Abstract

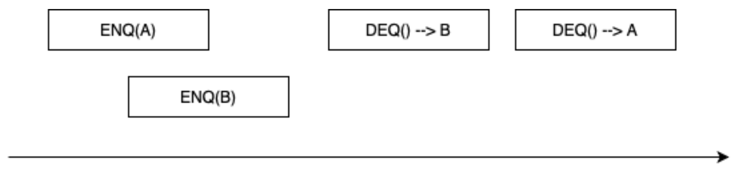
In this work, we are going to introduce a novel lock-free queue implementation. Linearizability and lock-freedom are standard requirements for designing shared data structures. All existing linearizable, lock-free queues in the literature have a common problem in their worst case called CAS Retry Problem. Our contribution is solving this problem while outperforming the previous algorithms.

1 Introduction

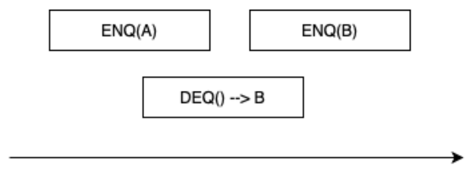
Shared data structures have become an essential field in distributed algorithms research. We are reaching the physical limits of how many transistors we can place on a CPU core. The industry solution to provide more computational power is to increase the number of cores of the CPU. This is why distributed algorithms have become important. It is not hard to see why multiple processes cannot update sequential data structures designed for one process. For example, consider two processes trying to insert some values into a sequential linked list simultaneously. Processes p, q read the same tail node, p changes the next pointer of the tail node to its new node and after that q does the same. In this run, p's update is overwritten. One solution is to use locks; whenever a process wants to do an update or query on a data structure, the process locks it, and others cannot use it until the lock is released. Using locks has some disadvantages; for example, one process might be slow, and holding a lock for a long time prevents other processes from progressing. Moreover, locks do not allow complete parallelism since only the one process holding the lock can make progress.

The question that may arise is, "What properties matter for a lock-free data structure?", since executions on a shared data structure are different from sequential ones, the correctness conditions also differ. To prove a concurrent object works perfectly, we have to show it satisfies safety and progress conditions. A *safety condition* tells us that the data structure does not return wrong responses, and a *progress property* requires that operations eventually terminate.

The standard safety condition is called *linearizability*, which ensures that for any concurrent execution on a linearizable object, each operation should appear to take effect instantaneously at some moment between its invocation and response. Figure fig::goodexample 1 is an example of an execution on a linearizable queue that is initially empty. The arrow shows time, and each rectangle shows the time between the invocation and the termination of an operation. Since `Enqueue(A)` and `Enqueue(B)` are concurrent, `Enqueue(B)` may or may not take effect before `Enqueue(A)`. The execution in Figure fig::badexample 2 is not linearizable since `A` has been enqueued before `B`, so it has to be dequeued first.



odexample Figure 1: An example of a linearizable execution. Either `Enqueue(A)` or `Enqueue(B)` could take effect first since they are concurrent.



adexample Figure 2: An example of an execution that is not linearizable. Since `Enqueue(A)` has completed before `Enqueue(B)` is invoked the `Dequeue()` should return `A` or nothing.

There are various progress properties; the strongest is wait-freedom, and the more common is lock-freedom. An algorithm is *wait-free* if each operation terminates after a finite number of its own steps. We call an algorithm *lock-free* if, after a sufficient number of steps, one operation terminates. A wait-free algorithm is also lock-free but not vice versa; in an infinite run of a lock-free algorithm there might be an operation that takes infinitely many steps but never terminates.

In section 2 we talk about previous queues and their common problems. We also talk about polylogarithmic construction of shared objects.

Jayanti DBLP:conf/podc/Jayanti98a [?] 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 [?] 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 [?]. In section 3 we introduce a polylogarithmic step wait-free universal construction. Our main ideas in of the universal construction also appear in our Queue Algorithm (algQ ??). The main short come of our universal construction is using big CAS objects. We use the universal construction as a stepping stone towards our queue algorithm, so we will not explain it in too much detail.

In section 4 we introduce a concurrent wait-free datastructure, to agree on the order of the operations invoked on some processes.

In section 5 we introduce our main work, the queue; prove its linearizability and wait-freeness.

2 Related Work

2.1 List-based Queues

In the following paragraphs, we look at previous lock-free queues. Michael and Scott DBLP:conf/podc/MichaelS96 [?] introduced a lock-free queue which we refer to as the MS-queue. A version of it is included in the standard Java Concurrency Package. Their idea is to store the queue elements in a singly-linked list (see Figure fig::msq 3). Head points to the first node in the linked list that has not been dequeued, and Tail points to the last element in the queue. To insert a node into the linked list, they use atomic primitive operations like `LL/SC` or `CAS`. If p processes try to enqueue simultaneously, only one can succeed, and the others have to retry. This makes the amortized number of steps to be $\Omega(p)$ per enqueue. Similarly, dequeue can take $\Omega(p)$ steps.

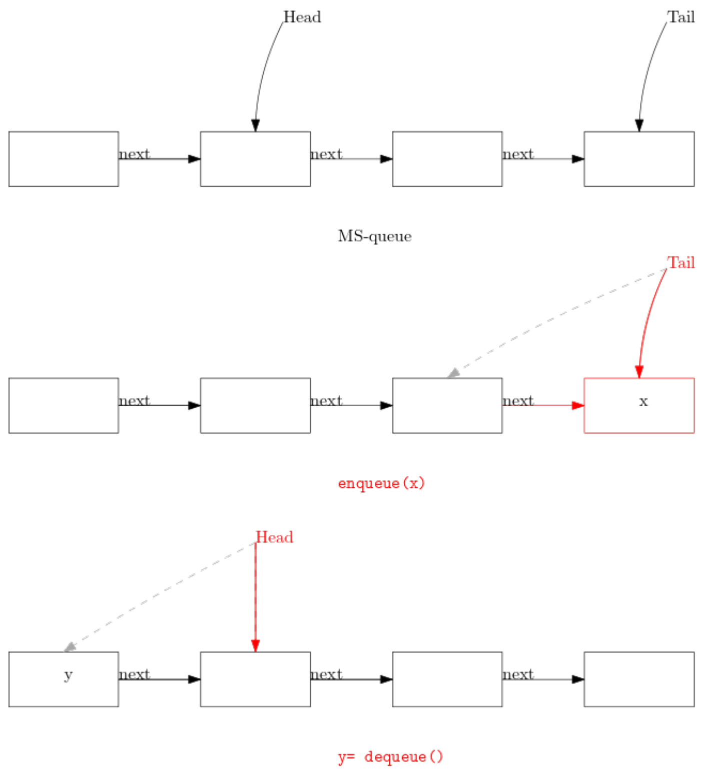


fig::msq Figure 3: MS-queue structure, enqueue and dequeue operations. In the first diagram the first element has been dequeued. Red arrows show new pointers and gray dashed arrows show the old pointers.

Moir, Nussbaum, and Shalev DBLP:conf/spaa/MoirNSS05 [?] presented a more sophisticated queue by using the elimination technique. The elimination mechanism has the dual purpose of allowing operations to complete in parallel and reducing contention for the queue. An Elimination Queue consists of an MS-queue augmented with an elimination array. Elimination works by allowing opposing pairs of concurrent operations such as an enqueue and a dequeue to exchange values when the queue is empty or when concurrent operations can be linearized to empty the queue. Their algorithm makes it possible for long-running operations to eliminate an opposing operation. The empirical evaluation showed the throughput of their work is better than the MS-queue, but the worst case is still the same; in case there are p concurrent enqueues, their algorithm is not better than MS-queue.

Hoffman, Shalev, and Shavit DBLP:conf/opodis/HoffmanSS07 [?] tried to make the MS-queue more parallel by introducing the Baskets Queue. Their idea is to allow more parallelism by treating the simultaneous enqueue operations as a basket. Each basket has a time interval in which all its nodes' enqueue operations overlap. Since the operations in a basket are concurrent, we can order them in any way. Enqueues in a basket try to find their order in the basket one by one by using `CAS` operations. However, like the previous algorithms, if there are still p concurrent enqueue operations in a basket, the amortized step complexity remains $\Omega(p)$ per operation.

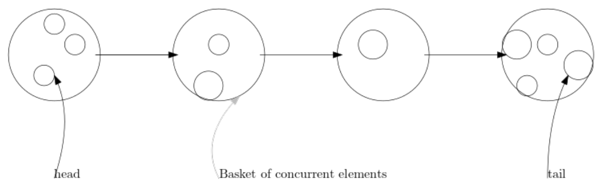
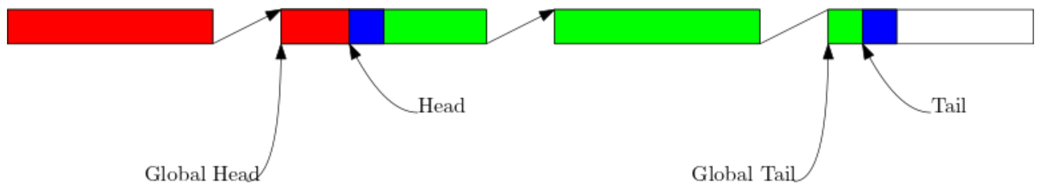


Figure 4: Baskets queue idea. There is a time that all operations in a basket were running concurrently, but only one has succeeded to do `CAS`. To order the operations in a basket, the mechanism in the algorithm for processes is to `CAS` again. The successful process will be the next one in the basket and so on.

Ladan-Mozes and Shavit DBLP:journals/dc/Ladan-MozesS08 [?] presented an Optimistic Approach to Lock-Free FIFO Queues. They use a doubly-linked list and do fewer `CAS` operations than MS-queue. But as before, the worst case is when there are p concurrent enqueues which have to be enqueued one by one. The amortized worst-case complexity is still $\Omega(p)$ `CAS`es.

Hendler et al. DBLP:conf/spaa/HendlerIST10 [?] proposed a new paradigm called flat combining. Their queue is linearizable but not lock-free. Their main idea is that with knowledge of all the history of operations, it might be possible to answer queries faster than doing them one by one. In our work we also maintain the whole history. They present experiments that show their algorithm performs well in some situations.

Gidenstam, Sundell, and Tsigas DBLP:conf/opodis/GidenstamST10 [?] introduced a new algorithm using a linked list of arrays. Global head and tail pointers point to arrays containing the first and last elements in the queue. Global pointers are up to date, but head and tail pointers may be behind in time. An enqueue or a dequeue searches in the head array or tail array to find the first unmarked element or last written element (see Figure fig::sundell 5). Their data structure is lock-free. Still, if the head array is empty and p processes try to enqueue simultaneously, the step complexity remains $\Omega(p)$.



::sundell Figure 5: Global pointers point to arrays. Head and Tail elements are blue, dequeued elements are red and current elements of the queue are green.

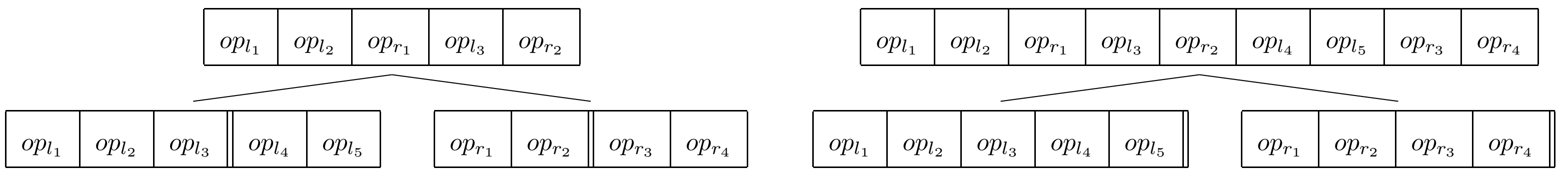
Kogan and Petrank DBLP:conf/ppopp/KoganP11 [?] introduced wait-free queues based on the MS-queue and use Herlihy's helping technique to achieve wait-freedom. Their step complexity is $\Omega(p)$ because of the helping mechanism.

