# Wait-free Queues with Polygarithmic Step Complexity

July 14, 2022

## 1 Previous Work

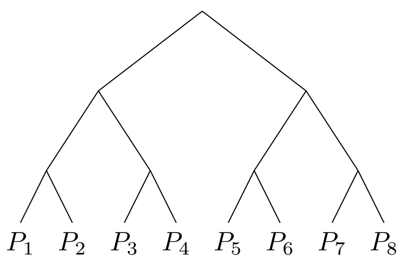
| | Type | Progress Property | Complexity |
|---|---|---|---|
| MichaelS96 DBLP:conf/podc/MichaelS96 [8] | Singly Linked List | Lock-free | $\Omega(p)$ |
| MoirNSS05 DBLP:conf/spaa/MoirNSS05 [10] | Linked List with Elemination | | |
| HoffmanSS07 DBLP:conf/opodis/HoffmanSS07 [3] | Linked List with Baskets | | $\Omega(p)$ |
| Ladan-MozesS08 DBLP:journals/dc/Ladan-MozesS08 [7] | Doubly linked list | Lock-free | $\Omega(p)$ |
| MilmanKLLP18 DBLP:conf/spaa/MilmanKLLP18 [9] | Linked List with batching same type ops | Lock-free | |
| GidenstamST10 DBLP:conf/opodis/GidenstamST10 [2] | Linked list of arrays | Lock-free | $\Omega(p + len(array))$ |
| KoganP11 DBLP:conf/ppopp/KoganP11 [6] | Linkedlist with head and tail pointers | Wait-free | $O(p \times \text{bakery-max})$ |

Table 1: Linearizable ME-MD Queus

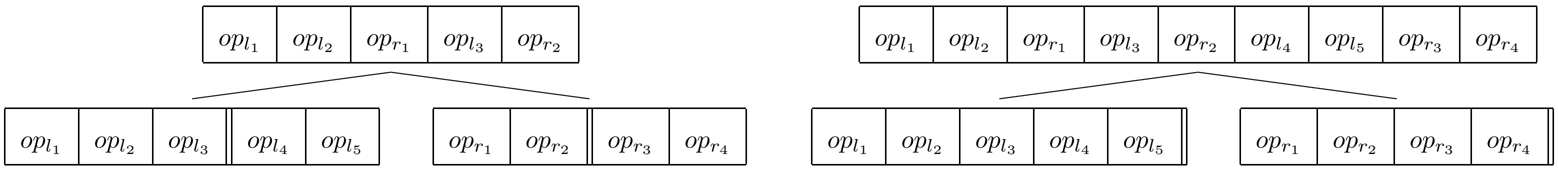
## 2 Universal Construction using Tournament Tree with Big CAS Objects

Jayanti DBLP:conf/podc/Jayanti98a [4] proved an $\Omega(\log p)$ lower bound on the worst-case shared-access time complexity of $p$-process universal constructions. He also introduced DBLP:conf/podc/ChandraJT98 [1] a construction that achieves $O(\log^2 p)$ shared accesses. Here, we first introduce a universal construction using $O(\log p)$ CAS operations DBLP:conf/fsttcs/JayantiP05 [5]. We use the universal construction as a stepping stone towards our queue algorithm, so we will not explain it in too much detail.

We introduce our universal construction in Algorithm alg1 with a tournament tree with $p$ leaves and height $\log(p)$ is shared among $p$ processes. Nodes are CAS objects, and each leaf is assigned to one process exclusively. Each CAS object stores a sequence of operations. When process $P_i$ wishes to apply an operation *op* on the implemented object, it appends *op* to its assigned leaf and tries to propagate it up to the root. The sequence of operations stored in Node $n$'s CAS object shows the order of the operations propagated up to $n$. The history of operations stored at the root is the linearization ordering. The operation *op* is linearized when it is appended to the root.




Leaf $l_i$ stores the sequence of the operations invoked by $P_i$. The algorithm uses a subroutine Refresh($n$) that concatenates new operations from node $n$'s children (that have not already been propagated to $n$) to the sequence of operations stored in $n$ and tries to




(a) Operations after || are new.

(b) New operations are added to the parent node.

ucexample

Figure 1: Propagate Step in Universal Construction

CAS the new sequence into $n$. In other words, Refresh($n$) tries to append $n$'s children's new operations to $n$'s sequence. After a process adding a new operations to its leaf, it has to propagate new operations up to the root. Propagate($n$) tries to append $n$'s new operations to the root $n$ by recursively calling Refresh($n$). In each step if a Refresh($n$) fails, it means another CAS operation has succeeded; if so, it tries to Refresh($n$) again. If the second attempt fails too, another process has already appended the operations the current Propagate is trying to append. Operations that were in $l_i$ before Propagate($l_i$.parent) was invoked are guaranteed to be added to the root by the time the Propagate($l_i$.parent) finishes.

alg1

**Algorithm** Universal Construction Idea

```
1: response Do(operation op, pid i)
2:     l_i.APPEND(op)
3:     PROPAGATE(parent of l_i)
4:     Run the sequence stored in root
5:     return op's response from line 4
6: end Do

7: void PROPAGATE(node n)
8:     if n==root then return
9:     else if !REFRESH(n) then
10:        REFRESH(n)
11:    end if
12:    PROPAGATE(parent of n)
13: end PROPAGATE

14: boolean REFRESH(node n)
15:    old= READ(n)
16:    new= ops that n's children contain but old does not
17:    new= old·new
18:    return n.CAS(old, new)
19: end REFRESH
```

O($\log n$) CAS operations are invoked to do a Propagate, but the CAS words store sequences of unbounded length. The problem is that we are trying to store unbounded sequence of operations in each node $n$ (see Figure ~~2~~ fig::uc). However, to compute the result of an operation, we only use the total ordering that is stored at the root. Although we use a similar construction for our queue implementation, we develop an implicit representation of the sequence of operations, so that we can use reasonable size CAS objects and still achieve polygarithmic step complexity.

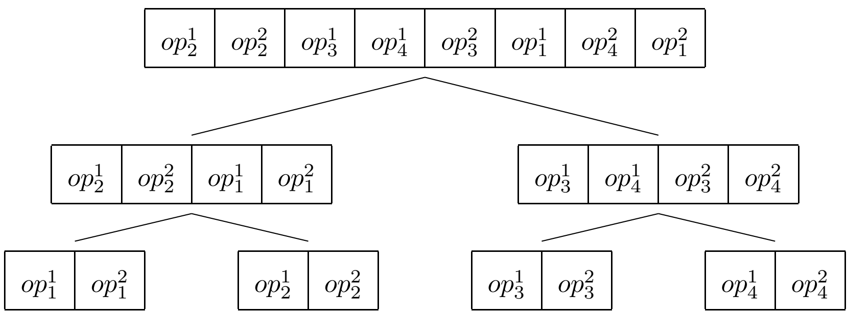



fig::uc

Figure 2: Universal Construction: $op_j^i$ denotes the $i$th operation from process $j$. In each node, we store the ordering of all the operations propagated up to it.

# 3 Block Tree

We apply two ideas from universal construction to create a new linearizable data structure agreeing on a sequence of elements among processes. First, there is a shared tournament tree among processes, in which each process appends its element to its leaf in the tree and then tries to propagate it up to the root by performing Refresh() operations at each node. Second, each operation is linearized when its element is appended to the root.

## 3.1 Sequence of propagated operations

The basic idea behind the universal construction is to create a linearization of operations invoked by processes. If we design a fast shared data structure in which processes can append elements to a sequence, we can use it to implement practical, fast implementaion of an object $O$ by using the sequence data structure to keep track of the sequence of operations on $O$. In the following sections, we introduce our solution called Block Tree.

## ec::block 3.2 Sequence of Sets of Concurrent Operations

In the universal construction, we order new concurrent operations at each Refresh() and maintain that order in the path up to the root. However, we can instead keep track of sets of concurrent operations and create the total ordering of all operations at the root (see Figure fig::set 3).

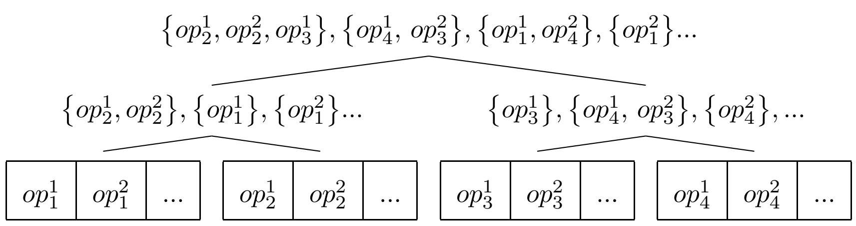



fig::set Figure 3: In each internal node, we store the set of all the operations propagated together, and one can arbitrarily linearize the sets of concurrent operations among themselves. Since we linearize operations when they are added to the root, ordering the blocks in the root is important.

The definition of linearizability allows concurrent operations to be reordered arbitrarily. Thus, a group of concurrent operations can be appended to our root sequence as one block without specifying the order among the operations.

