

# A Wait-free Queue with Poly-logarithmic Worst-case Step Complexity

ANONYMOUS AUTHOR(S)\*

In this work, we introduce a novel linearizable wait-free queue implementation. Linearizability and lock-freedom are standard requirements for designing shared data structures. To the best of our knowledge, all of the existing linearizable lock-free queues in the literature have a common problem in their worst case, called the CAS Retry Problem. We show that our algorithm avoids this problem with the helping mechanism which we use and has a worst-case running time better than prior lock-free queues. The amortized number of steps for an Enqueue or Dequeue in our algorithm is  $O(\log^2 p + \log q)$ , where  $p$  is the number of processes and  $q$  is the size of the queue when the operation is linearized.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: concurrent data structures, wait-free queues

## 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. It is not hard to see why multiple processes cannot share a sequential data structure. For example, consider two processes trying to append to 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 then  $Q$  does the same. In this run,  $P$ 's update is overwritten, which makes the queue inconsistent. One solution is to use locks; whenever a process wants to update or query a data structure, the process locks the data structure, and others have to wait until the lock is released to read or update the data structure. 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 process holding the lock can progress when the data structure is locked. For these reasons, we are interested in the design of an efficient shared queue without using locks.

A sequential queue stores a sequence of elements and supports two operations, enqueue and dequeue. Enqueue( $e$ ) appends the element  $e$  to the sequence stored. Dequeue removes and returns the first element in the queue. If the queue is empty, it returns null. In a concurrent version of the queue data structure, operations do not happen one at a time. The question that may arise is, "What properties matter for the implementation of a shared 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.

A system is called *asynchronous* when processes in the system run at arbitrarily varying speeds, i.e., the scheduling of each process is independent from the scheduling of other processes. Our model is an asynchronous shared-memory distributed system of  $p$  processes. Herlihy [?] showed that one cannot implement concurrent versions of all data structures with multi-reader multi-writer registers alone and introduced a hierarchy based on how powerful objects are to solve the consensus problem among processes. Objects like LL/SC or CAS can be used to reach a consensus among any number of processors. A CAS object provides an atomic Compare&Swap(new, old) operation:

if the value stored in the object is old, it updates the value to new and returns true; otherwise it returns false. In this work, we use Compare&Swap (CAS) objects to synchronize among processes.

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.

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.

We will present a wait-free linearizable queue, which to the best of our knowledge is the first wait-free queue whose operations run in a poly-logarithmic number of steps. We design a tournament tree in which each process tries to propagate its operations along a path from the process's leaf up to the root to be linearized. Processes can do queries to get information about the linearization stored in the root. We create a helping mechanism for when a process is propagating its operation to a node, such that new operations are ensured to be propagated one step further up the tree after at most 2 CAS invocations. The amortized number of steps for an Enqueue or Dequeue, is  $O(\log^2 p + \log q)$ , where  $q$  is the size of the queue when the operation is linearized. CAS instructions cost more than other instructions for the processor. Each operation in our queue does  $\Theta(\log p)$  CAS operations compared to  $\Omega(p)$  CAS steps for previous queues in the worst case. We use unbounded memory space in our queue and present a way to reduce the memory size to the total number of operations. Furthermore, we do not address garbage collection but we describe a way we think it can be handled.

The rest of the thesis is organized as follows. Chapter 2 gives an outline of the related work done in the area and the motivation for our implementation. Previous lock-free queues and their common problem are presented in Section 2.1. In Section 2.2, we mention some restricted lock-free queues. Section 2.3 talks about poly-logarithmic constructions of shared objects. Section 2.4 ends with a lower bound on the amortized time complexity of shared queues.

In Chapter 3 we introduce a poly-logarithmic step wait-free queue. In Section 3.1 we give a high-level description of our implementation. We also discuss the motivation and requirements in our design to achieve poly-log time. Section 3.2 contains the algorithm itself and there is an example of a concurrent execution of the algorithm described in Section 3.3.

We prove the correctness of our queue in Chapter 4 by showing it is linearizable.

Chapter 5 is devoted to a complexity analysis for our implementation. We compute the number of CAS instructions our algorithm invokes and the worst-case and amortized running time of the queue. In the end, we prove our queue is wait-free.