In the worst-case step complexity of all the list-based queues discussed above, there is a p term that comes from the case all p processes try to do an enqueue simultaneously. Morrison and Afek call this the *CAS retry problem* DBLP:conf/ppopp/MorrisonA13 [?]. It is not limited to list-based queues and array-based queues share the CAS retry problem as well DBLP:conf/spaa/TsigasZ01,DBLP:conf/icdcn/Shafiei09,DBLP:conf/iceccs/ColvinG05 [?, ?, ?] . We are focusing on seeing if we can implement a queue in sublinear steps in terms of p or not.

2.2 Universal Constructions

Herlihy discussed the possibility of implementing shared objects from other objects 10.1145/114005.102808 [?]. A *universal construction* is an algorithm that can implement a shared version of any given sequential object. We can implement a concurrent queue using a universal construction. Jayanti proved an $\Omega(\log p)$ lower bound on the worst-case shared-access time complexity of p-process universal constructions DBLP:conf/podc/Jayanti98a [?]. He also introduced a construction that achieves $\mathrm{O}(\log^2 p)$ shared accesses DBLP:conf/podc/ChandraJT98 [?]. His universal construction can be used to create any data structure, but its implementation is not practical because of using unreasonably large-sized `CAS` operations.

Ellen and Woelfel introduced an implementation of a Fetch&Inc object with step complexity of $O(\log p)$ using $O(\log n)$-bit `LL/SC` objects, where n is the number of operations 10.1007/978-3-642-41527-2_20 [?]. Their idea has similarities to Jayanti's construction, and they represent the value of the Fetch&Inc using the history of successful operations.



op_{l_1} op_{l_2} op_{r_1} op_{l_3} op_{r_2}
op_{l_1} op_{l_2} op_{l_3} op_{l_4} op_{l_5}
op_{r_1} op_{r_2} op_{r_3} op_{r_4}
op_{l_1} op_{l_2} op_{r_1} op_{l_3} op_{r_2} op_{l_4} op_{l_5} op_{r_3} op_{r_4}

(a) Operations after || are new.

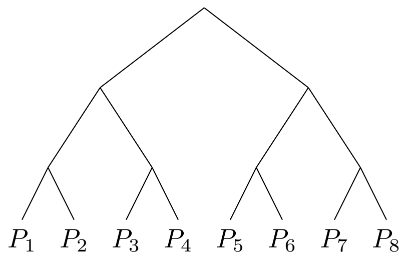
(b) New operations are added to the parent node.

ucexample

Figure 6: Propagate Step in Universal Construction

3 Universal Construction using Tournament Tree with Big CAS Objects

A universal construction gets any sequential object and creates a concurrent version of the given datastructure. Our universal construction in Algorithm alg1 relies on a tournament tree with p leaves and height $\log(p)$ is shared among p processes. Nodes in the tree are CAS objects that store an ordering and support `append()` and `diff()` operations. Leaf l_i stores the sequence of the operations invoked by P_i. Each internal node stores the sequence of operations propagated up to it. 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 history of operations stored at the root is the linearization ordering. The operation *op* is linearized when it is appended to the root.



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 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 `Propagate()` 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) terminates.