## 3.3 Using arrays instead of big CAS Objects

We used unbounded CAS objects storing sequences as big words in the universal construction. One can represent sequences as arrays to overcome this implementation problem. Each array element will store one of the blocks of concurrent operations described in section subsec::block 3.2.

## 3.4 Augmenting Tree to make Refresh() Step faster

Copying operation sequences from children to their parent in a Refresh() takes time proportioned to the number of operations being copied. This is time-consuming, so we propose a way to augment the tree to calculate lines 15,16 in $O(\log p)$ steps which reads new operation and concats them with old operations. Instead of representing the set of operations by explicitly listing them in a node, we represent a set of operations implicitly by recording which of the children's sets were unioned to create the set. Having operation sequences stored at leaves, we can deduce a set of operations in a node using this implicit representation. (see Figure fig:block 4.)

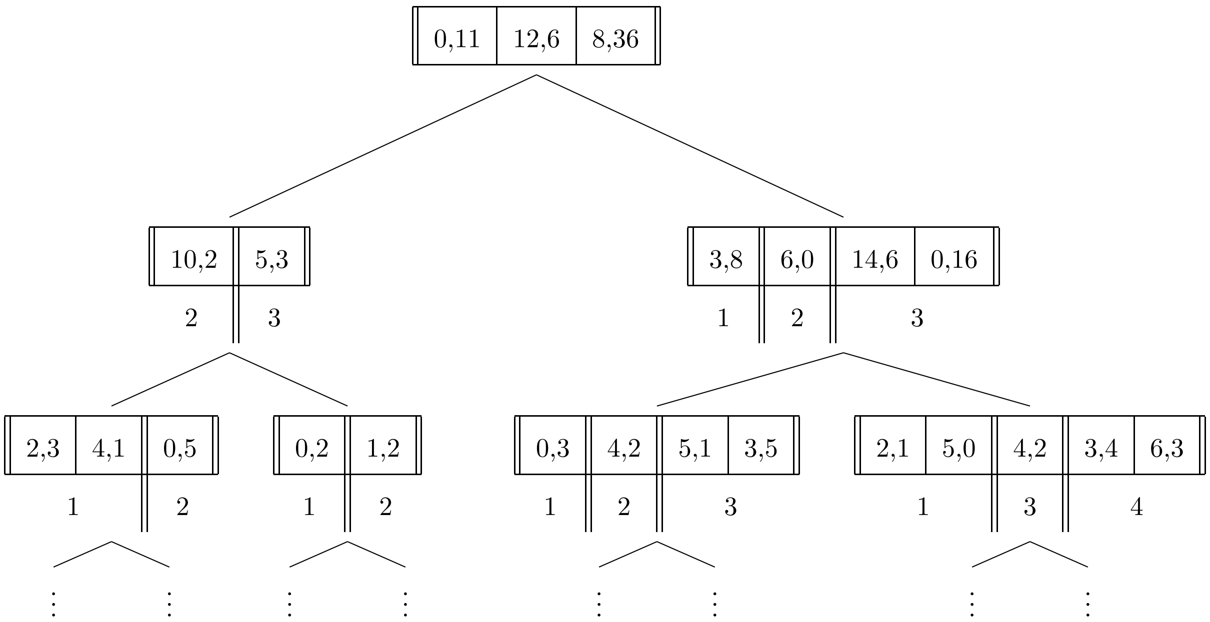



fig:block

Figure 4: Showing concurrent operation sets with blocks. Each block consists of a pair(left, right) indicating the number of operations from the left and the right child, respectively. Block (12,6) in the root contains blocks (10,2) from the left child and (6,0) from the right child. Blocks between two lines || are propagated together to the parent. For example, Blocks (2,3) and (4,1) from the leftmost leaf and (0,2) from its sibling are propagated together into the block (10,2) in their parent. The number underneath a group of blocks in a node indicates which block in the node's parent those blocks were propagated to.

Each block $b$ in node $n$ is the aggregation of blocks in the children of $n$ that are newly read by thePropagate() step that created block $b$. For example, the third block in the root (8,36) is created by merging block (5,3) from the left child and (14,6) and (0,16) from the right child. Block (5,3) also points to elements from blocks (0,5) and (1,2).

**Definition 1.** {Existence of an operation in a block} Operation op exists in block b if it has propagated up to block b.

**Definition 2.** {Subblock} The blocks that are aggregated into block $b$ in a Propagate() step are called subblocks of $b$. Block $b_1$ is a subblock of $b_2$ if and only if $b_1$ is a block in node $v$ and in $b_2$ is a block in the parent of $v$ and $b_1$'s elements exits in $b_2$'s elements.

We choose to linearize operations in a block from the left child before those from the right child as a convention. Operations within a block of the root can be ordered in any way that is convenient. In effect, this means that if there are concurrent new blocks in a Refresh() step from several processes we linearize them in the order of their process ids. So for example operations aggregated in block (10,2) are in the order (2,3),(4,1),(0,2). All blocks from the left child with come before the right child and the order of blocks of each child is preserved among themselves.

In a Propagate() invocation path from a leaf to root, there will be Refresh() steps with merges from $2, 4, 8, ..., p$ processes. So in a complete propagation, at most $2p$ blocks are merged into one block. `(maybe useful for analysis)`

### 3.5 Using pointers and prefix sum to make GetIndex($i$) faster

GetIndex($i$) returns the $i$th operation stored in the block tree sequence. We do that by finding the block $b_i$ containing $i$th element in the root, and then recursively finding the subblock of $b_i$ which contains $i$th element. To make this recursive search faster, instead of iterating over all elements in sequence of blocks we store prefix sum of number of elements in the blocks sequence and pointers to make BSearch faster.

Furthermore, in each block, we store the prefix sum of left and right elements. Moreover, for each block, we store two pointers to the last left and right subblock of it (see fig fig::pointer 6 and fig:prefix 5).

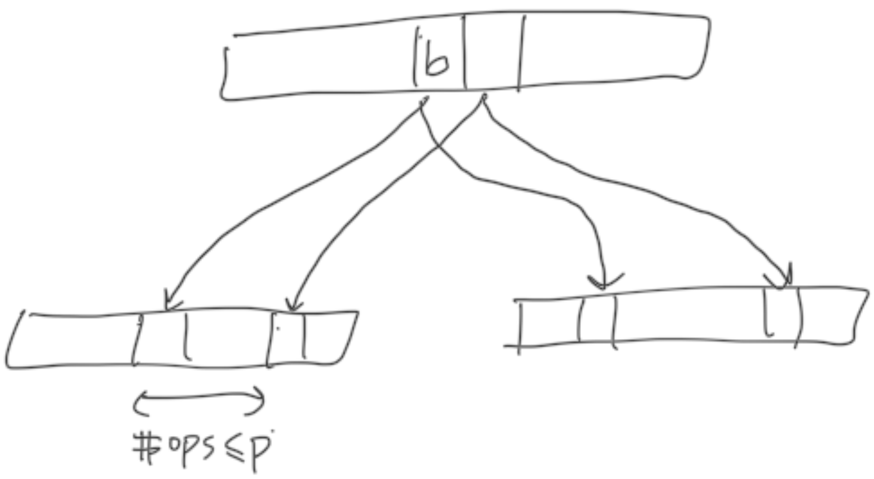
| | | | |
|---|---|---|---|
| 3,8 | 6,0 | 14,6 | 0,16 |
| 0,0 | 3,8 | 9,8 | 23,14 |

| | | | |
|---|---|---|---|
| 0,3 | 4,2 | 5,1 | 3,5 |
| 0,0 | 0,3 | 4,5 | 9,6 |

| | | | | |
|---|---|---|---|---|
| 2,1 | 5,0 | 4,2 | 3,4 | 6,3 |
| 0,0 | 2,1 | 7,1 | 11,3 | 14,7 |

ig:prefix

Figure 5: Using Prefix sums in blocks. When we want to find block b elements in its children, we can use binary search. The number below each block shows the count of elements in the previous blocks.




:: pointer

Figure 6: Block have pointers to the starting block of theirs for each child.

Starting from the root, GetIndex($i$) BSearches $i$ in the prefix sum array to find block containing $i$th operation, then continues recursively calling GetElement($b, i$) to find $i$th element of block $b$. From lemma lem:block_size ?? we know a block size is at most $p$. So BSearch takes at most (O)($\log p$), since with knowing pointers of a block and its previous block we can determine the base `(domain ?)` to search and its size is O($p$).

### 3.6 Block Tree Algorithm

Our Block Tree is a linearizable implementation of a data structure that stores a sequence of elements. It has two methods (see Algorithm alg2 ), Append($e$) which appends element $e$ to the sequence, and Get($i$) which returns the $i$th element in the sequence.