Finally, we give some concluding remarks in Chapter 6. It contains how we can improve our algorithm's memory usage and how to use its idea in designing other wait-free data structures.

## 2 RELATED WORK

In this section, we look at previous lock-free queues. The amortized step complexity of all the lock-free queues includes an  $\Omega(p)$  term that comes from the worst case, where all  $p$  processes try to do an enqueue or a dequeue simultaneously. Morrison and Afek call this the *CAS retry problem* [?].

### 2.1 List-based Queues

Michael and Scott [?] 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. A shared variable *Head* points to the first node in the linked list that has not been dequeued, and the *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  $\Omega(p)$  per enqueue. Similarly, dequeue can take  $\Omega(p)$  steps.

Moir, Nussbaum, and Shalev [?] 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 [?] 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 using CAS operations. However, like the previous algorithms, if there are  $p$  concurrent enqueue operations in a basket, the amortized step complexity remains  $\Omega(p)$  per operation.

Ladan-Mozes and Shavit [?] presented an optimistic approach to implement a queue. MS-queue uses two CASes to do an enqueue: one to change the tail to the new node and another one to change the next pointer of the previous node to the new node. They use a doubly-linked list to do fewer CAS operations in an Enqueue than MS-queue. As in previous algorithms, the worst case happens in the case where the contention is high: when  $p$  concurrent enqueues happen, their nodes have to be appended to the linked list one by one. The amortized complexity is still  $\Omega(p)$  CASes.

Hendler et al. [?] proposed a new paradigm called flat combining. The key idea behind flat combining is to allow a combiner who has acquired the global lock on the data structure to learn about the requests of threads on the queue, combine them and apply the combined results on the data structure. Their queue is linearizable but not lock-free and they present experiments that show their algorithm performs well in some situations.

Gidenstam, Sundell, and Tsigas [?] introduced a new algorithm using a linked list of arrays. The queue is stored in a shared array where head and tail pointers point to the current elements in the queue. When the array is full, an empty array is linked to the array and tail pointers are updated. A global head points to the array containing the first element in the queue, and each process has a local head index that points to the first element in that array. Global tail and local tail pointers are similar. A process updates the position of the pointers after it does an operation. One process might go to sleep before setting the pointers, so the pointers might be behind their real places. They mention how to scan the arrays to update pointers while doing an operation. A process writes an element in the location head by a CAS instruction, so if  $p$  processes try to enqueue simultaneously, the amortized step (and CAS) complexity remains  $\Omega(p)$ . Their design is lock-free but not wait-free.

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

Milman et al. [?] designed a lock-free queue supporting futures. In their queue, operations return future objects instead of responses. Later when the response is needed, it can be evaluated from the future object. They also define a weaker linearizability condition such that each operation can

be linearized between its invocation and when its future is evaluated. Their idea of batching allows a sequence of operations to be submitted as a batch for later execution on the MS-queue. They use some properties of the queue size before and after a batch, similar to a part of our work. Their queue is not wait-free: in fact, if the batch sizes are 1, then the queue is like MS-queue.

Nikolaev and Ravindran [?] present a wait-free queue that uses the fast-path slow-path methodology introduced by Kogan and Petrank [?]. Their work is based on a circular queue using bounded memory. When a process wishes to do an enqueue or a dequeue, it starts two paths. The fast path ensures good performance while the slow path ensures termination. They show that these two paths do not affect each other and the queue remains consistent. If a process makes no progress, other processes help its slow path to finish. The helping phase suffers from the CAS Retry Problem because processes compete in a CAS loop to decide which succeeds to help. Because of this, the amortized complexity cannot be better than  $\Omega(p)$ .

The CAS Retry Problem is not limited to list-based queues; array-based queues also share it [?]. Our motivation is to overcome this problem and present a wait-free sublinear queue.

## 2.2 Restricted Queues

David introduced the first sublinear concurrent queue [?]. Even though his algorithm does  $O(1)$  steps for each operation, it is a single-enqueuer single-dequeuer queue and uses infinite memory. The author states that to reduce memory usage to be bounded, the time per operation increases linearly.

