
end Advance

Block CreateBlock(int i)               ▷ Creates and returns the block to be installed in blocks[i].
    block new= new(InternalBlock)
    for each dir in {left, right} do
        index_prev= blocks[i-1].end_dir
        new.end_dir= dir.head-1        ▷ new contains dir.blocks[blocks[i-1].end_dir..dir.head-1].
        block_prev= dir.blocks[index_prev]
        block_last= dir.blocks[new.end_dir]
        new.sum_enq-dir= blocks[i-1].sum_enq-dir + block_last.sum_enq - block_prev.sum_enq
        new.sum_deq-dir= blocks[i-1].sum_deq-dir + block_last.sum_deq - block_prev.sum_deq
    end for
    if this is root then
        new.type= InternalBlock-->RootBlock
        new.size= max(root.blocks[i-1].size  + new.num_enq- new.num_deq, 0)
    end if
    return new
end CreateBlock

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

**Algorithm** *Node*

```
⇝ Precondition: blocks[b].num_enq ≥ i ≥ 1
element GetEnqueue(int b, int i)                ▷ Returns the element of E_i(this, b).
    if this is leaf then
        return blocks[b].element
    else if i <= blocks[b].num_enq-left then    ▷ E_i(this, b) is in the left child of this node.
        subblockIndex= left.BinarySearch(i+blocks[b-1].sum_enq-left, blocks[b-1].end_left+1,
                        blocks[b].end_left)      ▷ start and end values are not needed anymore?
        return left.GetEnqueue(subblockIndex, i)
    else
        i= i-blocks[b].num_enq-left
        subblockIndex= right.BinarySearch(i+blocks[b-1].sum_enq-right, blocks[b-1].end_right+1,
                        blocks[b].end_right)     ▷ start and end values are not needed anymore?
        return right.GetEnqueue(subblockIndex, i)
    end if
end GetEnqueue

⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num_deq ≥ i.
<int, int> IndexDequeue(int b, int i)   ▷ Update needed: return null when superblock in the root was not found.
    if this is root then
        return <b, i>
    else
        dir= (parent.left==n ? left: right)
        superblockIndex= parent.blocks[blocks[b].super].sum_deq-dir > blocks[b].sum_deq ?
                        blocks[b].super: blocks[b].super+1      ▷ Preconditions might be not met.
        if dir is left then
            i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-left
        else
            i+= blocks[b-1].sum_deq-parent.blocks[superblockIndex-1].sum_deq-right
            i+= parent.blocks[superblockIndex].num_deq-left
        end if
        return this.parent.IndexDequeue(superblockIndex, i)
    end if
end IndexDequeue
```

**Algorithm** *MaxByProcess*

```
int[p] lastDequeuedbyProcess

int Get
    return max(lastDequeuedbyProcess)
end Get

Update(int b)
    if lastDequeuedbyProcess[pid]<b then
        lastDequeuedbyProcess[pid]=b
    end if
end Update
```

**Algorithm** *Tree*

```
int Help
    for each process P
    h=P.leaf.head
    if P.leaf.blocks[h].num_deq==1 and P.leaf.IndexDequeue(h,1)!=null then
        <b, i>= IndexDequeue(h, 1)
        output= FindResponse(b, i)
        P.leaf.blocks[h].response= output
    end if
    end for
end Help
```

**Algorithm** *Node*

```
FreeMemory(int b)
    if not leaf then
        left.FreeMemory(blocks[i].end_left-1)
        right.FreeMemory(blocks[i].end_right-1)
    end if
    blocks= blocks.splitGreater(i)                ▷ I think CAS is not needed.
end FreeMemory
```

**Algorithm** *PBRT*

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

GetByBlock(int i)
    return rbt.get(i)
    if not found then
        return written response
    end if
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 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 $d$o 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 ofits 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 th 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.*