**Design of a block tree** Each process is assigned to a leaf in a shared tournament tree. Thus, for example, the leaf node for process $p_i$ contains an array of elements by $p_i$ in the order they were invoked. Each internal node of the tree contains an array of blocks of elements. Block $b$ in node $n$ is created in a Propagate() step and is merged block of new blocks at the time of Propagate() reading $n$'s children blocks. Each block consists of pointers left and right, to the last block merged into itself from left and right child in that order. Moreover, two numbers, left and right, indicate the count of elements in the blocks from the left and right child consecutively. Furthermore, prefix left, and right can be computed from the prefix sum of left and right values. Elements of block $b$ can be determined recursively (GetElements($b$)). The $i$th element in the sequence can be determined in O($\log^2 p$) steps by recursively finding $i$th element in block $b$ (GetElement($i$)) After element $e$ is propagated (appended to a block int the root), its index can be computed with GetIndex($op$).

In order to compute elements of a block faster we store prefix-sum blocks(block i has tuple(right-sum=#right ops in previous block, left-sum=#left ops in previous blocks)[See Figure fig:prefix 5]. Here is the algorithm to get elements of a block.

**Specification** A block tree is a shared data structure that stores a sequence of elements. It has two methods `Append(e)` and `Get(i)`. `Append(e)` adds `e` to the end of the sequence and returns the index of `e` in the sequence. `Get(i)` returns `i`th element stored in the sequence.

**SubBlock** Block `s` is a subblock of `b` if `s` is between blocks `start..end` in `n` from Lines 41,42 of `CreateBlock()`.

**Membership** Element `e` is a member of block `b` in:

- internal node `n`, if `e` is a memeber of `s` that `s` is a subblock of `b`.
- leaf node `n`, if `e` belongs to $\texttt{n.dir.blocks[b}'\texttt{.end}_{\texttt{dir}}\texttt{+1..b.end}_{\texttt{dir}}\texttt{]}$ for `dir` $\in$ {`left, right`} which b$'$ is the previous block of `b` in `n`.

**Order of elements inside node** Element `d` is before element `e` in node `n`, if:

- The block containing d is before the block containing e.
- e and d are in the same block and d is in the left child and e is in the right child.
- d is before e in the same child's order.

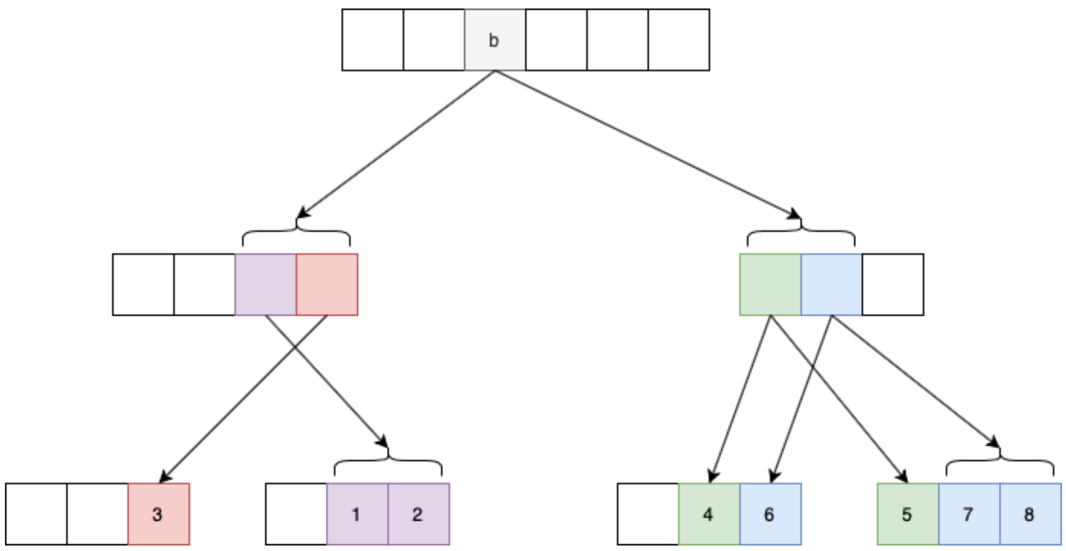
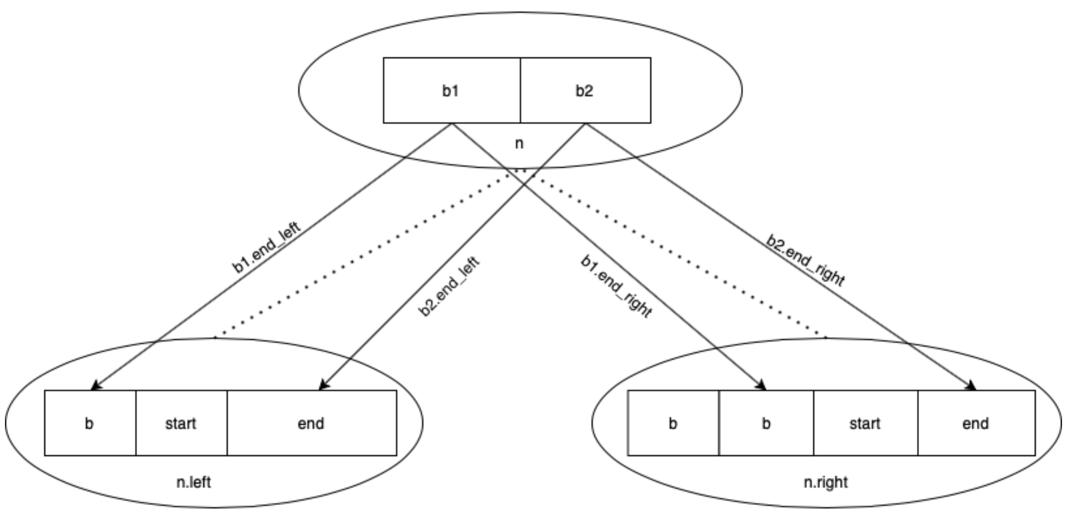



Figure 7: Order of elements in b: elements in leaves are ordered with numerical order in the drawing.

`CreateBlock()` `CreateBlock(n)` returns a block containing new operations of `n`'s children. $\texttt{b}'\texttt{.end}_{\texttt{left}}$ stores the index of the rightmost subblock of left child of `b`'s previous block. Other attributes are assigned values followed by definition.




`eateBlock`

Figure 8: Snapshot of a CreateBlock()

**Double Refresh** Elements in `n`'s children's blocks before line `13` are guaranteed to be in `n`'s blocks after Line `15`.

*Proof.* `CreateBlock()` reads blocks in the children that does not exist in the parent and aggregates them into one block. If a `Refresh` procedure returns true it means it has appended the block created by `CreateBlock()` into the parent node's sequence. So suppose two `Refresh`es fail. Since the first `Refresh` was not successful, it means another CAS operation by a `Refresh`, concurrent to the first`Refresh`, was successful before the second `Refresh`. So it means the second failed `Refresh` is concurrent with a successful `Refresh` that assuredly has read block before the mentioned line `13`. After all it means if any of the `Refresh` attempts were successful the claim is true, and also if both fail the mentioned claim still holds. ☐

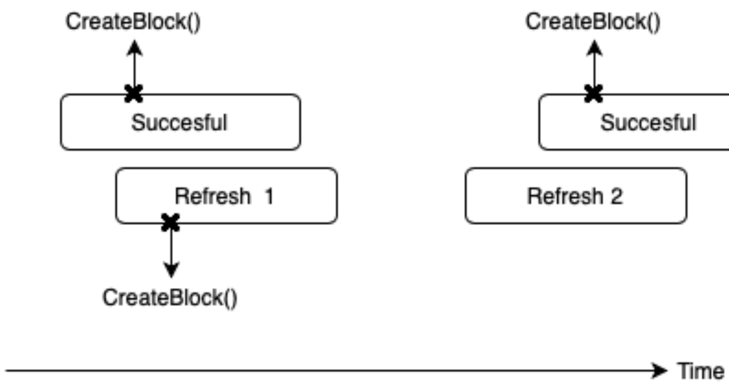



Figure 9: The second failed Refresh is assuredly concurrent to a Successful `Refresh()` with `CreateBlock` line after first failed `Refresh`'s `CreateBlock()`.

**Disjunction** Blocks in node `n`'s contain disjoint sets of elements.

*Proof.* Without loss of generality, assume blocks `b1, b2` contain common element `e` from the left child, and `b2` is after `b1` in `n`'s sequence of blocks. So block start of b2's `CreateBlock()` is after block end of `b1`'s end. Since `b2`'s start is the end of the block before itself, it cannot be before `b1`'s end. ☐

**Total Order** Sequence represented by the Block Tree is the sequence of the blocks stored in the root.

**Linearization Points** `Get(i)` is linearized when it terminates. `Append(e)` is linearized right after when a block containing `e` is appended to the root, if there are multiple elements appended together, they are linearized by the defined order in the root.

**Subblocks Upperbound** Block b has at most $p$ subblocks.

*Proof.* If there are more than $p$ subblocks, then there is more than one block from process pl. `Append(e)` finishes after propagating and appending `e` to the root(line 9). So these blocks cannot be appended to root already, so pl has invoked two concurrent `Append()`s(line 1) without terminating the first one. ☐

**Computing `Get(n, b, i)`** To find the `i`th element in block `b` of node `n`, we search among subblocks of `b` that is bounded by $p$. Subblocks of a block are within the start and end block of the `CreateBlock()` procedure of it.