Jayanti and Petrovic introduced a wait-free poly-logarithmic multi-enqueuer single-dequeuer queue [?]. We use their idea of having a tournament tree among processes to agree on the linearization of operations to design a poly-logarithmic multi-enqueuer multi-dequeuer queue. Unlike their work, our algorithm does not put a limit on the number of concurrent dequeuers.

## 2.3 Universal Constructions and Other Poly-log Time Data Structures

A *universal construction* is an algorithm that can implement a shared version of any given sequential object. The first universal construction was introduced by Herlihy [?]. 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 [?]. He also mentions that the universal construction by Afek, Dauber, and Touitou [?] can be modified to  $O(\log p)$  worst-case step complexity, using atomic access to  $\Omega(p \log p)$ -bit words. Chandra, Jayanti and Tan introduced a semi-universal construction that achieves  $O(\log^2 p)$  shared accesses [?]. However, their algorithm cannot be used to create a queue. We mention a non-practical universal construction with a poly-log number of CAS instructions in the last paragraph of page 13.

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 [?]. Their idea to achieve logarithmic complexity is to use a tree storing the Fetch&Inc operations invoked by processes. When a process wants to do a Fetch&Inc it adds its Fetch&Inc to the tree and returns the number of elements in the tree. There are some similarities between designing a queue and a Fetch&Inc object. A Fetch&Inc object can be constructed from a queue. The algorithm by Ellen and Woelfel is interesting because of the similarities between Fetch&Inc objects and queues. Also, it is one of the few wait-free data structures achieving poly-logarithmic complexity.

## 2.4 Attiya–Fouren Lower Bound

Because of the CAS retry problem in previous list-based queues one might guess the  $\Omega(p)$  term is inherent in the time complexity of concurrent queues. Attiya and Fouren gave a lower bound on amortized complexity of lock-free queues with regard to  $c$ , the number of concurrent processes. Their

result says if  $c$  is  $O(\log \log p)$ , any implementation of queues using reads, writes and conditional operations like CAS has  $\Omega(c)$  amortized complexity [?]. The surprising point is that since their result is about when contention is low, their lower bound does not contradict our algorithm's complexity and we manage to reach poly-log time complexity.

### 3 QUEUE IMPLEMENTATION

In our model there are  $p$  processes doing Enqueue and Dequeue operations on a queue concurrently. We design a linearizable wait-free queue with  $O(\log^2 p + \log q)$  steps per operation, where  $q$  is the number of elements in the queue at the time of linearization. We avoid the  $\Omega(p)$  worst-case step complexity of existing shared queues based on linked lists or arrays, which suffer from the CAS Retry Problem.

There is a shared binary tree among the processes to agree on one total ordering of the operations invoked by processes. The tree is called a *tournament tree* (see Figure 1). Each process has a leaf in which the operations invoked by the process are stored in order. When a process wishes to do an operation it appends the operation to its leaf and tries to propagate its new operation up to the tree's root. Each node of the tree keeps an ordering of operations propagated up to it. All processes agree on the sequence of operations in the root and this ordering is used as the linearization ordering.

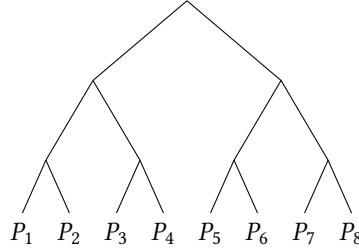


Fig. 1. Each of the processes  $P_1, P_2, \dots, P_p$  has a leaf and in each node there is an ordering of operations stored. Each process tries to propagate its operations up to the root, which stores a total ordering of all operations.

To propagate operations to a node  $n$  in the tree, a process observes the operations in both of  $n$ 's children that are not already in  $n$ , merges them to create an ordering and then tries to append the ordering to the sequence stored in  $n$ . We call this procedure  $n$ .Refresh (see Figure 2). A Refresh on  $n$  with a successful append helps to propagate their operations up to  $n$ . We shall prove that if a process invokes Refresh on the node  $n$  two times and fails to append the new operations to  $n$  both times, the operations that were in  $n$ 's children before the first Refresh are guaranteed to be in  $n$  after the second failed Refresh. We sketch the argument here.

