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 `Node`s 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* $\texttt{end}_{\texttt{left}}$, $\texttt{end}_{\texttt{right}}$: 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 does not need start and end, we can search over whole the red-black tree..

int `BinarySearch(`*int* `x)`

  return $\texttt{min}\{\texttt{j: blocks[j].sum}_{\texttt{enq}} \geq \texttt{x}\}$

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)` ▷ I think with garbage collection, doubling search is not needed any more and a binary search on `root.blocks` would be good enough.

end `DoublingSearch`

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
    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


TryAppend(block B, int i)
    Tries to append B with key i to the red-black tree.
    if i%p^2=0 then
        Help()
        root.FreeMemory()(lastDequeuedFrom.Get()-1)
        garbageCollectRound=floor(root.head/p^2)
    end if
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 garbage collect the unnecessary blocks in a node every p^2 block appended to the node, the garbage collection cost is amortized over p^2 blocks which is $O(p^3)$ and $\Omega(p^2)$.

In our design a process garbage collects a node when it attempts to append a block in the kp^2 position in the. `Garbage Collect` corresponded to the kp^2th block in node n is called the kth round of garbage collect on n. Every p^2 block appended to a node at least one garbage collect happens because the node's `head` cannot advance until the garbage collect is done (see Lines).

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

An `Enqueue` operation can be removed from the tree after its `element` has been computed to be the response of some `Dequeue`. It is safe to remove a `Dequeue` operation from the tree after the `Dequeue` is terminated, but a `Dequeue` might sleep for a long time and prevents to be garbage collected. In that situation other processes can help the `Dequeue` to compute its response and wtite down its response somewhere. If we remove a `Dequeue` from the tree after it is helped the `Dequeue` can read the helped response writtern when it encountered a problem (computing the `Dequeue` index in the `root` or getting the response `Enqueue`). A `Dequeue` operation can be removed from the tree after its response has been computed. We say a block is *finished* if all of its operations can be removed.

Lemma 2. *There are at most* $O(p^2 + q)$ *unfinished blocks in a node's* `blocks` *at a time.*

If the ith `Enqueue` gets dequeued in a FIFO queue, it means 1 to $i - 1$th `Enqueue` operations are also 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 3. *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 4. *PBRT supports poly-log operations ....*

Lemma 5. *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 6. *The* `response` *written is correct.*

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

Lemma 7. *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 8. `GetLastDequeuedFrom(`$n -$ *index of the last finished block in the node n is $O(p)$.*

Lemma 9. *After* `FreeMemory`*, the space taken by each node of the tree is $O(p^2 + q)$. The total space in the tree is $PolyLog(p + q)$*