**How `Refresh(n)` works.**

1. Read n's counter and head
2. Create block b
3. CAS b into n

4. If previous succeed:

   (a) Update sup of b's ending subblocks

   (b) Increment children's counters

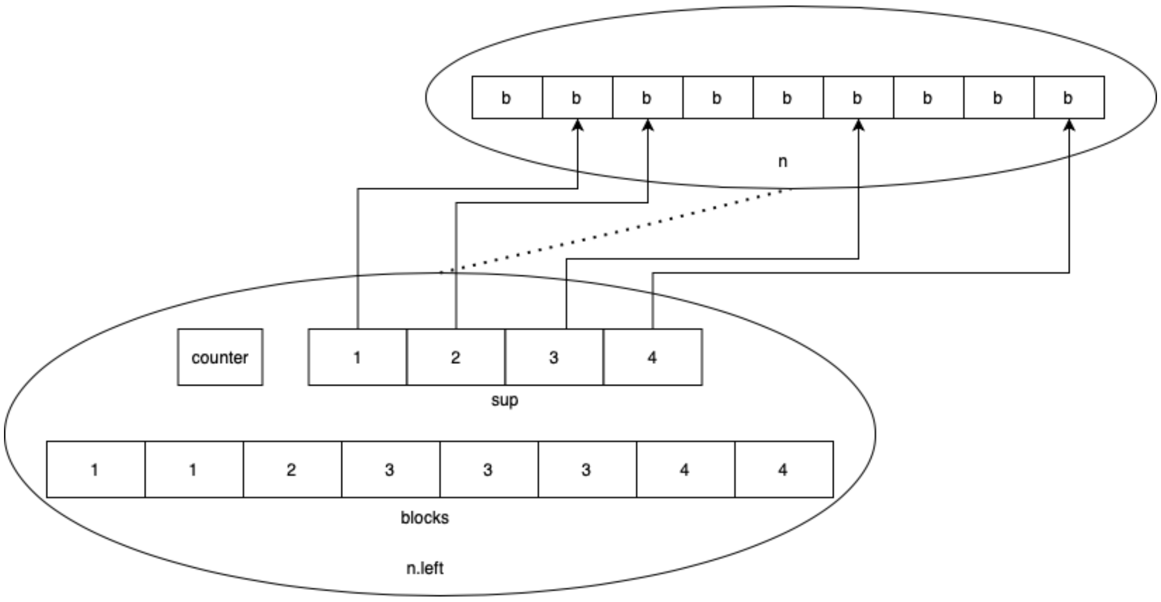



Figure 10: `Sup` and `timer` in a node, numbers on blocks are their `time` values.

**Computing superblock**

1. Value read for `super[b.time]` in line 71 is not null.

   *Proof.* `Index()` is invoked after finishing `Propagate()` in line 10. For each value $\mathtt{c_{dir}}$ read in lines 23, `super` is set before incrementing in lines 26,27. □

2. `super[]` preserves order from child to parent; if in a child block `b` is before `c` then `b.time` $\leq$ `c.time` and `super[b.time]` $\leq$ `super[c.time]`

   *Proof.* Follows from the order of lines 37, 26, 27. □

3. `super[i+1]-super[i]`$\leq p$

   *Proof.* In a Refresh with successful CAS in line 24, `super` and `counter` are set for each child in lines 26,27. Assume the current value of the counter in node `n` is `i+1` and still `super[i+1]` is not set. If an instance of successful `Refresh(n)` finishes `super[i+1]` is set a new value and a block is added after `n.parent[sup[i]]`. There could be at most $p$ successful unfinished concurrent instances of `Refresh()` that have not reached line 27. So the distance between `super[i+1]` and `super[i]` is less than $p$. □

4. Superblock of `b` is within range $\pm 2p$ of the `super[b.time]`.

   *Proof.* `super[i]` is the index of the superblock of a block containing block b. It is trivial to see that `n.super` and `n.b.counter` are increasing. `super(b)` is the real superblock of b. `super(t]` is the index of the superblock of the last block with time `t`. If `b.time` is `t` we have:

$$super[t] - p \leq super[t-1] \leq super(t-1] \leq super(b) \leq super(t+1) \leq super(t+1] \leq super[t] + p$$

□

alg2

**Algorithm** Block Tree

```
Structure
▶ element e
  • int pid
  • int loc: location in l_pid.ops
▶ node n
  • *node left, right, parent
  • block[] blocks: blocks stored in n
  • int head= 1: index of first empty cell of blocks
  • int counter= 0
  • int[] super: super[i] stores the index of a superblock in parent that
    contains some block of this node whose time is i
▶ leaf node l_i extends node
  • operation[] ops: elements that are invoked to Append() to the blok
    tree by process i
▶ block b
  • int num_left, sum_left
      #operations from the left subblocks of b, prefix sum of num_left
  • int num_right, sum_right
      #operations from the right subblocks of b, prefix sum of num_right
  • int sum
      # operation in b
  • int end_left, end_right
      index of b's last subblock
  • int time

int Append(operation op, int pid)
    op.loc= l_pid.head
    block b= new(block)
    b.time= op.loc
    b.sum= 1
    l_pid.ops[op.loc]= op
    l_pid.blocks[op.loc]= b
    op.head+= 1
    Propagate(l_pid.parent)
    return Index(l_pid, op.loc, b)
end Append

void Propagate(node n)
    if not Refresh(n) then
        Refresh(n)
    end if
    if n is root then return
    end if
    Propagate(n.parent)
end Propagate

boolean Refresh(node n)
    i= n.head
    c= n.counter
    new, c_left, c_right= CreateBlock(n, i+1, c)
    if CAS(n.blocks[i+1], null, new) then
        for each dir in {left, right} do
            CAS(n.dir.super[c_dir], null, i+1)
            CAS(n.dir.counter, c_dir, c_dir+1)
        end for
        return true
    else
        return false
    end if
    CAS(n.head, i, i+1)
end Index

block, int, int CreateBlock(node n, int i, int c)
                                  ▷ Creates a block to insert into n.blocks[i]
    block b= new(block)
    b.time= c
    for each dir in {left, right} do
        j= n.dir.head
        start= n.blocks[i-1].end_dir
        end= n.dir.blocks[j]
        c_dir= n.dir.blocks[j].time
        b.end_dir= j
        b.num_dir= end.sum - start.sum
        b.sum_#dir= n.blocks[i-1].sum_#dir + b.num_dir
    end for
    b.sum= b.sum_left + b.sum_right
    return b, c_left, c_rightt
end CreateBlock
```

**Algorithm** Block Tree Continued

```
element GET(int i)
    res= BSEARCH(root, sum, i, 0, root.head)
    return GET(root, res, i-root.blocks[res-1].sum)
end GET

⇝ Precondition: n.blocks[start..end] contains a block with field f ≥ i
int BSEARCH(node n, field f, int i, int start, int end)
▷ Searches the value i of the given prefix sum type on the node n in the domain from start to end blocks. f is one of sum, sum_left, sum_right. Does binary search on field f of the blocks stored in n.blocks and returns the index of the leftmost block in n.blocks[start..end] whose field f is ≥ i.
    return result block's index
end BSEARCH

element GET(node n, int b, int i)                                    ▷ Returns the ith operation in bth block of node n
    if n is leaf then return n.ops[i]
    else
        if i ≤ n.blocks[b].num_left then                             ▷ i exists in left child of n
            subBlock= BSEARCH(n.left.sum, i, n.blocks[b-1].end_left+1, n.blocks[b].end_left)
            return GET(n.left, subBlock, i-n.left.blocks[subBlock-1].sum)
        else
            i= i-n.blocks[b].num_left
            subBlock=BSEARCH(n.right.sum, i, n.blocks[b-1].end_right, n.blocks[b].end_right)
            return GET(n.right, subBlock, i-n.right.blocks[subBlock-1].sum)
        end if
    end if
end GET

⇝ Precondition: ith operation innode n is in block b of node n.
index INDEX(node n, int i, int b)                                    ▷ Returns rank in the root of the ith operation in node n.
    if n is root then return i
    else
        dir= (n.parent.left==n)?  left:  right
        superBlock= BSEARCH(n.parent, n.sum_dir, i, sup[n.blocks[b].time]-p, sup[n.blocks[b].time]+p)
        if dir is left then
            i+= n.parent.blocks[superBlock-1].sum_right
        else
            i+= n.parent.blocks[superBlock].sum_left + n.parent.blocks[superBlock].sum_left + n.blocks[nBlock-1].sum
        end if
        return INDEX(n.parent, i, superBlock)
    end if
end INDEX
```

# 4 Implementing Queue using Block Tree

## 4.1 Idea in a nutshell

With the ideas introduced in block tree we are going to create a shared wait-free queue with $O(\log^2 p + \log n)$ steps. A queue stores a sequence of elements and supports two operations, enqueue and dequeue. `Enqueue(e)` appends element `e` to the sequence stored.`Dequeue()` removes and returns the first element among in the sequence. If the queue is empty it returns `null`. Knowing index $i$ is the tail of the queue, we can return the dequeue response using `Get(i)`. So in the rest we modify block tree to compute `i` for each `Dequeue()` to achieve a FIFO queue.