alg1

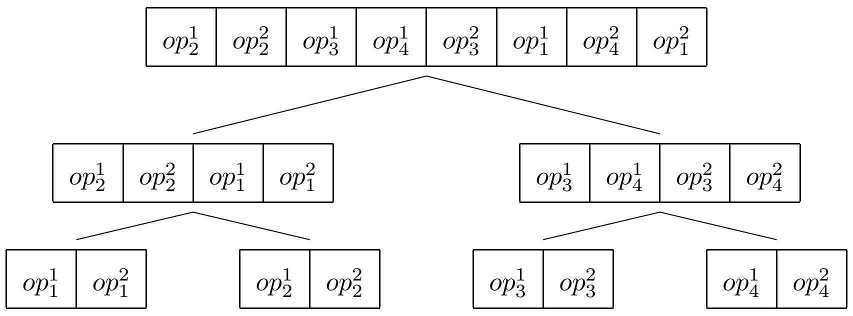
Algorithm Universal Construction Idea

```
 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 fig::uc 7). 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 sized CAS objects and still achieve polygarithmic step complexity.



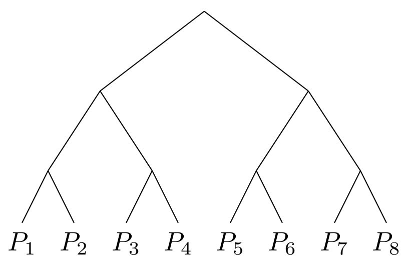
op_2^1 op_2^2 op_3^1 op_4^1 op_3^2 op_1^1 op_4^2 op_1^2
op_2^1 op_2^2 op_1^1 op_1^2
op_3^1 op_4^1 op_3^2 op_4^2
op_1^1 op_1^2
op_2^1 op_2^2
op_3^1 op_3^2
op_4^1 op_4^2

fig::uc

Figure 7: Universal Construction: op_j^i denotes the ith operation from process j. In each node, we store the ordering of all the operations propagated up to it.

4 Block Trees

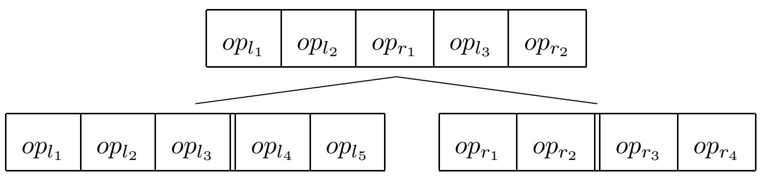
Here we introduce a data structure that allows processes to agree on the linearization ordering of their operations using O($\log p$) `CAS` per operation called a *block tree*. Then we use the block tree as a stepping stone towards our queue algorithm. A block tree is a tournament tree shared among p processes (see Figure fig::blocktree 8). Each process has a leaf, and it appends its operations to its leaf. After that, the process tries to propagate its new operation up to the tree's root. An ordering of operations propagated up to a node is stored in that node. All processes agree on the sequence stored in the root and this is used as the linearization ordering. Our idea is similar to Jayanti and Petrovic's multi-enqueuer single-dequeuer Queue DBLP:conf/fsttcs/JayantiP05 [?], but we do not use `CAS` operations with big words and do not put a limit on the number of concurrent operations.



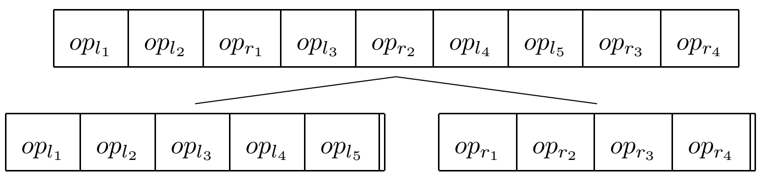
blocktree Figure 8: In the block tree each process has a leaf and in each node there is an ordering of operations stored. Each node tries to propagate its operations up to the root, which stores the final ordering of all operations.

The goal here is to ensure that in each propagate step the new operations are propagated up to the parent in $O(\log p)$ steps (see Figure fig::propagstep 9). Then, a dequeue operation uses the linearization ordering to compute its answer.

In each propagate step, our algorithm uses a subroutine Refresh(n) that aggregates new operations from node n's children (that have not already been propagated to n) and tries to append them into n using a `CAS` operation. The general idea is that if we call Refresh(n) twice, the operations in n's children before the first Refresh(n) are guaranteed to be in n. Instead of storing operations explicitly in the nodes, we only keep track of the number of them. This allows us to `CAS` fixed-size objects in each Refresh(n). To do that, we introduce blocks that only contain the number of operations from the left and the right child in a `Refresh`() procedure and only propagate the block of the new operations.


op_{l_1} op_{l_2} op_{r_1} op_{l_3} op_{r_2}
op_{l_1} op_{l_2} op_{l_3} op_{l_4} op_{l_5}
op_{r_1} op_{r_2} op_{r_3} op_{r_4}

(a) Operations after || are new.


op_{l_1} op_{l_2} op_{r_1} op_{l_3} op_{r_2} op_{l_4} op_{l_5} op_{r_3} op_{r_4}
op_{l_1} op_{l_2} op_{l_3} op_{l_4} op_{l_5}
op_{r_1} op_{r_2} op_{r_3} op_{r_4}

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

ropagstep

Figure 9: Successful `Refresh`, operations in children after || are new.

We also implement methods `Get(i)`, `Index(op)` to get the ith propagated operation and compute the rank of a propagated operation in the linearization. `Get(i)` finds the block containing the ith operation in the root and then finds its sub-block recursively to reach a leaf. `Index()` is similar but more complicated, finding super-blocks from a leaf to the root. The main challenge in each level of `Get(i)` and `Index(op)` is that it should take polylogarithmic steps with respect to p. After appending operation `op` to the root, processes can find out information about the linearization ordering using `Get(i)` and `Index(op)`.

`Get()` and `Index()` search among blocks in each level of the tree to find the sub-block or super-block containing the given operation. Each block stores a constant amount of information (like prefix sums) to allow binary searches to find the required block in a node quickly.

Block tree can be used to implement queue, but `Get(i)` may take a long time since it has to find the block containing the ith operation at the root level.

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.

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 10).

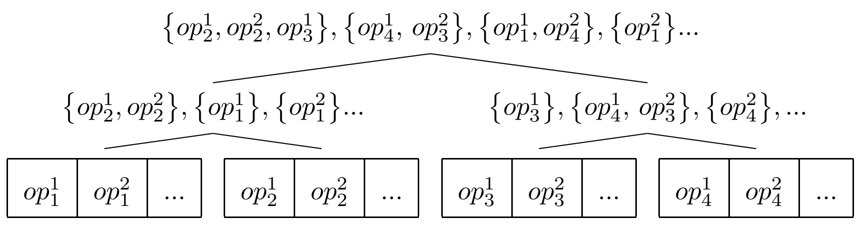

$\{op_2^1, op_2^2, op_3^1\}, \{op_4^1, op_3^2\}, \{op_1^1, op_4^2\}, \{op_1^2\}...$
$\{op_2^1, op_2^2\}, \{op_1^1\}, \{op_1^2\}...$
$\{op_3^1\}, \{op_4^1, op_3^2\}, \{op_4^2\}, ...$
op_1^1 op_1^2 ...
op_2^1 op_2^2 ...
op_3^1 op_3^2 ...
op_4^1 op_4^2 ...

fig::set

Figure 10: 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.

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 ??.

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 11.)

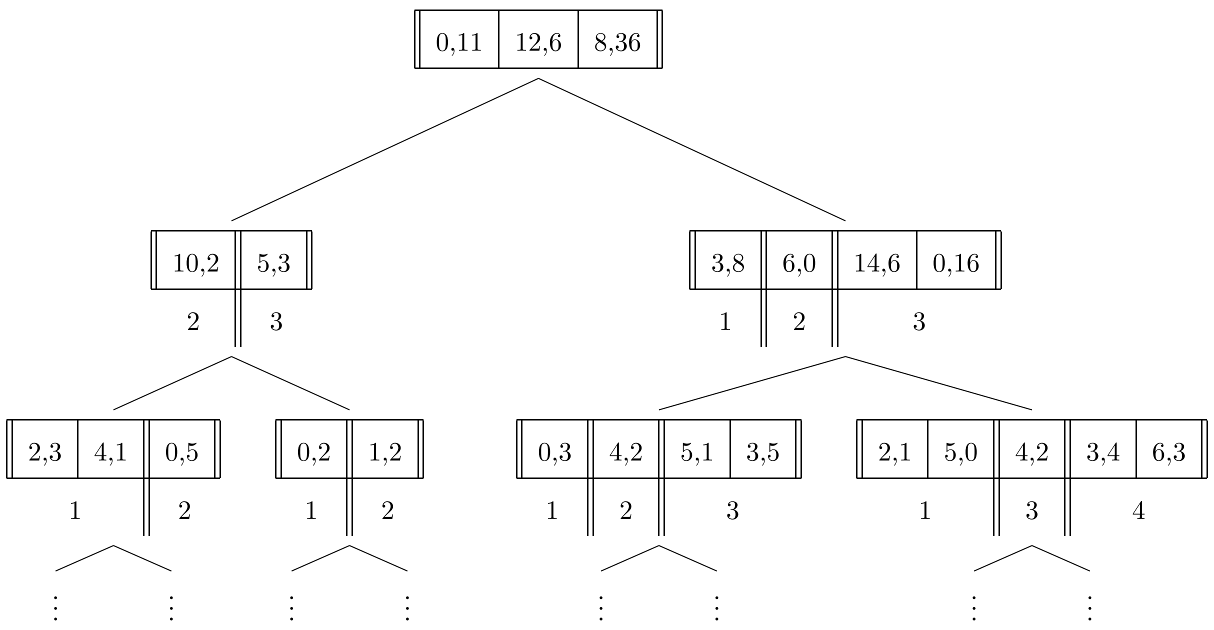


fig:block Figure 11: Showing concurrent operation sets with blocks. Each block consists of a pair(left, right) indicating the number of operations from the left and the right child, respectively. Block (12,6) in the root contains blocks (10,2) from the left child and (6,0) from the right child. Blocks 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.

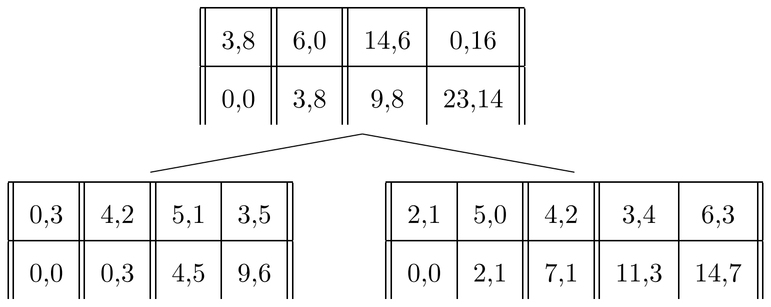
Each block b in node n is the aggregation of blocks in the children of n that are newly read by thePropagate() step that created block b. For example, the third block in the root (8,36) is created by merging block (5,3) from the left child and (14,6) and (0,16) from the right child. Block (5,3) also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

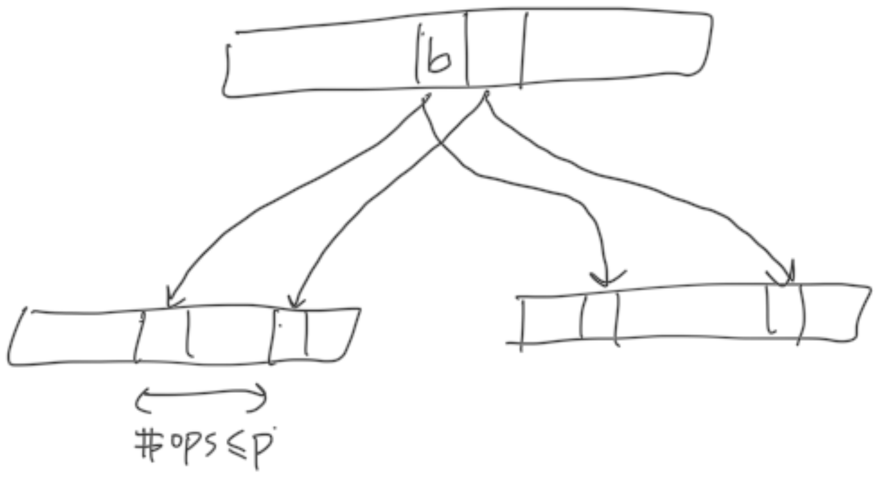


ig:prefix Figure 12: Using Prefix sums in blocks. When we want to find block b elements in its children, we can use binary search. The number below each block shows the count of elements in the previous blocks.

4.1 Using pointers and prefix sum to make GetIndex(i) faster

GetIndex(i) returns the ith operation stored in the block tree sequence. We do that by finding the block b_i containing ith element in the root, and then recursively finding the subblock of b_i which contains ith 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 13 and fig:prefix 12).



::pointer Figure 13: 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 ith operation, then continues recursively calling GetElement(b, i) to find ith 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).

4.2 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 ith 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 ith element in the sequence can be determined in $O(\log^2 p)$ steps by recursively finding ith 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 12]. 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.

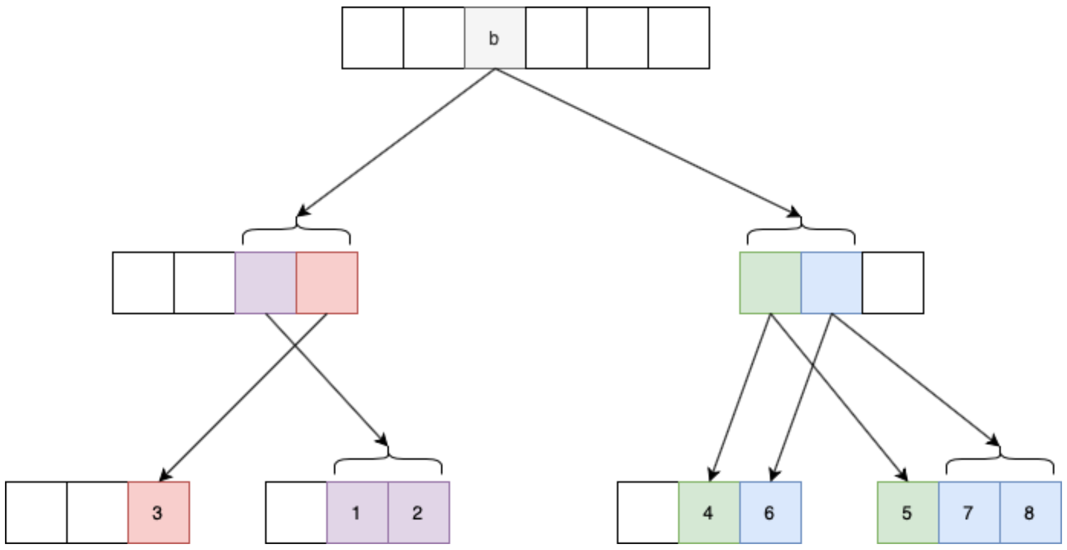
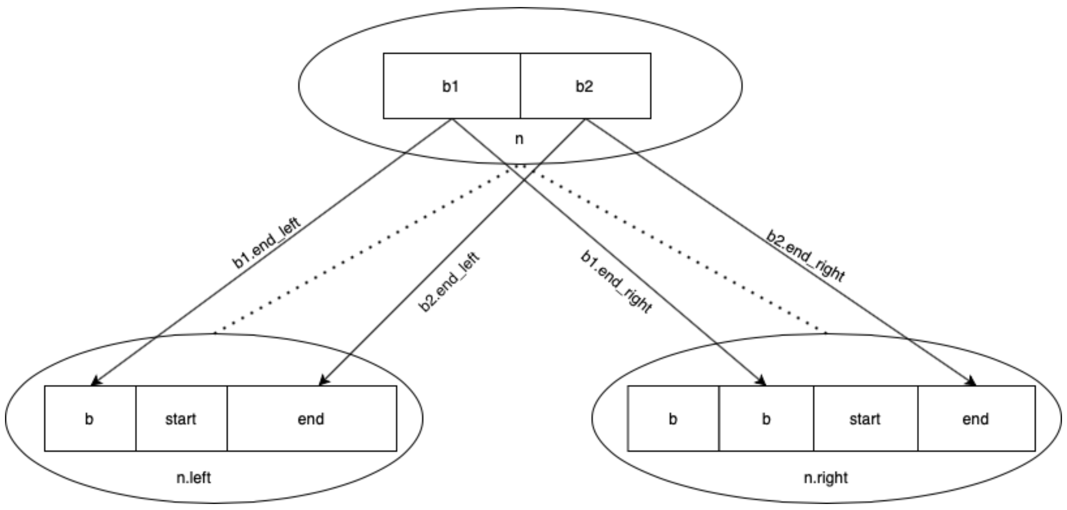


Figure 14: Order of elements in b: elements in leaves are ordered with numerical order in the drawing.

`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 15: 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. ☐

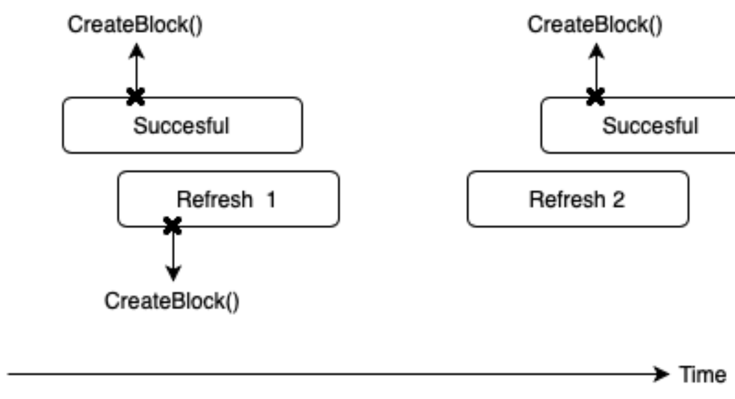


Figure 16: The second failed Refresh is assuredly concurrent to a Successful `Refresh()` with `CreateBlock` line after first failed `Refresh`'s `CreateBlock()`.

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

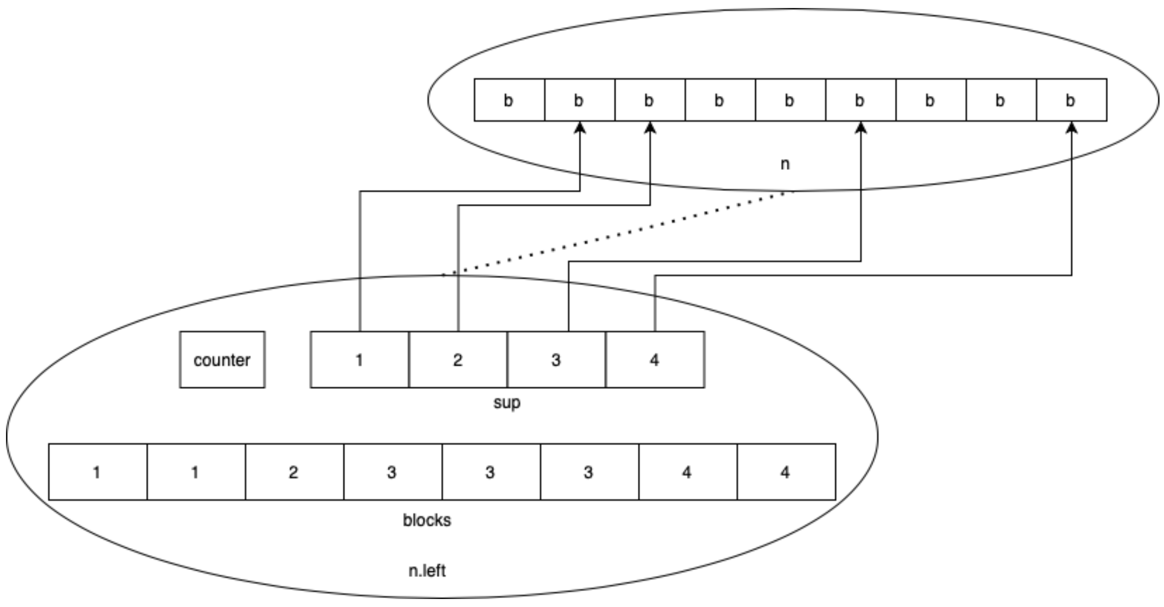


Figure 17: `Sup` and `timer` in a node, numbers on blocks are their `time` values.

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 $\texttt{c}_{\texttt{dir}}$ 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:

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

□

5 Implementing Queue using Block Tree

In this work, we design a queue with $O(\log^2 p + \log n)$ steps per operation, where n is the number of total operations invoked. We avoid the $\Omega(p)$ worst-case step complexity of existing shared queues based on linked lists or arrays (CAS Retry Problem). 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? We can implement Enqueue and Dequeue using our block tree. An `Enqueue(e)` appends an operation with input argument `e` in the block tree. To do a `Dequeue()`, process p first appends a `DEQ` operation to the tree. Then p finds the rank of the `DEQ` using `Index()`, the rank of the `DEQ` and the information stored in the root about the queue p computes the rank of the `ENQ` having the answer of the `DEQ`. Finally p returns the argument of that `ENQ` using `Get(i)`.

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

Table 1: 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 ith `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 2: 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 3: 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:

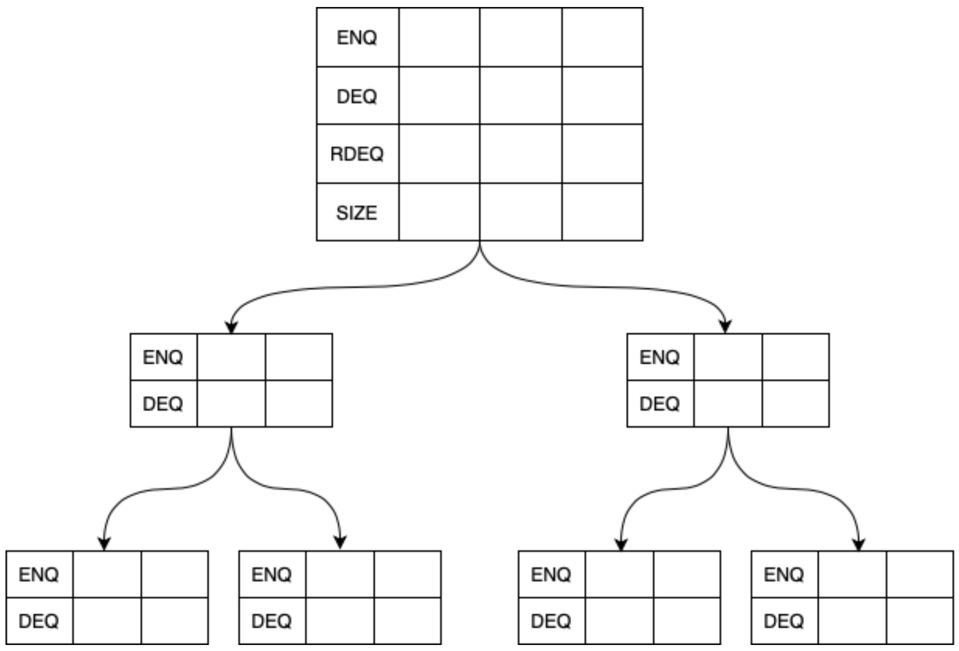
```
size[b]= max(size[b-1] +enqueues[b] -dequeues[b], 0)
non-null dequeues[b]= non-null dequeues[b-1] +dequeues[b] -size[b-1] -enqueues[b]
```

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;
```