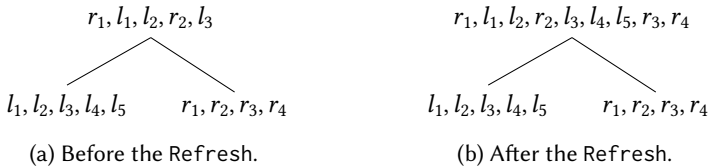


Fig. 2. Before and after a  $n$ .Refresh with a successful append. Operations propagating from the left child are labelled with  $l$  and from the right child with  $r$ .

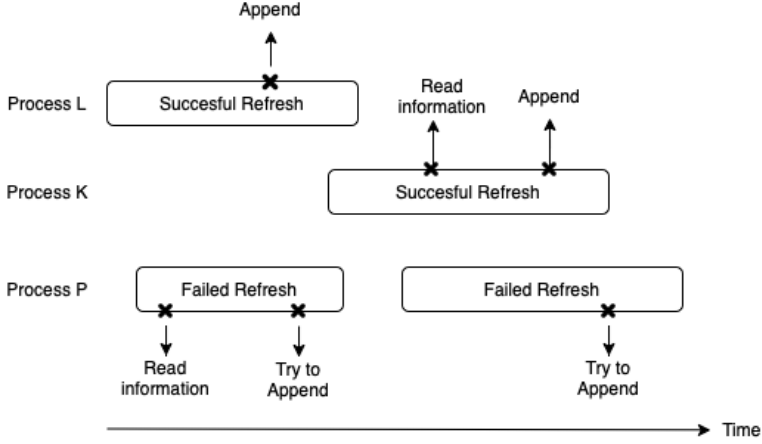


Fig. 3. Time relations between the concurrent successful Refreshes and the two consecutive failed Refreshes.

We use CAS (Compare&Swap) instructions to implement the Refresh's attempt to append described in the previous paragraph. The second failed Refresh of  $P$  is assuredly concurrent with a successful Refresh that has read its information after the invocation of the first failed Refresh (see Figure 3). This is because some process  $L$  does a successful append during  $P$ 's first failed Refresh attempt, and some process  $K$  performs a Refresh that reads its information after  $L$ 's append and then performs a successful append during  $P$ 's second failed Refresh. Process  $K$ 's Refresh helps to append the new operations that were in  $n$ 's children before  $P$ 's first failed Refresh, in case they were not already appended. After a process appends its operation into its leaf it can call Refresh on the path up to the root at most two times on each node. So, with  $O(\log p)$  CASes an operation can ensure it appears in the linearization. This cooperative solution allows us to overcome the CAS Retry Problem.

It is not efficient to explicitly store the sequence of operations in each node because each operation would have to be copied all the way up to the root; doing this would not be possible in poly-logarithmic time. Instead we use an implicit representation of the operations propagated together. Furthermore, we do not need to maintain an ordering on operations propagated together in a node until they have reached the root. It is sufficient to only keep track of sets of operations propagated together in each Refresh and then define the linearization ordering only in the root (see Figure 4). Achieving a constant-sized implicit representation of operations in a Refresh allows us to do CAS instructions on fixed-size objects in each Refresh. To do that, we introduce *blocks*. A block stores information about the operations propagated by a Refresh. It contains the number of operations from the left and the right child propagated to the node by the Refresh procedure. See Figure 5 for an example. A node stores an array of blocks of operations propagated up to it. A propagate step aggregates the new blocks in the children into a new block and stores the new block in the parent. We call the aggregated blocks *subblocks* of the new block and the new block the *superblock* of them. In each Refresh there is at most one operation from each process trying to be propagated, because one process cannot invoke two operations concurrently. Thus, there are at most  $p$  operations in a block. Furthermore, since the operations in a Refresh step are concurrent we can linearize them among themselves in any order we wish, because if two operations are read in one successful Refresh step in a node they are going to be propagated up to the root together. Our choice is to put the operations propagated from the left child before the operations propagated

from the right child. In this way, if we know the number of operations from the left child and the number of operations from the right child in a block, we have a complete ordering of the operations.