Next, we describe how to use block tree to implement queues. The block tree, maintains the history of all operations, not only the current state of the queue. Now consider the following history of operations. What should each `Dequeue()` return?

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `ENQ(5)` | `ENQ(2)` | `DEQ()` | `ENQ(3)` | `DEQ()` | `DEQ()` | `DEQ()` | `ENQ(4)` | `ENQ(6)` | `DEQ()` |

Table 2: An example histoy of operations on the queue

**Definition 3.** A non-null dequeue is one that returns a non-null value.

In the example above, `Dequeue()` operations return `5, 2, 3, null, 4` in order. Before `ENQ(4)` the queue gets empty so the last `DEQ()` returns null. If the queue is non-empty and $r$ `Dequeue()` operations have returned a non-null response, then $i$th `Dequeue()` returns the input of the $r+1$th `Enqueue()`. So, in order to answer a Dequeue, it's sufficent to know the size of the queue and the number of previous non-null dequeues.

In the Block Tree, we did not store the sequence of operations explicitly but instead stored blocks of concurrent operations to optimize `Propagate()` steps and increase parallelism. So now the problem is to find the result of each Dequeue. From lemma lem:block_size ?? we know we can linearize operations in a block in any order; here, we choose to decide to put Enqueue operations in a block before Dequeue operations. In the next example, operations in a cell are concurrent. `DEQ()` operations return `null, 5, 2, 1, 3, 4, null` respectively. We will next describe how these values can be computed efficiently.

| | | | |
|---|---|---|---|
| `DEQ()` | `ENQ(5)`, `ENQ(2)`, `ENQ(1)`, `DEQ()` | `ENQ(3)`, `DEQ()` | `ENQ(4)`, `DEQ()`, `DEQ()`, `DEQ()`, `DEQ()` |

Table 3: An example history of operation blocks on the queue

Now, we claimed that by knowing the current size of the queue and the number of non-null dequeue operations before the current dequeue, we could compute the index of the resulting `Enqueue()`. We apply this approach to blocks; if we store the size of the queue after each block of operations happens and the number of non-null dequeues dequeues till a block, we can compute each dequeue's index of result in $O(1)$ steps.

| | `DEQ()` | `ENQ(5)`, `ENQ(2)`, `ENQ(1)`, `DEQ()` | `ENQ(3)`, `DEQ()` | `ENQ(4)`, `DEQ()`, `DEQ()`, `DEQ()`, `DEQ()` |
|---|---|---|---|---|
| #enqueues | 0 | 3 | 1 | 1 |
| #dequeues | 1 | 1 | 1 | 4 |
| #non-null dequeues | 0 | 1 | 2 | 5 |
| size | 0 | 2 | 2 | 0 |

Table 4: Augmented history of operation blocks on the queue

Size and the number of non-null dequeues for $b$th block could be computed this way:

`size[b]= max(size[b-1] +enqueues[b] -dequeues[b], 0)`

`non-null dequeues[b]= non-null dequeues[b-1] +dequeues[b] -size[b-1] -enqueues[b]`

Given `DEQ` is in block `b`, `response(DEQ)` would be:

```
(size[b-1]- index of DEQ in the block's dequeus >=0) ? ENQ[non-null dequeus[b-1]+ index of DEQ in the block's dequeus]
: null;
```

# 5 Main Algorithm

**Specification** A Queue is a shared data structure that stores a sequence of elements. It has two methods `Enqueue(e)` and `Dequeue()`. `Enqueue(e)` adds `e` to the end of the sequence. `Dequeue()` returns the first element stored in the sequence and removes it from the sequence.

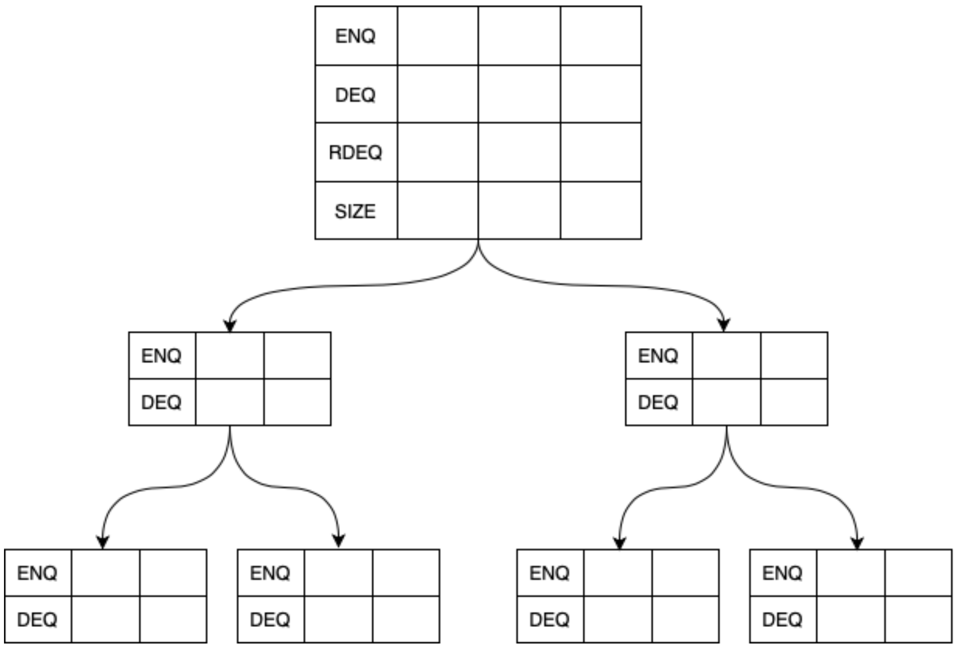



Figure 11: Fields stored in the Queue nodes.

ig::queue

## 5.1 Pseudocode description

**Tournament Tree** In order to reach an agreement on the order of operations among $p$ processes, we use a Tournament Tree. Leaf $\texttt{l}_\texttt{i}$ is assigned to a process `i`. Each process adds *op* to its leaf. In each internal node an ordering of operations in its subtree is stored. All processes agree on the total ordering of all operations stored in the root. This ordering will be the linearization of the operations.

**Implicit Storing Blocks** For efficiency, instead of storing explicit sequence of operations in nodes of the Tournament Tree, we use Blocks. A Block is a constant size object that implicitly represents a sequence of operations. In each node there is an array of Blocks.

**Definition 4** (Block)**.** A block is an object that stores some statistics described in Algorithm Queue.

**Definition 5** (Subblock)**.** Block `b` is a subblock of `n.blocks[i]` if it is in $\texttt{n.left.blocks[n.blocks[i-1].end}_{\texttt{left}}\texttt{+1..n.blocks[i].end}_{\texttt{left}}\texttt{]}$ or $\texttt{n.right.blocks[n.blocks[i-1].end}_{\texttt{right}}\texttt{+1..n.blocks[i].end}_{\texttt{right}}\texttt{]}$.

Block $b$ contains subblocks in the left and right children. WLOG left subblocks of $b$ are some consecutive blocks in the left child starting from where previous block of $b$ has ended to the the end of $b$. See Figure 8 .

**Definition 6** (Membership of an operation in a block)**.** Operation `e` is a member of block `b` in:

- leaf node `n`, if `e` belongs to `n.ops[b's index]`.

algQ

**Algorithm** Queue

▶ *Operation*

- *Object* element: argument of an enqueue, if null this is a dequeue
- *int* loc: location in the leaf's ops

▶ *Node*

- **Node* left, right, parent
- *Block[]* blocks: index 0 contains an empty block with all fields equal to 0
- *int* head= 1: index of the first empty cell of blocks
- *int* counter= 0
- *int[]* super: super[i] stores the index of a superblock in parent that contains some block of this node whose time is field i

▶ *Leaf* extends *Node*

▷ $l_i$ is the Leaf for process i and i is in range 0 to p-1

- *Operation[]* ops: invoked operations

▶ *Root* extends *Node*

- override *Root Block[]* blocks

▶ *Block*

- *int* $num_{enq\text{-}left}$, $sum_{enq\text{-}left}$

  #enqueues from subblocks in left child, prefix sum of $num_{enq\text{-}left}$
- *int* $num_{deq\text{-}left}$, $sum_{deq\text{-}left}$

  #dequeues from subblocks in left child, prefix sum of $num_{deq\text{-}left}$
- *int* $num_{enq\text{-}right}$, $sum_{enq\text{-}right}$

  #enqueues from subblocks in right child, prefix sum of $num_{enq\text{-}right}$
- *int* $num_{deq\text{-}right}$, $sum_{deq\text{-}right}$

  #dequeues from subblocks in right child, prefix sum of $num_{deq\text{-}right}$
- *int* $num_{enq}$, $num_{deq}$