6 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.



ig::queue

Figure 18: Fields stored in the Queue nodes.

6.1 Pseudocode description

Tournament Tree In order to reach an agreement on the order of operations among p processes, we use a Tournament Tree. Leaf `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 (def::subblock Subblock)**.** Block `b` is a subblock of `n.blocks[i]` if it is in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` or `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`.

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 fig::createBlock 15 .

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

We store ordering among `operations` in the tournament tree constructed by `nodes`. In each `node` we store pointers to its relatives, an array of `blocks` 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 `blocks` 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 pn'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.

7 Pseudocode

Algorithm Tree Fields Description

```
◇ Shared
    • A binary tree of Nodes with one leaf for each process. root is the root node.
◇ Local
    • Node leaf: process's leaf in the tree.
◇ Structures
▶ Node
    • *Node left, right, parent : initialized when creating the tree.
    • BlockList
    • int head= 1: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.
    • int $num_{propagated}$= 0 : # groups of blocks that have been propagated from the node to its parent.
▶ Block
    • int group : the value read from $num_{propagated}$ when appending this block to the node.
▶ LeafBlock extends Block
    • Object element : Each block in a leaf represents a single operation. If the operation is enqueue(x) then element=x, otherwise element=null.
    • int $sum_{enq}$, $sum_{deq}$ : # enqueue, dequeue operations in the prefix for the block
▶ InternalBlock extends Block
    • int $end_{left}$, $end_{right}$ : indices of the last subblock of the block in the left and right child
    • int $sum_{enq\text{-}left}$ : # enqueue operations in the prefix for left.blocks[$end_{left}$]
    • int $sum_{deq\text{-}left}$ : # dequeue operations in the prefix for left.blocks[$end_{left}$]
    • int $sum_{enq\text{-}right}$ : # enqueue operations in the prefix for right.blocks[$end_{right}$]
    • int $sum_{deq\text{-}right}$ : # dequeue operations in the prefix for right.blocks[$end_{right}$]
▶ RootBlock extends InternalBlock
    • int size : size of the queue after performing all operations in the prefix for this block
```

Abbreviations:

- blocks[b].sum_x=blocks[b].$sum_{x\text{-}left}$+blocks[b].$sum_{x\text{-}right}$ (for $b \geq 0$ and $x \in \{enq, deq\}$)
- blocks[b].sum=blocks[b].sum_{enq}+blocks[b].sum_{deq} (for $b \geq 0$)
- blocks[b].num_x=blocks[b].sum_x-blocks[b-1].sum_x
  (for $b>0$ and $x \in \{\emptyset$, enq, deq, enq-left, enq-right, deq-left, deq-right$\}$)

Algorithm *Queue*

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

208: Object Dequeue()     ▷ Creates a block with null value element, appends it to the tree, computes its order among operations, and returns its response.
209:     block newBlock= new(LeafBlock)
210:     newBlock.element= null
211:     newBlock.sum_enq= leaf.blocks[leaf.head].sum_enq
212:     newBlock.sum_deq= leaf.blocks[leaf.head].sum_deq+1
213:     leaf.Append(newBlock)
214:     <b, i>= IndexDeq(leaf.head, 1)
215:     output= FindResponse(b, i)                                  deqRest
216:     return output
217: end Dequeue

218: <int, int> FindResponse(int b, int i)
                                         ▷ Returns the the response to the $D_{root,b,i}$.
219:     if root.blocks[b-1].size  + root.blocks[b].num_enq - i < 0 then
220:         return null                   ▷ Check if the queue is empty.    checkEmpty
221:     else
222:         e= i - root.blocks[b-1].size + root.blocks[b-1].sum_enq     computeE
                                         ▷ $E_e(root)$ is the response.
223:         return root.GetENQ(root.DSearch(e, b))                  findAnswer
224:     end if
225: end FindResponse
```

Algorithm *Node*

```
301: void PROPAGATE()
firstRefresh 302:     if not REFRESH() then
secondRefresh 303:         REFRESH()
304:     end if
305:     if this is not root then
306:         parent.PROPAGATE()
307:     end if
308: end PROPAGATE

309: boolean REFRESH()
readHead 310:     h= head
keCreateBlock 311:     <new, np$_{left}$, np$_{right}$>= CREATEBLOCK(h)    ▷ np$_{left}$, np$_{right}$ are the
         values read from the children's num$_{propagated}$ field.
addOP 312:     if new.num==0 then return true    ▷ The block contains nothing.
cas 313:     else if blocks.tryAppend(new, h) then
okcas 314:         for each dir in {left, right} do
setSuper 315:             CAS(dir.super[np$_{dir}$], null, h)    ▷ Write would work too.
incNP 316:             CAS(dir.num$_{propagated}$, np$_{dir}$, np$_{dir}$+1)
317:         end for
ncrementHead1 318:         CAS(head, h, h+1)
319:         return true
320:     else
321:         CAS(head, h, h+1)    ▷ Even if another process wins, help
         to increase the head. The winner might have fallen sleep before increasing
ncrementHead2    head.
322:         return false
323:     end if
324: end REFRESH

⇝ Precondition: blocks[start..end] contains a block with field f ≥ i
325: int BSEARCH(field f, int i, int start, int end)
                                   ▷ Does binary search for the value
         i of the given prefix sum field. Returns the index of the leftmost block in
         blocks[start..end] whose field f is ≥ i.
326: end BSEARCH

327: <Block, int, int> CREATEBLOCK(int i)    ▷ Creates a block
         to be inserted as ith block in blocks. Returns the created block as well as
         values read from each child's num$_{propagated}$ field. These values are used for
         incrementing the children's num$_{propagated}$ field if the block was appended to
         blocks successfully.
328:     block newBlock= NEW(block)
setGroup 329:     newBlock.group= num$_{propagated}$
330:     for each dir in {left, right} do
lastLine 331:         index$_{last}$= dir.head-1
prevLine 332:         index$_{prev}$= blocks[i-1].end$_{dir}$
endDefLine 333:         newBlock.end$_{dir}$= index$_{last}$
334:         block$_{last}$= dir.blocks[index$_{last}$]
335:         block$_{prev}$= dir.blocks[index$_{prev}$]
336:                 ▷ newBlock includes dir.blocks[index$_{prev}$+1..index$_{last}$].
setNP 337:         np$_{dir}$= dir.num$_{propagated}$
338:         newBlock.sum$_{enq-dir}$= blocks[i-1].sum$_{enq-dir}$ + block$_{last}$.sum$_{enq}$
         - block$_{prev}$.sum$_{enq}$
339:         newBlock.sum$_{deq-dir}$= blocks[i-1].sum$_{deq-dir}$ + block$_{last}$.sum$_{deq}$
         - block$_{prev}$.sum$_{deq}$
340:     end for
341:     if this is root then
342:         newBlock.size = max(root.blocks[i-1].size + newBlock.num$_{enq}$
computeLength    - newBlock.num$_{deq}$, 0)
343:     end if
344:     return <b, np$_{left}$, np$_{right}$>
345: end CREATEBLOCK
```

Algorithm Root

```
⇝ Precondition: root.blocks[end].sum$_{enq}$ ≥ e
801: <int, int> DSEARCH(int e, int end)    ▷ Returns <b,i> if $E_e(root) = E_i(root, b)$.
802:     start= end-1
803:     while root.blocks[start].sum$_{enq}$≥e do
doubling 804:         start= max(start-(end-start), 0)
805:     end while
806:     b= root.BSearch(sum$_{enq}$, e, start, end)
hComputei 807:     i= e- root.blocks[b-1].sum$_{enq}$
808:     return <b,i>
809: end DSEARCH
```

Algorithm Node

```
    ⇝ Precondition: blocks[b].num$_{enq}$$\geq$i$\geq 1$
401: element GetEnq(int b, int i)                                    ▷ Returns the element of $E_i(this, b)$.
402:     if this is leaf then
403:         return blocks[b].element
404:     else if i $\leq$ blocks[b].num$_{enq\text{-}left}$ then                ▷ $E_i(this, b)$ is in the left child of this node.
405:         subBlock= left.BSearch(sum$_{enq}$, i+blocks[b-1].sum$_{enq\text{-}left}$, blocks[b-1].end$_{left}$+1, blocks[b].end$_{left}$)
406:         return left.GetEnq(subBlock, i)
407:     else
408:         i= i-blocks[b].num$_{enq\text{-}left}$
409:         subBlock= right.BSearch(sum$_{enq}$, i+right.blocks[b-1].sum$_{enq\text{-}right}$, blocks[b-1].end$_{right}$+1, blocks[b].end$_{right}$)
410:         return right.GetEnq(subBlock, i)
411:     end if
412: end GetEnq

    ⇝ Precondition: bth block of the node has propagated up to the root and blocks[b].num$_{enq}$$\geq$i.
413: <int, int> IndexDeq(int b, int i)                                ▷ Returns <x, y> if $D_{this,b,i} = D_{root,x,y}$.
414:     if this is root then
415:         return <b, i>
416:     else
417:         dir= (parent.left==n)? left: right                      ▷ check if this node is a left or a right child
418:         superBlock= parent.BSearch(sum$_{deq\text{-}dir}$, i+blocks[b-1].sum$_{deq}$, super[blocks[b].group]-p, super[blocks[b].group]+p)
                                ▷ superblock's group has at most $p$ difference with the value stored in super[].
419:         if dir is left then
420:             i+= blocks[b-1].sum$_{enq}$-blocks[superBlock-1].sum$_{enq\text{-}left}$     ▷ consider the enqueues in the previous blocks from the left child
421:         end if
422:         if dir is right then
423:             i+= blocks[b-1].sum$_{enq}$-blocks[superBlock-1].sum$_{enq\text{-}right}$    ▷ consider the enqueues in the previous blocks from the right child
424:             i+= blocks[superBlock].num$_{deq\text{-}left}$                 ▷ consider the dequeues from the right child
425:         end if
426:         return this.parent.IndexDeq(superBlock, i)
427:     end if
428: end IndexDeq
```

Algorithm Leaf

```
601: void Append(block blk)                                  ▷ Append is only called by the owner of the leaf.
602:     blk.group= head
603:     blocks[head]= blk
604:     head+=1
605:     parent.Propagate()
606: end Append
```

Algorithm BlockList

▷ : Supports two operations blocks.tryAppend(Block b), blocks[i]. Initially empty, when blocks.tryAppend(b, n) returns true b is appended to blocks[n] and blocks[i] returns ith block in the blocks. If some instance of blocks.tryAppend(b, n) returns false there is a concurrent instance of blocks.tryAppend(b′, n) which has returned true.blocks[0] contains an empty block with all fields equal to 0 and end$_{left}$, end$_{right}$ pointers to the first block of the corresponding children.

block[] blocks: array of blocks

int[] super: super[i] stores an approximate index of the superblock of the blocks in blocks whose group field have value i.