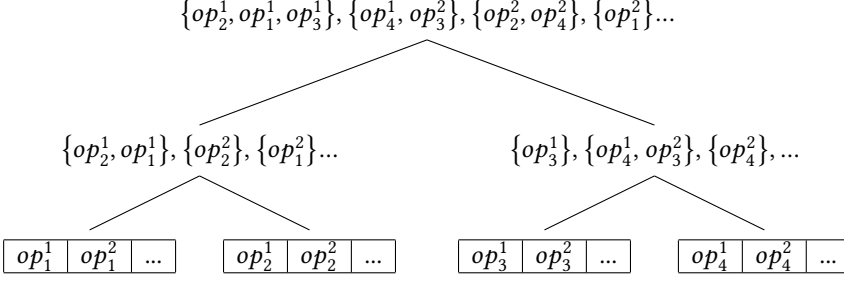


Fig. 4. Leaves are for processes  $P_1$  to  $P_4$  from left to right. In each internal node one can arbitrarily linearize the sets of concurrent operations propagated together in a Refresh. For example  $op_1^1$  and  $op_3^2$  have propagated together in one Propagate step and they will be propagated up to the root together. Since their execution time intervals overlap, they can be linearized in any order.

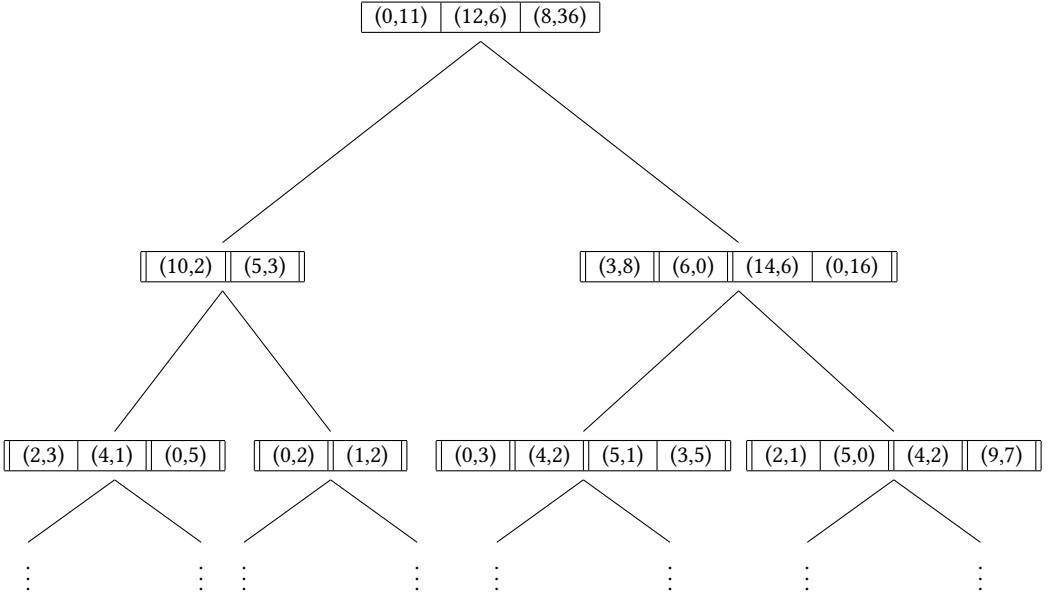


Fig. 5. Using blocks to represent operations. Blocks between two lines || are propagated together to the parent. Each block consists of a pair (left, right) indicating the number of operations from the left and the right child, respectively. For example, (12,6) in the root contains (10,2) from the left child and (6,0) from the right child. The third block in the root (8,36) is created by merging (5,3) from the left child and (14,6) and (0,16) from the right child. (5,3) is the superblock of (0,5) and (1,2) and (5,1),(3,5) and (4,2) are subblocks of (14,6).

So far, we have a shared tree that processes use to agree on the implicit ordering stored in its root. With this agreement on the linearization ordering, we can design a universal construction; for a given object, we can perform an operation  $op$  by applying all the operations up until  $op$

in the root on a local copy of the object and then returning the response for  $op$ . However, this approach is not enough for an efficient queue. We show that we can build an efficient queue if we can compute two things about the ordering in the root: (1) the  $i$ th propagated operation and (2) the rank of a propagated operation in the linearization. We explain how to implement (1) and (2) in poly-logarithmic steps.