  # enqueue, dequeue operations in the block
- *int* $sum_{enq}$, $sum_{deq}$

  sum of # enqueue, dequeue operations in blocks up to this one
- *int* num, sum

  total # operations in block, prefix sum of num
- *int* $end_{left}$, $end_{right}$

  index of this block's last subblock in the left and right child
- *int* group

▶ *Root Block* extends *Block*

- *int* size

  size of queue after this block's operations finish
- *int* $sum_{non\text{-}null\ deq}$

  count of non-null dequeus up to this block

```
void Enqueue(Object e)
    op= NEW(operation)
    op.element= e
    Append(op)
end Enqueue

Object Dequeue()
    op= NEW(operation)
    Append(op)                                  ▷ ith DEQ in bth block
    <i, b>= Index(l_pid, op.loc, 1)
    res= ComputeHead(i, b)      ▷ Index of the enqueue whose argument should be returned
    if res is null then
        return null
    else
        return Get(res)
    end if
end Dequeue

int ComputeHead(int i, int b)   ▷ Computes head of the queue when ith dequeue in bth block occurs. The dequeue should return the argument of the head enqueue.
    if root.blocks[b-1].size + root.blocks[b].num_enq - i < 0 then
        return -1
    else return root.blocks[b-1].sum_non-null deq + i
    end if
end ComputeHead

void Append(operation op)
    pid= id of process performing Append
    op.loc= l_pid.head
    op.head+= 1
    block b= NEW(block)
    b.group= op.loc
    if op.element==null then b.sum_deq=1
    else b.sum_enq=1
    end if
addOP
    l_pid.ops[op.loc]= op
    l_pid.blocks[op.loc]= b
    Propagate(l_pid.parent)
end Append
```

**Algorithm** Queue Continued

```
void Propagate(node n)
    if not Refresh(n) then
        Refresh(n)
    end if
    if n.parent is null then
        Propagate(n.parent)
    end if
end Propagate

boolean Refresh(node n)
    h= n.head
    t= n.counter
    <new, t_left, t_right>= CreateBlock(n, h, t)
    if new.num==0 then return true
    else if CAS(n.blocks[h], null, new) then
        for each dir in {left, right} do
            CAS(n.dir.super[t_dir], null, h+1)
            CAS(n.dir.counter, t_dir, c_dir+1)
        end for
        CAS(n.head, h, h+1)
        return true
    else
        CAS(n.head, h, h+1)
        return  false
    end if
end Refresh

element Get(int i)                              ▷ Returns ith Enqueue.
    res= BSearch(root, sum_enq, i, 0, root.head)
    return Get(root, res, i-root.blocks[res-1].sum_enq)
end Get

    ⇝ Precondition: n.blocks[start..end] contains a block with field f ≥ i
int BSearch(node n, field f, int i, int start, int end)
                ▷ Does binary search for the value i of the given
    prefix sum feild. f is one of sum, sum_left, sum_right. Returns the index of
    the leftmost block in n.blocks[start..end] whose field f is ≥ i.
end BSearch

<Block, int, int> CreateBlock(node n, int i, int t)
    ▷ Creates a block to insert into n.blocks[i] with time field=t. Returns
the created block as well as values read from each child counter feild.
    block b= new(block)
    b.group= t
    for each dir in {left, right} do
lastLine    lastIndex= n.dir.head
prevLine    prevIndex= n.blocks[i-1].end_dir
        lastBlock= n.dir.blocks[lastIndex]
        prevBlock= n.dir.blocks[prevIndex]
        t_dir= n.dir.counter
        b.end_dir= lastIndex
        b.num_enq-dir= lastBlock.sum_enq - prevBlock.sum_enq
        b.num_deq-dir= lastBlock.sum_deq - prevBlock.sum_deq
        b.sum_enq-dir= n.blocks[i-1].sum_enq-dir + b.num_enq-dir
        b.sum_deq-dir= n.blocks[i-1].sum_deq-dir + b.num_deq-dir
    end for
    b.num_enq= b.num_enq-left + b.num_enq-right
    b.num_deq= b.num_deq-left + b.num_deq-right
    b.num= b.num_enq + b.num_deq
    b.sum= n.blocks[i-1].sum + b.num
    if n.parent is null then
        b.size= max(root.blocks[i-1].size + b.num_enq - b.num_deq, 0)
        b.sum_non-null deq= root.blocks[i-1].sum_non-null deq + max(
b.num_deq - root.blocks[i-1].size - b.num_enq, 0)
    end if
    return b, t_left, t_rightt
end CreateBlock
```

**Algorithm** Queue Continued

```
⇝ Precondition: n.blocks[b] contains ≥i enqueues.
element Get(node n, int b, int i)                              ▷ Returns the ith Enqueue in bth block of node n
    if n is leaf then return n.ops[b]
    else
        if i ≤ n.blocks[b].num_enq-left then                                     ▷ i exists in left child of n
            subBlock= BSearch(n.left, sum_enq, i, n.blocks[b-1].end_left+1, n.blocks[b].end_left)
            return Get(n.left, subBlock, i-n.left.blocks[subBlock-1].sum_enq)
        else
            i= i-n.blocks[b].num_enq-left
            subBlock=BSearch(n.right, sum_enq, i, n.blocks[b-1].end_right+1, n.blocks[b].end_right)
            return Get(n.right, subBlock, i-n.right.blocks[subBlock-1].sum_enq)
        end if
    end if
end Get

⇝ Precondition: bth block of node n has propagated up to the root and ith dequeue resides in node n is in block b of node n.
<int, int> Index(node n, int b, int i)      ▷ Returns the order in the root among dequeus, of ith dequeue in bth block of node n
    if n is root then return root.blocks[b-1]+i, b
    else
        dir= (n.parent.left==n)? left: right
        superBlock= BSearch(n.parent, n.sum_deq-dir, i, super[n.blocks[b].group]-p, super[n.blocks[b].group]+p)
        if dir is left then
            i+= n.parent.blocks[superBlock-1].sum_deq-right
        else
            i+= n.parent.blocks[superBlock-1].sum_deq + n.blocks[superBlock].sum_deq-left
        end if
        return Index(n.parent, superBlock, i)
    end if
end Index
```

- internal node `n`, if `e` is a member of `s` that `s` is a subblock of `b`.

We store ordering among `operation`s in the tournament tree constructed by `node`s. In each `node` we store pointers to its relatives, an array of `block`s and an index to the first empty `block`. Furthermore in `leaf` nodes there is an array of `operations` where each `operation` is stored in one cell with the same index in `blocks`. There is a `counter` in each `node` incrementing after a successful `Refresh()` step. It means after that some bunch of `block`s in a node have propagated into the parent then the `counter` increases. Each new `block` added to a node sets its `time` regarding `counter`. This helps us to know which blocks have aggregated together to a block, not precisely though. We also store the index of the aggregated `block` of a `block` with `time` $i$ in `super[i]`.

In each `block` we store 4 essential stats that implicitly summarize which operations are in the block $\texttt{num}_{\texttt{enq-left}}$, $\texttt{num}_{\texttt{deq-left}}$, $\texttt{num}_{\texttt{enq-right}}$, $\texttt{num}_{\texttt{deq-right}}$. In order to make `BSearch()`es faster we store prefix sums as well and there are some more general stats that help to make pseudocode more readable but not necessary.

To compute the head of the `queue` before a `dequeue` two more fields are stored in the root `size` and $\texttt{sum}_{\texttt{non-null deq}}$. `size` in a `block` shows the number of elements after the `block` has finished and $\texttt{sum}_{\texttt{non-null deq}}$ is the total number of non-null dequeues till the `block`.

`Enqueue(e)` just `append`s an `operation` with `element e` to the `root`. `Dequeue()` appends an `operation` to the root and computes its ordering and the `enqueue operation` containing the head before it calling `ComputeHead()` and then `gets` and returns the `operation`'s element.

`Append(op)` adds `op` to the invoking process's leaf's `ops` and `blocks`, `propagate`s it up to the root and if the `op` is a dequeue returns its order in residing block in the root and the block's index. As we said later `Propagate()` assuredly aggregates new blocks to a block in the parent by calling `Refresh()` two times. `Refresh(n)` creates a block, tries to CAS it into the p`n`'s `blocks` and if it was successful updates `super` and `counter` in both of `n`'s children.

We only want to know the `element` of `enqueue` operations and compute ordering for `dequeue` operations. That's the reason here `Get()` searches between enqueues only and `Index()` returns ordering of a dequeue among dequeues. `Get(n, b ,i)` decides the requested element is in which child of n and continues to search recursively. `index(n, i, b)` calculates the ordering of the given operation in `n`'s parent each step and finally returns the result among total ordering.

## 5.2 Complexity Analysis

`Enqueue()` operations do a constant number of steps and an `Append()` which calls `Propagate()`. Let $p$ be the number of processes and $m$ be the count of operations invoked. In a `Propagate()` step there are a constant number of steps taken at each level and the height of the tree is $\log(p)$. So `Enqueue()` takes $O(\log p)$ steps.

`Get()` searches among all blocks at first and then iterates over a path from the root to a leafof the tree and in each step searches in a domain of size $p$ , so it takes $\log^2 p + \log m$. `Index()` is a path from a leaf to the root which each step calls a $\log p$ step search so it takes $\log^2 p$. `Dequeue()` calls `Append()`, it also calls `Index()` and `Get()` that take $\log^2 p$, $\log^2 p + \log n$.