```
701: boolean TryAppend(block blk, int n)
702:     return CAS(blocks[n], null, blk)
703: end TryAppend
```

8 Proof of Linearizability

$\boxed{\text{TEST}}$ Fix the logical order of definitions (cyclic refrences).

$\boxed{\text{TEST}}$ Is it better to show `ops(EST`$_{\texttt{n, t}}$`)` with `EST`$_{\texttt{n, t}}$?

$\boxed{\text{Question}}$ A good notation for *the index of the* `b`?

$\boxed{\text{Question}}$ How to remove the notion of time? To say pre(n,i) contains n.blocks[0..i] instead of EST(n,t) which head=i at time t. Is it good? Furthermore, can we remove the notion of established blocks?

Definition 7 (Block)**.** A block is an object storing some statistics, as described in Algorithm Queue. A block in a node's blocklist implicitly represents a set of operations. If `n.blocks[i]==b` we call `i` the *index* of block `b`. Block `b` is before block `b`$'$ in node `n` if and only if the index of the `b` is smaller than the index of the `b`$'$'s. For a `block` in a `BlockList` we define *the prefix for the block* to be the blocks in the `BlockList` up to and including the `block`.

$\boxed{\texttt{::headInc}}$ **Lemma 8** (head Increment)**.** *Let* `R` *be an instance of* `Refresh` *on node* `n` *that reaches Line* $\overset{\texttt{cas}}{313}$*. After* `R` *terminates* `n.head` *is greater than h, the value read in line* $\overset{\texttt{readHead}}{310}$ *of* `R`.

Proof. If Line $\overset{\texttt{incrementHead1}}{318}$ or $\overset{\texttt{incrementHead2}}{321}$ are successful then the claim holds, otherwise another process has incremented the head from `h` to `h+1`. ☐

$\boxed{\texttt{dPosition}}$ **Invariant 9** (headPosition)**.** If the value of `n.head` is `h` then, `n.blocks[i]=null` for `i>h` and `n.blocks[i]`$\neq$`null` for `i<h`.

Proof. The invariant is true initially since 1 is assigned to `n.head` and `n.blocks[x]` is null for every `x`. The truth of the invariant may be affected by writing into `n.blocks` or incrementing `n.head`. We show the invariant still holds after these two changes.

In the algorithm, some value is appended to `n.blocks[]` by writing into `n.blocks[head]` only in Line 313. Writing into `n.blocks[head]` preserves the invariant, since the claim does not talk about `n.blocks[head]`. The value of `n.head` is modified only in lines $\overset{\texttt{incrementHead1}}{318}$ and $\overset{\texttt{incrementHead2}}{321}$. Depending on whether the `TryAppend()` in Line $\overset{\texttt{cas}}{313}$ succeeded or not, we show that the claim holds after the increment of `n.head` in either case. If `n.head` is incremented to h it is sufficient to show `n.blocks[h]`$\neq$`null` to prove the invariant still holds. In the first case the process applied a successful `TryAppend(new,h)` in line $\overset{\texttt{okcas}}{314}$, which means `n.blocks[h]` is not null anymore. Note that whether $\overset{\texttt{incrementHead1}}{318}$ or $\overset{\texttt{incrementHead1}}{318}$ return true or false, after they finish we know that `n.head` has been incremented from the value read in Line $\overset{\texttt{readHead}}{310}$ (Lemma $\overset{\texttt{lem::headInc}}{8}$). The failure case is also the same since it means some non-null value has been written into `n.blocks[head]` by some process. ☐

Explain More

$\boxed{\texttt{dProgress}}$ **Lemma 10** (headProgress)**.** `n.head` *is non-decreasing over time. If* `n.blocks[i]`$\neq$`null` *and* i.0 *then* `n.blocks[i].end`$_{\texttt{left}}$ $\geq$ `n.blocks[i-1].end`$_{\texttt{lef}}$ *and* `n.blocks[i].end`$_{\texttt{right}}$ $\geq$ `n.blocks[i-1].end`$_{\texttt{right}}$.

Proof. The first claim follows trivially from the pseudocode since `n.head` is only incremented in the pseudocode in lines $\overset{\texttt{incrementHead1}}{318}$ and $\overset{\texttt{incrementHead2}}{321}$ of `Refresh()` .

Consider the block `b` written into `n.blocks[i]` by `TryAppend()` at Line $\overset{\texttt{cas}}{313}$. It is created by the `CreateBlock(i)` called at Line $\overset{\texttt{invokeCreateBlock}}{311}$. Prior to this call to `CreateBlock(i)`, `n.head=i` at Line $\overset{\texttt{readHead}}{310}$, so `n.blocks[i-1]` is already a non-null value `b`$'$ by Invariant $\overset{\texttt{lem::headPosition}}{9}$. Thus the `CreateBlock(i-1)` that creates `b`$'$ terminates before `CreateBlock(i)` that creates `b` is invoked. The value written into `b.end`$_{\texttt{left}}$ at Line $\overset{\texttt{endDefLine}}{333}$ of `CreateBlock(i)` was read from `n.left.head-1` at Line $\overset{\texttt{lastLine}}{331}$ of `CreateBlock(i)`. Similarly, the value in `n.blocks[i-1].end`$_{\texttt{left}}$ was read from `n.left.head-1` during the call to `CreateBlock(i-1)`. Since `n.left.head` is non-decreasing `b`$'$`.end`$_{\texttt{left}}$ $\leq$ `b.end`$_{\texttt{left}}$. The proof for `end`$_{\texttt{right}}$ is similar. ☐

$\boxed{\texttt{:subblock}}$ **Definition 11** (Subblock)**.** Block `b` is a *direct subblock* of `n.blocks[i]` if it is in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` $\cup$ `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]` . Block `b` is a subblock of `n.blocks[i]` if `b` is a direct subblock of `n.blocks[i]` or a subblock of a direct subblock of `n.blocks[i]`.

$\boxed{\texttt{append}}$ **Corollary 12** (No Duplicates)**.** *If* `op` *is in* `n.blocks[i]` *then there is no* `j`$\neq$`i` *such that* `op`$\in$`ops(n.blocks[j])`.

Proof. Operation `op` is invoked only one time in an execution because every operations invoked is distinct. Since there is node `n` which `op` is in two different blocks of `n`, there is node `n'` that is the lowest height node in the tree that contains `op` in two of its blocks `b1,b2`. By Definition def::subblock 11, `b1` and `b2` have distinct subblocks(not only direct subblocks) and since `op` is in only one leaf block, then it cannot be in both `b1` and `b2`. ☐

Definition 13 (Superblock)**.** Block `b` is *direct superblock* of block `c` if c is a direct subblock of b. Block `b` is *superblock* of block `c` if `c` is a subblock of `b`.

def::ops **Definition 14** (Operations of a block)**.** A leaf block `b` in a leaf represents `enqueue(x)` if `b.element=x`$\neq$`null`. Else if `b.element=null` b represents a `dequeue()`. The set of operations of block `b` are the operations in the subblocks of `b`. We denote the set of operations of block `b` by `ops(b)`.

We say block `b` is *propagated to node* `n` if `b` is in `n.blocks` or is a subblock of a block in `n.blocks`. We also say `b` contains `op` if `op`$\in$`ops(b)`.

Definition 15. A block `b` in `n.blocks` is *established* at time t if `n.head`$>$ index of `b` at time t. $EST_{n,\ t}$ is the set of established blocks of node `n` at time t.

head **Observation 16.** *Once a block* `b` *is written in* `n.blocks[i]` *then* `n.blocks[i]` *never changes.*

Lemma 17. *Every block has at most one direct superblock.*

Proof. To show this we are going to refer to the way `n.blocks[]` is partitioned while propagating blocks up to `n.parent`. `n.CreateBlock(i)` merges the blocks in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`..n.blocks[i].end`$_{\texttt{left}}$`]` and `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`..n.blocks[i]` (Lines lastLine prevLine 331, 332). Since `end`$_{\texttt{left}}$`,end`$_{\texttt{right}}$ are non-decreasing (`n.blocks[i].end`$_{\texttt{left|right}}$`>n.blocks[i-1].end`$_{\texttt{left|right}}$), so the range of the subblocks of `n.blocks[i]` which is `(n.blocks[i-1].end`$_{\texttt{dir}}$`+1..n.blocks[i].end`$_{\texttt{dir}}$`)` does not overlap with the range of the subblocks of `n.blocks[i-1]`. ☐

shedOrder **Lemma 18** (establishedOrder)**.** *If time $t <$ time t', then* `ops(EST`$_{\texttt{n, t}}$`)`$\subseteq$ `ops(EST`$_{\texttt{n, t'}}$`)`.

Proof. Blocks are only appended (not modified) with CAS to `n.blocks[n.head]` and `n.head` is non-decreasing, so the set of operations in established blocks of a node can only grow. ☐

useless?

▶ Processes are numbered from 1 to p and leaves of the tree are assigned from left to right. We will show in Lemma blockSize ~~28 that t~~here is at most one operation from each process in a given block.

ordering **Definition 19** (Ordering of operations inside the nodes).

- The prefix of an operation op in the sequence of operations S is the sequence of operations strictly before op.
- $E(n,b)$ is the sequence of enqueue operations in `ops(n.blocks[b])` defined recursively as follows. $E(leaf,b)$ is the single enqueue operation in `ops(leaf.blocks[b])` or an empty sequence if `leaf.blocks[b].num`$_{\texttt{enq}}$`=0`. If `n` is an internal node, then

$$\begin{aligned}E(n,b) =&E(n.left, n.blocks[b-1].end_{\text{left}}+1)\cdot E(n.left, n.blocks[b-1].end_{\text{left}}+2)\cdots E(n.left, n.blocks[b].end_{\text{left}})\cdot\\ &E(n.right, n.blocks[b-1].end_{\text{right}}+1)\cdot E(n.right, n.blocks[b-1].end_{\text{right}}+2)\cdots E(n.right, n.blocks[b].end_{\text{right}})\end{aligned}$$

- $E_i(n,b)$ is the ith enqueue in $E(n,b)$.
- The order of the enqueue operations in the node n is $E(n)=E(n,1)\cdot E(n,2)\cdot E(n,3)\cdots$