After propagating an operation  $op$  to the root, processes can find out information about the linearization ordering using (1) and (2). To get the  $i$ th operation in the root, we find the block  $B$  containing the  $i$ th operation in the root, and then recursively find the subblock of  $B$  in the descendent of the root that contains that  $i$ th operation. When we reach a block in a leaf, the operation is explicitly stored there. To make this search faster, instead of iterating over all blocks in the node, we store the prefix sum of the number of elements in the blocks sequence to permit a binary search for the required block. We also store pointers to determine the range of subblocks of a block to make the binary search faster. In each block, we store the prefix sum of operations from the left child and from the right child. Moreover, for each block, we store two attributes  $end_{left}$  and  $end_{right}$ , the indices of the last left and right subblock (see Figure 6). We know a block size is at most  $p$ , so binary search takes at most  $O(\log p)$  time, since the  $end_{left}$  and  $end_{right}$  indices of a block and its previous block reduce the search range size to  $O(p)$ .

To compute the rank in the root of an operation in a leaf, we need to find the superblock of the block that the operation is in. After a block is installed in a node we store the approximate index of its superblock in it to make this faster.

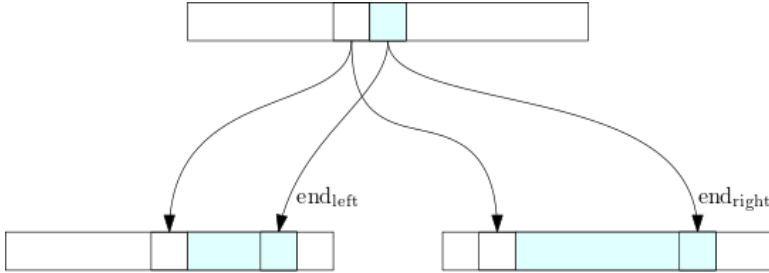


Fig. 6. Each block stores the index of its last subblock in each child.

In an execution on a queue where no dequeue operation returns null, the  $k$ th dequeue returns the argument of the  $k$ th enqueue. In the general case a dequeue returns null if and only if the queue size after the previous operation is 0. We refer to such a dequeue as a *null dequeue*. If the dequeue is the  $k$ th non-null dequeue, it returns the argument of the  $k$ th enqueue. Having the size of the queue after an operation we can compute the number of non-null dequeues from the number of enqueues before the operation. So, if we store the size of the queue after each block of operations in the root, we can compute the index of the enqueue whose argument is the response to a given dequeue in constant time.

In our case of implementing a queue, a process only needs to compute the rank of a Dequeue and get an Enqueue with a specific position. We know we can linearize operations in a block in any order; here, we choose to put Enqueue operations in a block before Dequeue operations. Consider the following operations, where operations in a cell are concurrent.

The Dequeue operations return null, 5, 2, 1, 3, 4, null, respectively. Now, we claim that by knowing the size of the queue, we can compute the rank of the required Enqueue for any non-null Dequeue. We apply this approach to blocks; if we store the size of the queue after each block of operations, we can compute the index of each Dequeue's result in  $O(1)$  steps.



Deq	Enq(5), Enq(2), Enq(1), Deq	Enq(3), Deq	Enq(4), Deq, Deq, Deq, Deq
-----	-----------------------------	-------------	----------------------------

Table 1. An execution on a queue separated by the operations in the root blocks.

	Deq	Enq(5), Enq(2), Enq(1), Deq	Enq(3), Deq	Enq(4), Deq, Deq, Deq, Deq
#Enqs	0	3	1	1
#Deqs	1	1	1	4
Size at end	0	2	2	0

Table 2. Augmented history of operation blocks on the queue.

The size of the queue after the  $b$ th block in the root could be computed as

$$\max\left(\text{size after } (b-1)\text{th block} + \#\text{Enqueues in } b\text{th block} - \#\text{Dequeues in } b\text{th block}, 0\right).$$

Moreover, the total number of non-null dequeues in blocks  $1, 2, \dots, b$  in the root is

$$\sum_{i=1}^b \#\text{Enqueues in } i\text{th block} - \text{size after } b\text{th block}.$$

Given a Dequeue is in block  $B$ , its response is the argument of the Enqueue whose rank is

$\# \text{non-null Dequeues in blocks after } b, b+1 \text{th index of Enqueues in } b\text{th block } B \text{'s Dequeue}$   
 $-\text{index of Dequeue in } B \text{'s Dequeues} \geq 0$ .