## 5.3 Linearizability Proof

**Definition 7.** If `n.blocks[i]==b` we call `i` the *index* of block `b` in node `n`. Block `b` is before block `b'` in node `n` if and only if `b`'s index is smaller than `b'`'s. Block `b` is propagated to node `n` or set `S` if `b` is in `n.blocks` or `S` or is a subblock of a block in `n.blocks` or `S`.

**Definition 8.** Block `b` in node `n` is in *Established(n, t)* if `n.head` is greater than `b`'s index at time $t$.

dProgress **Lemma 9** (headProgress). *`n.head` is non-decreasing over time.*

*Proof.* Simply because `n.head` is only incremented. □

dPosition **Lemma 10** (headPosition). *The value read in Line 52(`h=n.head`) might be 1 bit behind the first empty block.*

*Proof.* Because at the end of every `Refresh()` with block size greater than 0 (Lines 53,56) `n.head` is incremented. Maybe some process goes to sleep before incrementing the head, but after sleeping if `h` does not increase then `CAS` in Line 52 is going to be failed and nothing is going to be appended to `n.blocks`. ☐

shedOrder **Lemma 11** (establishedOrder). *If time $t <$ time $t'$, then Established$(n, t) \subseteq$ Established$(n, t')$.*

*Proof.* Because blocks are only appended(not modified) with CAS to `n.blocks[n.head]` and `n.head` is non-decreasing. ☐

eateBlock **Lemma 12** (createBlock). *Suppose `CreateBlock(n, h, x)` is invoked at time $t$. The blocks propagated to Established$(n.left, t)$ and Established$(n.right, t)$ that are not propagated to Established$(n, t)$, are subblock of the block returned by `CreateBlock(n, h, x)`.*

*Proof.* We prove the claim for the left child. Blocks in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` are all the new established operations at time $t$ by definition of Subblock. Line 70 is after $t$ and since the head is only increasing (Lemma 9) the lemma holds. See Figure 8. The right child is the same. ☐

ueRefresh **Lemma 13** (trueRefresh). *Suppose `Refresh(n)`'s `CAS(n.blocks[h], null, new)` returns `true`. Let $t$ be the time `Refresh(n)` is invoked, blocks propgatated to Established$(n.left, t)$ and Established$(n.right, t)$ are propagated to in Established$(n, t)$ after `CAS(n.blocks[h], null, new)`.*

*Proof.* By Lemma 12 `new` contains `n`'s childrens' established blocks before Line 43 which is appended to `n.blocks` by `CAS` in Line 48. ☐

seRefresh **Lemma 14** (falseRefresh). *If instance $r$ of `Refresh(n)` returns `false`, then there is another successful instance $r'$ of `Refresh(n)` that has performed a successful `CAS(n.blocks[h], null, new)`(Line 49) after Line 43(`h= n.head`) of $r$.*

*Proof.* If there is no other concurrent successful `Refresh(n)` then `Refresh(n)` would succeed in Line 48. So there is another `Refresh(n)`, that has to `CAS`ed successfully its block in `n.blocks[h]` after Line 43 of `Refresh(n)`. Otherwise the other `Refresh(n)` should have read `h'>h` instead of `h` for `n.head`(Line 52). ☐

leRefresh **Lemma 15** (Double Refresh). *Consider two consecutive instances $r_1$,$r_2$ of `Refresh(n)` by the same process (Lines 35,36). Let be the time before $r_1$ invoked. After $r_2$'s `CAS` all the blocks propagated to Established$(n.left, t)$ and Established$(n.right, t)$ are in Established$(n, t)$.*

*Proof.*

If Line 35 (first Refresh which we call $R_1$) returns `true`, the claim is held by Lemma 13. If not, then there is another successful instance of `Refresh()` $R_1'$ by Lemma 14. $R_1'$ may or may not have propagated some subblocks of `new` to `n`. It is obvious that the `new` constructed by the second Refresh in Line 36 contains the blocks in `new` by $R_1$ which $R_1'$ did not contain, since `n.head` is only increasing (Maybe Lemma ??). If $R_2$ succeeds by Lemma 13 the claim holds. If not, it is deduced that `n.blocks[h]` was not `null` before $R_2$'s `CAS`. Furthermore, `n.blocks[h]` was `null` before reading `h` by $R_1$. So there is a successful `Refresh()` after the read of `h` in $R_1$ and before the `CAS` of $R_2$. This `Refresh()` contains all the new established operations before Line 35 (Maybe Lemma ??) and by Lemma 13 our claim holds. See Figure ??. ☐

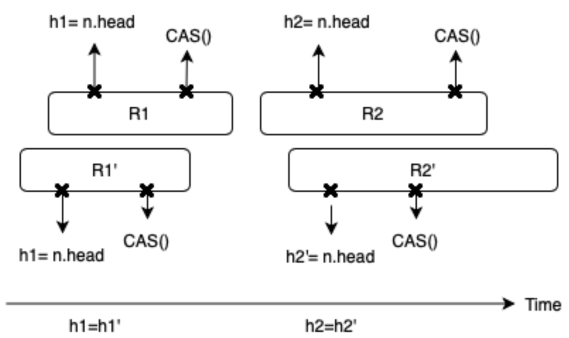



Figure 12: $R_2'$'s `CAS` is executed after `h1=n.head`.

Block `new` is created of new established subblocks of children of `n`(Lemma 12, Line 46). If `CAS` in Line 48 succeeds then by Lemma 13 new established blocks will be in `n`.

**Lemma 16** (Double Refresh)**.** *All operations in* `n`*'s children's blocks before line 35 are guaranteed to be in* `n`*'s blocks after Line 37.*

*Proof.* Suppose block `b` with index `i` is in the the left child of `n` before the line 35. By Lemma 26 it follows that `n.left.head` is greater than `i`. `Refresh()` calls `CreateBlock()` and creates a block from blocks between `n.blocks[n.head].end`$_{\texttt{left}}$ and `n.left.head` in the left child, which contains `b` as well. First it tries to append it in `n.blocks.head` and if it was succsuful it continues recursivley. If not it tries again, and if the second call of `Refresh()` in Line 36 fails. It means there is another `Refresh` which has reah its `i` after the Line 35, so it contains `b` as well. □

`CreateBlock()` reads blocks in the children that do not exist in the parent and aggregates them into one block. If a `Refresh()` procedure returns true it means it has appended the block created by `CreateBlock()` into the parent node's sequence. So suppose two `Refresh`es fail. Since the first `Refresh()` was not successful, it means another CAS operation by a `Refresh`, concurrent to the first `Refresh()`, was successful before the second `Refresh()`. So it means the second failed `Refresh` is concurrent with a successful `Refresh()` that assuredly has read block before the mentioned line `35`. After all it means if any of the `Refresh()` attempts were successful the claim is true, and also if both fail the mentioned claim still holds.

**Lemma 17** (Append)**.** *When* `Append(op)` *is finished,* `op` *appears exactly once in a block of* `root.blocks`*.*

*Proof.* `Append(op)` adds `op` to $\texttt{l}_\texttt{p}$`.blocks`(Line 32) and `Propagate()` recursively propagates `op` up to the root. By lemma 16 we know that operation `op` propagates from child to parent at each level. □

**Lemma 18** (Block Size Upper Bound)**.** *Each block in a node contains at most one operation from each processs.*

*Proof.* Note that `prevIndex` and `lastIndex` defined in lines 69, 70 are the indices defined in the Definition 5. After a block has propagated to the parent the blocks between `prevIndex` and `lastIndex` make up to the parent (Lemma 16). The number of new operations which have not propagated yet to the parent cannot be more than $p$. If so by the law of pigeonholes there is a process which has appended two concurrent operations. □

**Lemma 19** (Subblocks Upperbound)**.** *Each block in a node has at most $p$ subblocks.*

*Proof.* From Line 32 it is induced directly that each block contains at least one operation. Now it follows directly by Lemma 18. □

**Definition 20** (Ordering of operations inside a node)**.** ▶ Note that from Lemma 18 we know there is at most one operation from each process in a given block.

- $E(n,i)$ is the sequence of enqueue operations that are member of `n.blocks[i]` ordered by process id.
- $D(n,i)$ is the sequence of dequeue operations that are member of `n.blocks[i]` ordered by process id.
- $D(n) = D(n,1).D(n,2).D(n,3)...$
- $L = E(root,1).D(root,1).E(root,2).D(root,2).E(root,3).D(root,3)...$

**Theorem 21.** *The queue implementation is linearizable.*

*Proof.* We show that the ordering $L$ stored in the root, satisfies the properties of a linearizable ordering.

1. If $op_1$ ends before $op_2$ begins in $E$, then $op_1$ comes before $op_2$ in $T$.

   ▶ This is followed by Lemma 17. The time $op_1$ ends it is in root, before $op_2$, by Definition 20 $op_1$ is before $op_2$.