- $E_i(n)$ is the ith enqueue in $E(n)$.
- $D(n,b)$ is the sequence of dequeue operations in `ops(n.blocks[b])` defined recursively as follows. $D(leaf,b)$ is the single dequeue operation in `ops(leaf.blocks[b])` or an empty sequence if `leaf.blocks[b].num`$_{\texttt{deq}}$`=0`. If `n` is an internal node, then

$$\begin{aligned}D(n,b) =&D(n.left, n.blocks[b-1].end_{\text{left}}+1)\cdot D(n.left, n.blocks[b-1].end_{\text{left}}+2)\cdots D(n.left, n.blocks[b].end_{\text{left}})\cdot\\ &D(n.right, n.blocks[b-1].end_{\text{right}}+1)\cdot D(n.right, n.blocks[b-1].end_{\text{right}}+2)\cdots D(n.right, n.blocks[b].end_{\text{right}})\end{aligned}$$

- $D_i(n,b)$ is the ith enqueue in $D(n,b)$.
- The order of the dequeue operations in the node n: $D(n)=D(n,1)\cdot D(n,2)\cdot D(n,3)...$
- $D_i(n)$ is the ith dequeue in $D(n)$.

def::lin **Definition 20** (Linearization). $L=E(root,1).D(root,1).E(root,2).D(root,2).E(root,3).D(root,3)...$

▶ In the non-root nodes, we only need ordering of enqueues and dequeues among the operations of their own type. Since `GetENQ()` only searches among enqueues and `IndexDEQ()` works with dequeues.

rueRefresh **Lemma 21** (trueRefresh). *Let t_i be the time an instance* `R` *of* `n.Refresh()` *is invoked and t_t be the time it terminates. If the* `TryAppend(new, s)` *of* `R` *returns* `true`*, then* $\mathtt{ops(EST_{n.left,\ t_i})} \cup \mathtt{ops(EST_{n.right,\ t_i})} \subseteq \mathtt{ops(EST_{n,\ t_t})}$.

Proof. Since `TryAppend` returns true a block `new` is written into `n.blocks[h]` in Line cas 313.

We show $\mathtt{ops(EST_{n.left,\ t_i})}\subseteq\mathtt{ops(EST_{n,\ t_t})}$. Let `h` be the value `n.Refresh()` reads from `n.head` at line readHead 310, $\mathtt{h_{left,i}}$ be the value of `n.left.head` at $\mathtt{t_i}$ and $\mathtt{h_{left,read}}$ be the value read from `n.left.head-1` at line lastLine 331. $\mathtt{end_{left}}$ field of the block returned by `CreateBlock(i)` is $\mathtt{h_{left,read}}$. By lines prevLine 332 and lastLine 331 the `new` block in `n.blocks[h]` contains $\mathtt{n.left.blocks[n.blocks[h\text{-}1].end_{left}\text{+}1..h_{left,read}]}$. Since `left.head` is read after $\mathtt{t_i}$ then $\mathtt{h_{left,read}}\mathtt{>}\mathtt{h_{left,i}}$ which means $\mathtt{ops(EST_{n.left,\ t_i})}\subseteq\mathtt{ops(n.left.blocks\ [0..h_{left,read}])}$. After the successful `TryAppend` in line cas 313 we know all blocks in $\mathtt{n.left.blocks[0..h_{left,read}\text{-}1}$ are subblocks of `n.blocks[0..h]` by the definition of subblock. At $\mathtt{t_t}$ we have `n.head>h` by Lemma lem::headProgress 10. So `n.blocks[1..h]` are in $\mathtt{EST_{n,t_t}}$ by definition of `EST`. Note that after line incrementHead2 321 we are sure that the `head` is incremented by Lemma lem::headInc 8) which means `n.head=h+1` at t_t so the new block is established at t_t and the new block contains the new operations which is what we wanted to show. The proof for $\mathtt{ops(EST_{n.right,\ t_i})}\subseteq\mathtt{ops(EST_{n,\ t_t})}$is the same.

□

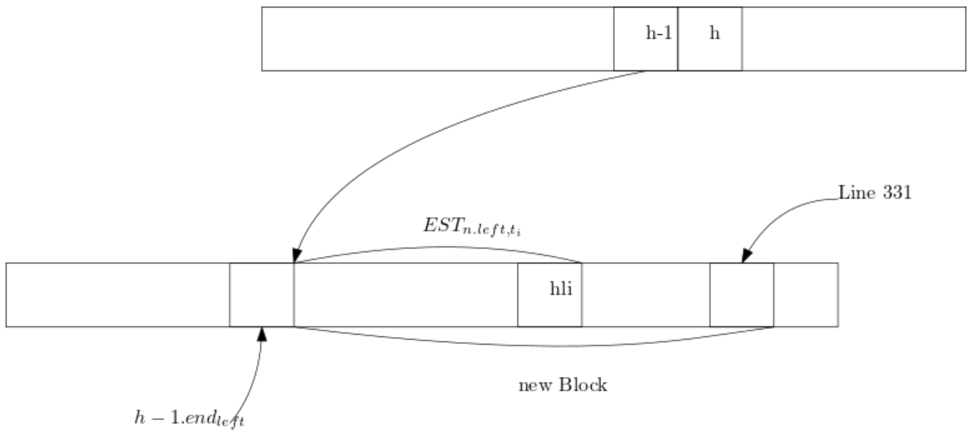


Figure 19: New established operations of the left child are in the new block.

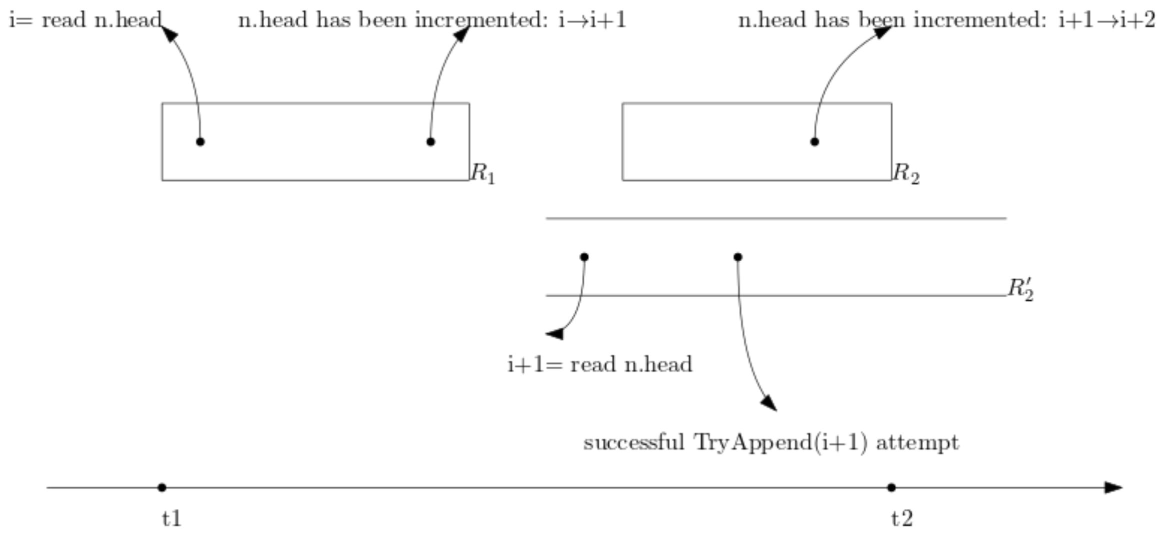
ueRefresh **Lemma 22** (Stronger True Refresh). *Let t_i be the time an instance of* `n.Refresh()` *read the head (Line readHead 310) and t_t be the time its* `TryAppend(new, s)` *terminates with and returns* `true` *(Line cas 313). We have* $\mathtt{ops(EST_{n.left,\ t_i})} \cup \mathtt{ops(EST_{n.right,\ t_i})} \subseteq \mathtt{ops(n.blocks)}$.

leRefresh **Lemma 23** (Double Refresh). *Consider two consecutive instances R_1, R_2 of* `Refresh()` *on* `n` *by process p. Let t_1 be the time R_1 is invoked and t_2 be the time R_2 terminated. If R_1 and R_2 both fail and return false, then we have* $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_2}\texttt{)}$.

Proof.

If Line $\overset{\texttt{cas}}{313}$ of R_1 or R_2 returns `true`, then the claim is held by Lemma $\overset{\texttt{lem::trueRefresh}}{21}$. Let R_1 read i and R_2 read $i+1$ from Line $\overset{\texttt{readHead}}{310}$. If R_2 reads some value greater than $i+1$ in Line $\overset{\texttt{readHead}}{310}$ it means a successful instance of `Refresh()` started after Line $\overset{\texttt{readHead}}{310}$ of R_1 and finished its Line $\overset{\texttt{incrementHead1}}{318}$ or $\overset{\texttt{incrementHead2}}{321}$ before $\overset{\texttt{readHead}}{310}$ of R_2, from Lemma $\overset{\texttt{lem::trueRefresh}}{21}$ by the end of this instance $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)}$ has been propagated.

Since R_2's `TryAppend()` returns `false` then there is another successful instance R_2' of `n.Refresh()` that has done `TryAppend()` successfully into `n.blocks[i+1]` before R_2 tries to append. Since R_2' creates the block after reading the value $i+1$ from `n.head`(Line $\overset{\texttt{readHead}}{310}$) and R_1 reads the value i from `n.head` and the `head`'s value is increaing by Lemma $\overset{\texttt{lem::headProgress}}{10}$ then $t_{R2'}\,{}_{\overset{\texttt{readHead}}{310}} > t_{R1}\,{}_{\overset{\texttt{readHead}}{310}} > t_1$ (See Figure $\overset{\texttt{fig::doubleRefresh}}{20}$). By Lemma $\overset{\texttt{lem::prectrueRefresh}}{22}$ after R_2''s `CAS` we have $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq \texttt{ops(n.blocks)}$. Also by Lemma $\overset{\texttt{lem::headInc}}{8}$ on R_2 the value of `n.head` head is more than $i+1$ after R_2́terminates, so the block appended by R_2' to `n` is established by then (`n.head>=i+2>i+1`). To summarize t_1 is before R_2''s `read n.head` and R_2''s successful `CAS` is before R_2's termination. So by Lemma $\overset{\texttt{lem::prectrueRefresh}}{22}$ $\texttt{ops(EST}_{\texttt{n.left, t}_1}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_1}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_2}\texttt{)}$. ☐



i= read n.head
n.head has been incremented: i→i+1
n.head has been incremented: i+1→i+2
R_1
R_2
R_2'
i+1= read n.head
successful TryAppend(i+1) attempt
t1
t2

leRefresh Figure 20: $t1 < r_1$ reading head $<$ incrementing `n.head` from i to $i+1 < R_2'$ reading head $<$ `TryAppend(i+1)` $<$ incrementing `n.head` from $i+1$ to $i+2 < t2$

this chain with more depth should be in the proof

Definition 24. $t_{before\ line}$ is the immediate time before running Line *line*. $t_{before\ line}$ is the immediate time after running Line *line*.

Corollary 25. $\texttt{ops(EST}_{\texttt{n.left, t}_{\texttt{before}\ \overset{\texttt{firstRefresh}}{302}}}\texttt{)} \cup \texttt{ops(EST}_{\texttt{n.right, t}_{\texttt{before}\ \overset{\texttt{firstRefresh}}{302}}}\texttt{)} \subseteq \texttt{ops(EST}_{\texttt{n, t}_{\texttt{after}\ \overset{\texttt{secondRefresh}}{303}}}\texttt{)}$

Proof. If the first `Refresh()` in line $\overset{\texttt{firstRefresh}}{302}$ returns `true` then by Lemma $\overset{\texttt{lem::trueRefresh}}{21}$ the claim holds. Also if first `Refresh()` failed and the second `Refresh()` succeeded the claim still holds by Lemma $\overset{\texttt{lem::trueRefresh}}{21}$. Finally, if both failed the claim is satisfied by Lemma $\overset{\texttt{doubleRefresh}}{23}$. ☐

lyRefresh **Corollary 26** (Propagate Step)**.** *All operations in* `n`*'s children's established blocks before running line 302 of a* `Propagate` *routine are guaranteed to be in* `n`*'s established blocks after line 303.*

Proof. If 302 or 303 succeed, the claim is true by Lemma 21. Otherwise Lines 302 and 303 satisfy the preconditions of Lemma 23. ☐

actlyOnce **Corollary 27.** *After* `Append(blk)` *finishes* `ops(blk)`⊆`ops(root.blocks[x])` *for exactly one* `x`*.*

Proof. After `Append(blk)`'s termination, `blk` is in `root.blocks` since `blk` is established in the leaf it has been added to. By applying Lemma 26 inductively it is propagated up to the root. Finally Lemma 12 shows only one block in the root contains `blk`. ☐

blockSize **Lemma 28** (Block Size Upper Bound)**.** *Each block contains at most one operation of each processs.*

Proof. To derive a contradiction, assume there are two operations op_1 and op_2 of process p in block b in node n. Without loss of generality op_1 is invoked earlier than op_2. A process cannot invoke more than one operations concurrently, so op_1 has to be finished before op_2. By Corollary lem::appendExactlyOnce 27, before appending op_2 to the tree op_1 exists in every node on the path from p's leaf to the root, because op_1's `Append` is finished before op_2's `Append` starts. So, there is some block b' before b in n containing op_1. Existence of op_1 in b and b' contradicts Lemma append 12. □

ocksBound **Lemma 29** (Subblocks Upperbound)**.** *Each block has at most p direct subblocks.*

Proof. The claim follows directly from Lemma blockSize 28 and the observation that each block appended to the tree contains at least one operation, due to the test on Line addOP 312. We can also see the blocks in the leaves have exactly one operation in the `Enqueue()` and `Dequeue()` routines. □