Otherwise, the response would be null.

### 3.1 Details of the Implementation

Section 3.2 gives the pseudocode for the queue implementation. It uses the following two types of objects.

**Node.** In each Node we store pointers to its parent and its left and right child, an array of Blocks called blocks and the index head of the first empty entry in blocks.

**Block.** The information stored in a Block depends on whether the Block is in an internal node or a leaf. If it is in a leaf, we use a LeafBlock which stores one operation. If a block  $B$  is in an internal node  $n$ , then it contains subblocks in the left and right children of  $n$ . The left subblocks of  $B$  are some consecutive blocks in the left child of  $n$  starting from where the left subblocks of the block prior to  $B$  ended. The right subblocks of  $B$  are defined similarly. In each block we store four essential fields that implicitly summarize which operations are in the block  $\text{sum}_{\text{enq-left}}, \text{sum}_{\text{deq-left}}, \text{sum}_{\text{enq-right}}, \text{sum}_{\text{deq-right}}$ . The  $\text{sum}_{\text{enq-left}}$  field is the total number of Enqueue operations in the blocks before the last subblock of  $B$  in the left child. The other fields' semantics are similar. The  $\text{end}_{\text{left}}$  and  $\text{end}_{\text{right}}$  field store the index of the last subblock of a block in the left and the right child, respectively. The approximate index of the superblock of non-root blocks is stored in their  $\text{super}$  field. The size field in a block in the root node stores the size of the queue after the operations in the block have been performed.

We now describe the routines used in the implementation.

**Enqueue( $e$ ).** An Enqueue operation does not return a response, so it is sufficient to propagate the Enqueue operation to the root and then use its position in the linearization for future Dequeue operations. Enqueue( $e$ ) creates a LeafBlock with  $\text{element} = e$ , sets its  $\text{sum}_{\text{enq}}$  and  $\text{sum}_{\text{deq}}$  fields and then appends it to the tree.

`Dequeue()`. `Dequeue` creates a `LeafBlock`, sets its `sumenq` and `sumdeq` fields and appends it to the tree. Then, it computes the position of the appended `Dequeue` operation in the root using `IndexDequeue` and after that finds the response of the `Dequeue` by calling `FindResponse`.

`Append(B)`. The `head` field is the index of the first empty slot in `blocks` in a `LeafBlock`. There are no multiple write accesses on `head` and `blocks` in a leaf because only the process that the leaf belongs to appends to it. `Append(B)` adds `B` to the end of the `blocks` field in the leaf, increments `head` and then calls `Propagate` on the leaf's parent. When `Propagate` terminates, it is guaranteed that the appended block is a subblock of a block in the root.

`Propagate()`. `Propagate` on node  $n$  uses the double refresh idea described earlier and invokes two `Refreshes` on  $n$  in Lines 52 and 53. Then, it invokes `Propagate` on  $n.parent$  recursively until it reaches the root.

`Refresh()` and `Advance()`. The goal of a `Refresh` on node  $n$  is to create a block of  $n$ 's children's new blocks and append it to  $n.blocks$ . The variable  $h$  is read from  $n.head$  at Line 60. The new block created by `Refresh` will be inserted into  $n.blocks[h]$ . Lines 61–66 of  $n.Refresh$  help to `Advance`  $n$ 's children. `Advance` increments the children's `head` if necessary and sets the `super` field of their most recently appended blocks. The reason behind this helping is explained later when we discuss `IndexDequeue`. After helping to `Advance` the children, a new block called `new` is created in Line 67. Then, if `new` is empty, `Refresh` returns `true` because there are no new operations to propagate, and it is unnecessary to add an empty block to the tree. Later we will use the fact that all blocks contain at least one operation. Line 70 tries to install `new`. If it was successful, all is good. If not, it means someone else has already put a block in  $n.blocks[h]$ . In this case, `Refresh` helps advance  $n.head$  to  $h+1$  and update the `super` field of  $n.blocks[h]$  at Line 71.