2. Responses to operations in $E$ are same as they would be if done sequentially in order of $L$.

   ▶ Enqueue operations do not have any response so it does no matter how they are ordered. It remains to prove Dequeue $d$ returns the correct response according to the linearization order. By Lemma 25 it is deduced that the head of the queue at time of the linearization of $d$ is computed properly. If the Queue is not empty by Lemma 22 we know that the returning response is the computed index element.

□

get **Lemma 22** (Get). *`Get(n,b,i)` returns ith Enqueue in $E(n,b)$.*

*Proof.* It is obvious that `Get(leaf l,b,1)` returns the operations stored in $b$th block of leaf $l$. To find the `i`th enqueue in block `b` of an internal node `n` in line 87 it is decided that it resides in the left child or the right child. This decision is made by Definition of $E(n,b)$. After that Lines 88, 92 search the proper subblocks of `b`. From Definition 5 weknow the subblocks of the $b$th block are within the `prevBlock` and `lastBlock` block of the `CreateBlock()`. □

**Lemma 23** (Index). *Index(n,b,i) returns the rank in the $D(root)$ of ith Dequeue in $D(n,b)$.*

*Proof.* `Index(n,b,i)` computes superblock of $i$th Dequeue in $b$th block of `n` in `n.parent` and then computes the order in $D(n.parent, superblock)$. Then calls `index()` on `n.parent` recursively. It is easy to see why the second is correct. Correctness of computing superblock comes from Lemma 24. □

uperBlock **Lemma 24** (Computing SuperBlock). *If `Index(n,b,i)` performs line 101, then `superblock` contains ith Dequeue in `b`th block of node `n`.*

*Proof.*

1. Value read for `super[b.time]` in line 101 is not null.

   ▶ Values $\texttt{c}_{\texttt{dir}}$ read in lines 23, `super` are set before incrementing in lines 26,27.

2. `super[]` preserves order from child to parent; if in a child block `b` is before `c` then `b.time` $\leq$ `c.time` and `super[b.time]` $\leq$ `super[c.time]`

   ▶ Follows from the order of lines 60, 48, 49.

3. `super[i+1]-super[i]`$\leq p$

   ▶ In a Refresh with successful CAS in line 46, `super` and `counter` are set for each child in lines 48,49. Assume the current value of the counter in node `n` is `i+1` and still `super[i+1]` is not set. If an instance of successful `Refresh(n)` finishes `super[i+1]` is set a new value and a block is added after `n.parent[sup[i]]`. There could be at most $p$ successful unfinished concurrent instances of `Refresh()` that have not reached line 49. So the distance between `super[i+1]` and `super[i]` is less than $p$.

4. Superblock of `b` is within range $\pm 2p$ of the `super[b.time]`.

   ▶ `super[i]` is the index of the superblock of a block containing block b, followed by Lemma 27. It is trivial to see that `n.super` and `n.b.counter` are increasing. `super(b)` is the real superblock of b. `super(t]` is the index of the superblock of the last block with time `t`. If `b.time` is `t` we have:

$$super[t]-p \leq super[t-1] \leq super(t-1] \leq super(b) \leq super(t+1) \leq super(t+1] \leq super[t]+p$$

□

mputeHead **Lemma 25** (Computing Queue's Head). *Let $Q$ be state of the queue if the operations before ith Dequeue in $L(root)$ are applied on the Queue sequentially and $X$ be the head of $Q$. If $Q$ is empty `ComputeHead(i,b)` returns -1, otherwise returns index in $E(root,b)$ of $X$.*

head **Lemma 26** (Validity of `head`). *No two blocks are written in the same index in `n.blocks`.*

*Proof.* `head` is incremented in lines 51, 54 after trying to append a block to the index of the last `head` read. If it was successful, we have to do this, but if it was unsuccessful, it means it has appended to the index before, so we have to update the `head`. If a process dies before line 51, another process will increment `head` in line 54. □

erCounter **Lemma 27** (Validity of `super` and `counter`)**.** *If `super[i]` $\neq$ `null`, then `super[i]` in node `n` is the superblock of a block with `time=i`.*

*Proof.* After a successful CAS in line 46 `super` and `counter` are modified in both children. `super[i]` is supposed to be the superblock of a block with `time`=i and `counter` is the timer in each node. `super[i]` and `counter=i` are expected to update after a bunch of blocks with `time=i` have been aggregated together into a block in the parent. If the process dies before line 48 these values remain unchanged and incoming blocks will get the same `time`. We claim that our algorithm still works since at most $p$ processes die and it will not change our complexity. If a process dies right after line 48, then `counter` will remain the same and `super[i]` is correct. Furthermore we are sure that when `super[i]` is read it will not be `null`. □

search **Lemma 28** (Search Ranges)**.** *Preconditions of all invocation of `BSearch` are satisfied.*

*Proof.* Line 83: `Get(i)` is called if the result of a dequeue is not null. The search is among all blocks in the root.

Line 88: This search tries to find the ith enqueue, knowing that it is in the left child. Search is done over the left subblocks. The start and end of the range are followed by definition. Line 92 is the same.

Line 101: Here, the goal is to find the superblock. We know the distance between answer and the `super[i]` is at most $p$, since at most $p$ processes could die.

□

**Overall** The main difference between the main algorithm and the block tree is that we separate enqueues and dequeues, compute the number of non-null dequeues and the Queue's size in each block in the root. We emphasize that these changes work correctly; every other claim in Block trees applies here.

## 6 Make first level search fast

## References


andraJT98 [1] T. D. Chandra, P. Jayanti, and K. Tan. A polylog time wait-free construction for closed objects. In B. A. Coan and Y. Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 287–296. ACM, 1998.

nstamST10 [2] A. Gidenstam, H. Sundell, and P. Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In C. Lu, T. Masuzawa, and M. Mosbah, editors, *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, volume 6490 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2010.

ffmanSS07 [3] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In E. Tovar, P. Tsigas, and H. Fouchal, editors, *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2007.

ayanti98a [4] P. Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In B. A. Coan and Y. Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 201–210. ACM, 1998.

algQx

**Algorithm** Modified Queue

```
▶ PRBTree[RBTNode] rootBlocks
A persistant redblack tree supporting append(n),get(i),split(j).
append(n) returns true in case successful.
▶ RBTNode
  • RootBlock block
  • int age
    number of finished operations in the block
  • int step
    If this node is the root of the RBTree, then step shows the number of
    appended nodes to the RBTree
▶ leaf extends Node
  • int[] response
    leaf.response[i] stores response of leaf.ops[i]
  • int maxOld
    Index of the youngest old block in the root that this process has seen
    yet.

boolean Refresh(node n)                                  ▷ if n is root
    new=CreateBlock()                                    ▷ TODO
    if new.num==0 then return true
    else if RBTAppend(new) then
..
    end if
end Refresh

<int, int> Index(node n, int b, int i)     ▷ Returns the order in the
root among dequeus, of ith dequeue in bth block of node n
    if n is root then return search in the current RBTRee
    else..
    end if
end Index

Object Dequeue()
    ..
    N_i= RBTnode n that n.block contains the invocation of Dequeue
    N_r= RBTnode n that n.block contains the response of Dequeue
    N_i.age= N_i.age+1                            ▷ this is a shared counter
    N_r.age= N_r.age+1
    if N_i or N_r become old then update maxOld
    end if
end Dequeue

void RBTAppend(block b)              ▷ adds block b to the rootBlocks
    root= rootBlocks.root
    if root.step%p^2==0 then
        Help()
        CollectGarbage()
    end if
    new= RBTnode(b,0,null)
    return rootBlocks.append(new)
end RBTAppend

void Help                                        ▷ Helps pending operations
    last= l.head-1
    for leaf l in leaves do
        if l.blocks[last] is not null then
            if op is DEQ then
                run Dequeue() for l.ops[last] after Propagate()
                write the response to l.responses[last]
            end if
        end if
    end for
end Help

void CollectGrabage                          ▷ Collects the old root blocks.
    l=FindYoungestOld(Root.Blocks.root)
    t1,t2= RBT.split(l)
    RBTRoot.CAS(t2.root)
end CollectGrabage

Block FindYoungestOld(b)
    return    read all maxOld values among leaves and decide the
largest one                     ▷ There is no need to do a sasDK;Lnapgshot.
end findYoungestOld

response FallBack(op i)
    if operation i in leaf l cannot find its desired RootBlock then
        return l.response[i]
    end if
end FallBack
```

ayantiP05 [5] P. Jayanti and S. Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In R. Ramanujam and S. Sen, editors, *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, volume 3821 of *Lecture Notes in Computer Science*, pages 408–419. Springer, 2005.

/KoganP11 [6] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In C. Cascaval and P. Yew, editors, *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 223–234. ACM, 2011.

-MozesS08 [7] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. *Distributed Comput.*, 20(5):323–341, 2008.

ichaelS96 [8] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In J. E. Burns and Y. Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996.

manKLLP18 [9] G. Milman, A. Kogan, Y. Lev, V. Luchangco, and E. Petrank. BQ: A lock-free queue with batching. In C. Scheideler and J. T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 99–109. ACM, 2018.

MoirNSS05 [10] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In P. B. Gibbons and P. G. Spirakis, editors, *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 253–262. ACM, 2005.