get **Lemma 30** (Get correctness). *If* `n.blocks[b].num`$_{\texttt{enq}}$$\geq$`i` *then* `n.GetENQ(b,i)` *returns the* `element` *enqueued by* $E_i(n,b)$.

Proof. We are going to prove this lemma by induction on the height of node `n`. For the base case `n` is a leaf. Leaf blocks each contain exactly one operation, so by the preconditions of `GetENQ()`, only `n.GetENQ(b,1)` can be called and `n.blocks[b]` contains an enqueue. At Line getBaseCase 403 `n.GetENQ(b,1)` returns the `element` of the `enqueue` operation stored in the bth block of leaf `n` .

For the induction step we prove `n.GetENQ(b,i)` returns $E_i(n,b)$, if `n.child.GetENQ(b,i)` returns $E_i(n.child,b)$. In Line leftOrRight 404 it is decided for the non-leaf nodes that the `i`th enqueue in `b`th block of internal node `n` is in the `n.blocks[b]`'s left child or right child subblocks. From Definition ordering 19 of $E(n,b)$ we know enqueue operations in a block are ordered by their process id and since the leaves of the tree are ordered by process id from left to right, thus operations from the left subblocks come before operations from the right subblocks in a block (See Figure figGet 21). Furthermore the `num`$_{\texttt{enq-left}}$ field in a block stores the number of `enqueue()` operations from the blocks's subblocks in the left child of `n`. So ith enqueue operation is propagated from the right child if `i` is greater than `b.num`$_{\texttt{enq-left}}$. otherwise we should search for the ith enqueue in the left child. By definition def::ops 14 and def::subblock 11 we need to search in subblocks of `n.blocks[b]` from the range `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` $\cup$ `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`.

If the ith `enqueue` of `n.blocks[b]` is in the left child it would be ith enqueue in `n.left.blocks[n.blocks[i-1].end`$_{\texttt{left}}$`+1..n.blocks[i].end`$_{\texttt{left}}$`]` by Definition def::subblock 11. Also we know there are $eb = n.blocks[b-1].sum_{enq-left}$ enqueues in the blocks before this range, so $E_i(n,b)$ is $E_{i+eb}(n.left)$ which is $E_{i'}(n.left,b')$ for some b' and i'. We can compute b' search for $i+eb$th enqueue in `n.left` and i' is `i+eb-n.left.blocks[`$b'-1$`].sum`$_{\texttt{enq}}$. The parameters in leftChildGet 405 are for searching $E_{i+eb}(n.left)$ in `n.left.block` in the expected range of blocks, so this `BSearch` returns the index of the subblock containing $E_i(n,b)$.

Else if the enqueue we are looking for is in the right child then there are `n.blocks[b].num`$_{\texttt{enq-left}}$ enqueues ahead of it in `n.blocks[b]` but not in `n.right.blocks[n.blocks[i-1].end`$_{\texttt{right}}$`+1..n.blocks[i].end`$_{\texttt{right}}$`]`. So we need to search for `i-n.blocks[b].num`$_{\texttt{enq-left}}$`+` `n.blocks[b-1].sum`$_{\texttt{enq-right}}$ (Line rightChildGet 409). Other parameters are assigned similar for the left child. So in both cases the direct subblock containing $E_i(n,b)$ is computed in Lines leftChildGet rightChildGet 405 and 409.

Finally, `n.child.GetENQ()` is invoked on the subblock containing $E_i(n,b)$ which returns $E_i(n,b)$ by the hypothesis of the induction. □

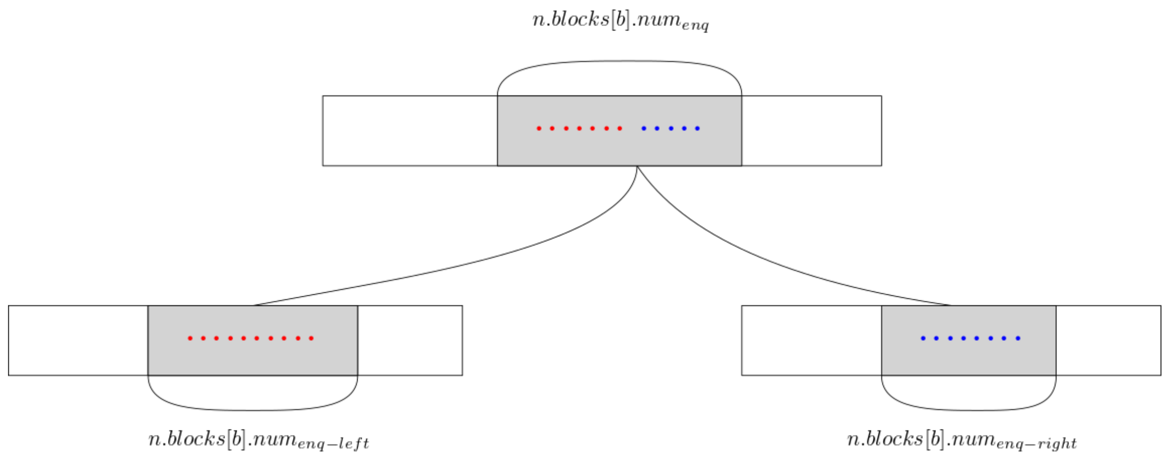


Figure 21: The number and ordering of the enqueue operations propagated from the left and the right child to `n.blocks[b]`. Enqueue operations from the left subblocks (colored red), are ordered before the enqueue operations from the right child (colored blue). figGet

dsearch **Lemma 31** (DSearch correctness)**.** *Assume* `root.blocks[end].sum`$_{\texttt{enq}}$$\geq$`e` *and* $E_e(root)$*'s* `element` *is the response to some* `Dequeue()` *operation in* `root.blocks[end]`. `DSearch(e, end)` *returns* `<b, i>` *such that* $E_i(root, b) = E_e(root)$.

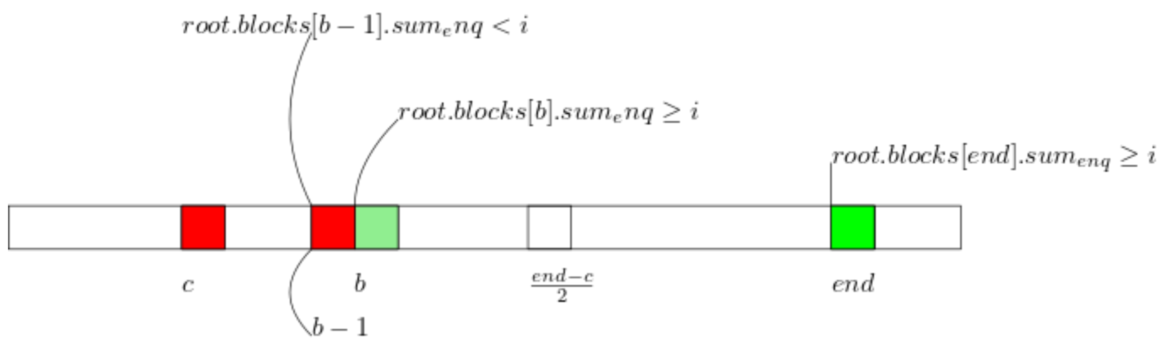
Proof. It is trivial to see that the doubling search from `root.blocks[end]` to `root.blocks[0]` will find $E_e(root)$ eventually. Because `root.blocks[].sum`$_{\texttt{enq}}$ is an increasing value from 0 to some value greater than `e`. So there is a `b` that `root.blocks[b].sum`$_{\texttt{enq}}$$> e$ but `root.blocks[b-1].sum`$_{\texttt{enq}}$$< e$.

First we show `end-b`$\leq$ 2$\times$`root.blocks[b].size +root.blocks[end].size +1`. From line addOP 312, we know that size of the every block in the tree is greater than 0. So each block in `root.blocks[b..end]` contains at least one `Enqueue` or at least one `Dequeue`. Suppose there were more than `root.blocks[b].size Dequeues` in `root.blocks[b+1..end-1]`. Then the queue would become empty at some point after `blocks[b]`'s last operations and before `root.blocks[end]`'s first operation. Which means the response to to a `Dequeue` in `root.blocks[end]` could not be in $E(n, b)$. Furthermore since the size of the queue would become `root.blocks[end].size` after the `root.blocks[end]`, there cannot be more than `root.blocks[b].size + root.blocks[end].size Enqueues`. Because there can be at most `root.blocks[b].size Dequeues` and the final size is `root.blocks[end].size` . Overall there can be at most 2$\times$`root.blocks[b].size + root.blocks[end].size` operations in `root.blocks[b+1..end-1]` and since each block size is ≥ 1 thus there are at most 2$\times$`root.blocks[b].size + root.blocks[end].size` blocks in between `root.blocks[b]` and `root.blocks[end]`. So `end-b`$\leq$ 2$\times$`root.blocks[b].size +root.blocks[end].size +1`. See Figure end-b ??.

Now that we know there are at most `root.blocks[b].size +root.blocks[end].size` blocks in between `root.blocks[b]` and `root.blocks[end]` then with doubling search in $\Theta\big($log(`root.blocks[b].size +root.blocks[end].size`)$\big)$ steps we reach `start=c` that the `root.blocks[c].sum`$_{\texttt{enq}}$ is less than `e` and `end-c` is not more than 2$\times$`root.blocks[b].size +root.blocks[end].size` . Beause otherwise, then `(end-c)/2` satisfied the `root.blocks[(end-c)/2].sum`$_{\texttt{enq}}$`<e`. In line doubling 804 the differnece between `end` and `start` is doubled. See Figure fig::doubling 22.

After computing `b`, the value `i` is computed via the definition of `sum`$_{\texttt{enq}}$ in constant time (Line DSearchComputei 807). So the routine non constant part is the binary search which takes Θ(log`root.blocks[b].size +root.blocks[end].size`)) steps from the first paragraph.

$\square$


$root.blocks[b-1].sum_enq < i$
$root.blocks[b].sum_enq \geq i$
$root.blocks[end].sum_{enq} \geq i$
c
b
$\frac{end-c}{2}$
end
$b-1$

:doubling Figure 22: Distance relations between b, c, end

Lemma 32. *Let n.propagates be the number of groups of blocks that have been propagated from node* `n` *to its parent (successful* `n.parent.Refresh()`*). We have* $\texttt{num}_{\texttt{propagated}} \leq$ *n.propagates* $\leq \texttt{num}_{\texttt{propagated}}+\texttt{p}$. `p` *is the number of processes.*

Proof. $\texttt{num}_{\texttt{propagated}}$ is incremented after propagating (Line $\overset{\texttt{incNP}}{316}$). Since maybe some process falls sleep before incrementing $\texttt{num}_{\texttt{propagated}}$ it may be behind by `p`. ☐

Lemma 33. `super[]` *preserves order from child to parent; i.e. if in node* `n` *block* `b` *is before* `c` *then* `b.group` $\leq$ `c.group`

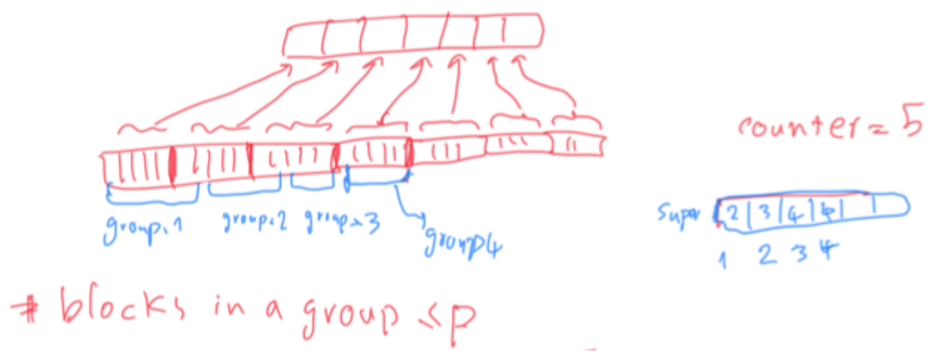
Proof. Line $\overset{\texttt{setGroup}}{329}$. Since $\texttt{num}_{\texttt{propagated}}$ is increasing. ☐

Lemma 34. *Let* `b, c` *be in node* `n`*, if* `b.group` $\leq$ `c.group` *then* `super[b.group]` $\leq$ `super[c.group]`

Proof. Line $\overset{\texttt{setSuper}}{315}$. ☐

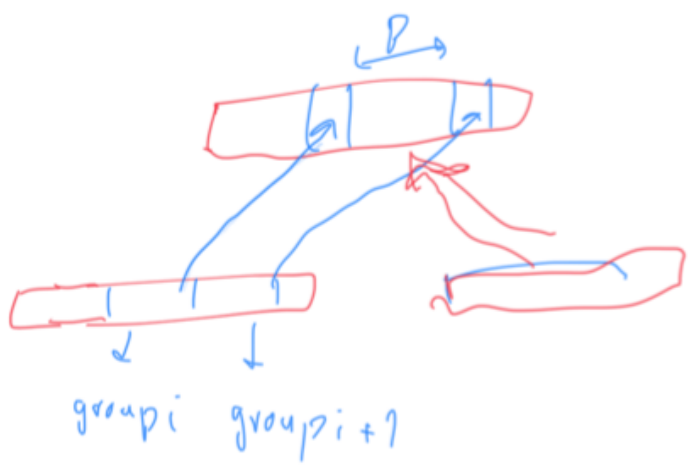
Lemma 35. *The number of the* `blocks` *with* `group=i` *in a node is* $\leq p$.

Proof. For the sake of simplicity we assumed all the blocks are propagated from the left child. ☐



Lemma 36. `super[i+1]-super[i]`$\leq p$

Proof. 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. ☐



erCounter **Lemma 37** (super property)**.** *If* `super[i]` $\neq$ `null` *in node* `n`*, then* `super[i]` *is the index of the superblock of a block with* `time=i` *in* `n.parent.blocks`.

uperRange **Lemma 38.** *Superblock of* `b` *is within range* $\pm 2p$ *of the* `super[b.group]`.

Proof. `super[i]` is the index of the superblock of a block containing block b, followed by Lemma superCounter 37. `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:

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

□

Lemma 39. *Search in each level of* `IndexDeq()` *takes* $O(\log p)$ *steps.*

Proof. Show preconditions are satisfied and the range is p. □

uperBlock

Lemma 40 (Computing SuperBlock)**.** *For the* `superblock` *value computed in line* computeSuper *418 of* `n.IndexDEQ(b,i)` *we have* `n.parent.blocks[superblock]` *contains* $D_{n,b,i}$.

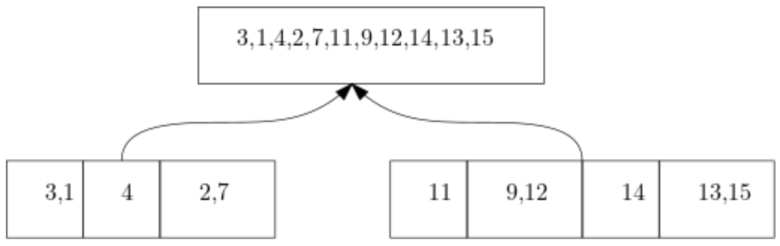
Proof. First we show the value read for `super[b.group]` in line 418 is not null. Values $\texttt{np}_{\texttt{dir}}$ read in lines setNP 337, `super` are set before incrementing in lines setSuper lineNP 315,316. So before incrementing $\texttt{num}_{\texttt{propagated}}$, $\texttt{super[num}_{\texttt{propagated}}\texttt{]}$ is set so it cannot be null while reading. Then by Lemma superRange 38if we search in the range p, we can find the superblock.

