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 deqeueued from.

◇ *Local*

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

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

▶ *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* $\texttt{sum}_{\texttt{enq}}$, $\texttt{sum}_{\texttt{deq}}$: # 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.DoublingSearch(e, b)
        lastDequeudFrom.update(x)
        return root.GetEnqueue(x,y)
    end if
end FindResponse
```

Algorithm *Node*

⇝ Precondition: `blocks[start..end]` contains a block with $\texttt{sum}_{\texttt{enq}}$ greater than or equal to `x`

▷ Update needed: search on RBT.

```
int BinarySearch(int x, int start, int end)
    while start<end do
        int mid= floor((start+end)/2)
        if blocks[mid].sum_enq<x then
            start= mid+1
        else
            end= mid
        end if
    end while
    return start
end BinarySearch
```

Algorithm *Root*

⇝ Precondition: $\texttt{root.blocks[end].sum}_{\texttt{enq}} \geq \texttt{e}$

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

```
<int, int> DoublingSearch(int e, int end)
    start= end-1
    while root.blocks[start].sum_enq>=e do
        start= max(start-(end-start), 0)
    end while
    b= root.BinarySearch(e, start, end)
    i= e- root.blocks[b-1].sum_enq
    return <b,i>
end DoublingSearch
```

Algorithm *Leaf*

```
void Append(block B)                                   ▷ Only called by the owner of the leaf.
    if root.head%p^2<p and garbageCollectRound<floor(root.head/p^2) then
        Help()
        root.FreeMemory()(lastDequeuedFrom.Get()-1)
        garbageCollectRound=floor(root.head/p^2)
    end if
    blocks[head]= B
    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
    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[h].CAS(null, new)
    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
```

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)
        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)
        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
        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)
        right.FreeMemory(blocks[i].end_right)
    end if
    while !blocks.CAS(blocks.splitGreater(i))
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 FreeMemory
```