□

Lemma 41 (Index correctness)**.** *If* $\texttt{n.blocks[b].num}_{\texttt{deq}}\texttt{≥i}$ *then* `n.IndexDEQ(b,i)` *returns the rank in* $D(root)$ *of* $D_{n,b,i}$.

Proof. We will prove this by induction on the distance of `n` from the `root`. We can see the base case where `n` is root is trivial (Line indexBaseCase 415). In the non-root nodes `n.IndexDEQ(b,i)` computes the superblock of the ith Dequeue in the bth block of `n` in `n.parent` by Lemma superBlock 40 (Line computeSuper 418). After that the order in $D(n.parent, superblock)$ is computed. Note that by Lemma blockSize 28 in each block there is at most one operation from each process and operations of one type are ordered based on the order in the subblocks (See Figure fig::orderFromSubblocks 23). Finally `index()` is called on `n.parent` recursively and it returns the correct response from induction hypothesis. If the operation was propagated from the right child the number of dequeues from the left child are added to it (Line considerRight ??), because the left child operations come before the right child operations (Definition ordering 19). □

Make sure to show preconditions of all invocation of `BSearch` *are satisfied.*



Subblocks

Figure 23: Relation of ordering of operations of a block from its subblocks

Definition 42. Assume the operations in L are applied on an empty queue. If element of `enqueue e` is the response to `dequeue d` then we say `R(d)=e`. If `d` 's response id `null` (queue is empty) then `R(d)=null`.

Definition 43. In an execution on a queue, the dequeue operations that return some value are called *non-null dequeues.*

nseToADeq **Observation 44.** *In a sequential execution on a queue, kth non-null dequeue returns the* `element` *of kth enqueue.*

rrectness **Lemma 45.** `root.blocks[b].size` *is the size of the queue if the operations in the prefix for the bth block in the root are applied with the order of L.*

Proof. need to say? :: If the size of a queue is greater than 0 then a `Dequeue()` would decrease the size of the queue, otherwise the size of the queue remains 0. By definition ordering 19 enqueue operations come before dequeue operations in a block in L.

We prove the claim by induction on `b`. Base case `b=0` is trivial since the queue is initialy empty and `root.blocks[0].size=0`. For `b=i` we are going to use the hypothesis for `b=i-1`. If there are more than `root.blocks[i-1].size+ root.blocks[i].sum`$_{\texttt{enq}}$ dequeue operations in `root.blocks[i]` then the queue would become empty after `root.blocks[i]`. Otherwise we can compute the size of the queue after bth block using with this equality `root.blocks[b].size= root.blocks[b-1].size+ root.blocks[b].sum`$_{\texttt{enq}}$`-` `root.blocks[b].sum`$_{\texttt{deq}}$ (Line computeLength 342). See Table qhistory 4 for an example of running some blocks of operations on an empty queue. □

mberOfNND **Lemma 46** (Duality of #non-null dequeues and `block.size`)**.** *If the operations are applied with the order of L, the number of non-null dequeues in the prefix for a block* `b` *is* `b.sum`$_{\texttt{enq}}$`-b.size`

Proof. There are `b.sum`$_{\texttt{enq}}$ enqueue operations in the prefix for `b`, then the size of the queue after the prefix for `b` is `#enqs - #non-null dequeues` in the prefix for `b`, by Observation 35. So `#non-null dequeues` is `b.sum`$_{\texttt{enq}}$`-b.size` . The correctness of the `block.size` field is shown in Lemma sizeCorrectness 45. □

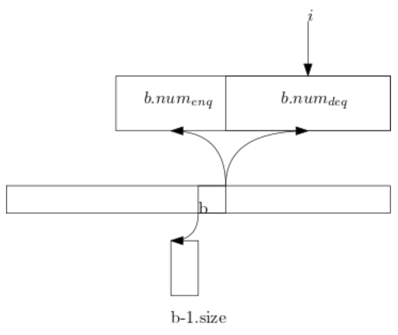
ullReturn **Lemma 47.** `R(D`$_{\texttt{root,b,i}}$`)` *is* `null` *iff* `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0`.

mputeHead **Lemma 48** (Computing Response)**.** `FindResponse(b,i)` *returns* `R(D`$_{\texttt{root,b,i}}$`).element`.

Proof. First note that by Definition ordering 19 the linearization ordering of operations will not change as new operations come so instead of talking about the linearization of operations before the $E_i(root, b)$ we talk about what if the whole operation in the linearization are applied on a queue.

$D_{root,b,i}$ is $D_{root,root.blocks[b-1].sum_{deq}+i}$ from the definition ordering 19 and sum_{enq}. $D_{root,b,i}$ returns `null` if `root.blocks[b-1].size + root.blocks[b].num`$_{\texttt{enq}}$`- i <0` by Lemma nullReturn 47 (Line checkEmpty 220). Otherwise if it is d'th non-null dequeue in L it returns d'th enqueue by Observation responseToADeq 44 . By Lemma numberOfNND 46 there are `root.blocks[b-1].sum`$_{\texttt{enq}}$ `- root.blocks[b-1].size` non-null dequeue operations before prefix for `root.blocks[b-1]`. Note that the dequeues in `root.blocks[b]` before the ith dequeue are non-null dequeues. So the response is $E_{i-root.blocks[b-1].size+root.blocks[b-1].sum_{\text{deq}}}(root)$ (Line computeE 222). See figure computeResponseDetail 24.

After computing `e` we can find `b,i` such that $E_i(root, b) = E_e(root)$ using `DSearch` and then find its `element` using `GetEnq` (Line findAnswer 223). □



nseDetail Figure 24: The position of $E_i(root, b)$.

	`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

`qhistory` Table 4: An example of root blocks fields. Blocks are from left to right and operations in the blocks are also from the left to right.

Theorem 49 (Main)**.** *The queue implementation is linearizable.*

Proof. We choose L in Definition `ordering` 19 to be linearization ordering of operations and prove if we linearize operations as L the queue works consistently. □

Lemma 50 (satisfiability)**.** *L can be a linearization ordering.*

Proof. To show this we need to say if in an execution, $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts then $\texttt{op}_1$ is linearized before $\texttt{op}_2$. If $\texttt{op}_1$ terminates before $\texttt{op}_2$ starts it means $\texttt{op}_1$`.Append()` is terminated before $\texttt{op}_2$`.Append()` starts. From Lemma `append` 12 $\texttt{op}_1$ is in `root.blocks` before $\texttt{op}_2$ propagates so $\texttt{op}_1$ is linearized before $\texttt{op}_2$ by Definition `ordering` 19.

Once some operations are aggregated in one block they will be propagated together up to the root and we can linearize them in any order among themselves. Furthermore in L we arbitrary choose the order to be by process id, since it makes computations in the blocks faster . □

Lemma 51 (correctness)**.** *If operations are applied as L on a sequential queue, the sequence of the responses would be the same as our algorithm.*

Proof. Old parts to review 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 `append` 12. The time op_1 ends it is in root, before op_2, by Definition `ordering` 19 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 `computeHead` 48 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 `get` 30 we know that the returning response is the computed index element.

□

Lemma 52 (Amortized time analysis)**.** `Enqueue()` *and* `Dequeue()`*, each take* $O(\log^2 p + \log q)$ *steps in amortized analysis. Where* p *is the number of processes and* q *is the size of the queue at the time of invocation of operation.*

Proof. `Enqueue(x)` consists of creating a `block(x)` and appending it to the tree. The first part takes constant time. To propagate `x` to the root the algorithm tries two `Refresh`es in each node of the path from the leaf to the root (Lines `firstRefresh` 302, `secondRefresh` 303). We can see from the code that each `Refresh` takes constant number of steps since creating a block is done in constant time and does $O(1)$ `CAS`es. Since the height of the tree is $\Theta(\log p)$, `Enqueue(x)` takes $O(\log p)$ steps.

A `Dequeue()` creates a block with null value element, appends it to the tree, computes its order among enqueue operations, and returns the response. The first two part is similar to an `Enqueue` operation. To compute the order of a `dqueue` in $D(n)$ there are some constant steps and `IndexDeq()` is called. `IndexDeq` does a search with range p in each level (Lemma `superRange` 38) which takes $O(log^2 p)$ in the tree. In the `FindResponse()` routine `DSearch()` in the root takes $\Theta(\log($`root.blocks[b].size +root.blocks[end].size`) by Lemma `dsearch` 31, which is $O(\log$ size of the queue when `enqueue` is invoked) + log size of the queue when `dequeue` is invoked). Each search in `GetEnq()` takes $O(\log p)$ since there are $\leq p$ subblocks in a block (Lemma `subBlocksBound` 29), so `GetEnq()` takes $O(\log^2 p)$ steps.

If we split `DSearch` time cost between the corresponding `Enqueue`, `Dequeue`, in amortized we have `Enqueue` takes $O(\log p + q)$ and `Dequeue` takes $O(\log^2 p + q)$ steps. □

Lemma 53 (CASes invoked)**.** *An* `Enqueue()` *or* `Dequeue()` *operation, does at most* $4 \log p$ `CAS` *operations.*

Proof. In each height of the tree at most 2 times `Refresh()` is invoked and every `Refresh()` has 2 `CAS`es, one in Line `cas` 313 and one in Lines `incrementHead1` 318 or `incrementHead2` 321. □