`CreateBlock()`.  $n.CreateBlock(h)$  is used by `Refresh` to construct a block containing new operations of  $n$ 's children. The block `new` is created in Line 80 and its fields are filled similarly for both left and right directions. The variable `indexprev` is the index of the block preceding the first subblock in the child in direction `dir` that is aggregated into `new`. Field `new.enddir` stores the index of the rightmost subblock of `new` in the child. Then `sumenq-dir` is computed from the sum of the number of `Enqueue` operations in the new block from direction `dir` and the value stored in  $n.blocks[h-1].sum_{enq-dir}$ . The field `sumdeq-dir` is computed similarly. Then, if `new` is going to be installed in the root, the `size` field is also computed.

`IndexDequeue(b, i)`. A call to  $n.IndexDequeue(b, i)$  computes the block and the rank within the block in the root of the  $i$ th `Dequeue` of the  $b$ th block of  $n$ . Let  $R_n$  be the successful `Refresh` on node  $n$  that did a successful `CAS(null, B)` into  $n.blocks[b]$ . Let  $par$  be  $n.parent$ . Without loss of generality, assume for the rest of this section that  $n$  is the left child of  $par$ . Let  $R_{par}$  be the first successful  $par.Refresh$  that reads some value greater than  $b$  for `left.head` and therefore contains  $B$  as a subblock of its created block in Line 67. Let  $j$  be the index of the block that  $R_{par}$  puts in  $par.blocks$ .

Since the index of the superblock of  $B$  is not known until  $B$  is propagated,  $R_n$  cannot set the `super` field of  $B$  while creating  $B$ . One approach for  $R_{par}$  is to set the `super` field of  $B$  after propagating  $B$  to  $par$ . This solution would not be efficient because there might be up to  $p$  subblocks that  $R_{par}$  propagated, which need to update their `super` field. However, intuitively, once  $B$  is installed, its superblock is going to be close to  $n.parent.head$  at the time of installation. If we know the approximate position of the superblock of  $B$  then we can search for the real superblock when it is needed. Thus,  $B.super$  does not have to be the exact location of the superblock of  $B$ , but we want it to be close to  $j$ . We can set  $B.super$  to  $par.head$  while creating  $B$ , but the problem is that

there might be many Refreshes on  $par$  that could happen after  $R_n$  reads  $par.head$  and before propagating  $B$  to  $par$ . If  $R_n$  sets  $B.super$  to  $par.head$  after appending  $B$  to  $n.blocks$  (Line 76),  $R_n$  might go to sleep at some time after installing  $B$  and before setting  $B.super$ . In this case, the next Refreshes on  $n$  and  $par$  help fill in the value of  $B.super$ .

Block  $B$  is appended to  $n.blocks[b]$  on Line 70. After appending  $B$ ,  $B.super$  is set on Line 76 of a call to Advance from  $n.Refresh$  by the same or another process or by Line 64 of a  $n.parent.Refresh$ . We shall show that this is sufficient to ensure that  $B.super$  differs from the index of  $B$ 's superblock by at most 1.

**FindResponse( $b, i$ ).** To compute the response of the  $i$ th Dequeue in the  $b$ th block of the root Line 19 computes whether the queue is empty or not. If there are more Dequeues than Enqueues the queue would become empty before the requested Dequeue. If the queue is not empty, Line 22 computes the rank  $e$  of the Enqueue whose argument is the response to the Dequeue. Knowing the response is the  $e$ th Enqueue in the root (which is before the  $b$ th block), we find the block and position containing the Enqueue operation using DoublingSearch and after that GetEnqueue finds its element.

**GetEnqueue( $b, i$ ) and DoublingSearch( $e, end$ ).** We can describe an operation in a node in two ways: the rank of the operation among all the operations in the node or the index of the block containing the operation in the node and the rank of the operation within that block. If we know the block and rank within the block of an operation, we can find the subblock containing the operation and the operation's rank within that subblock in poly-log time. To find the response of a Dequeue, we know about the rank of the response Enqueue in the root ( $e$  in Line 22). We also know the  $e$ th Enqueue is in  $root.blocks[1..end]$ . DoublingSearch uses doubling to find the range that contains the answer block (Lines 38–41) and then tries to find the required indices with a binary search (Line 42). A call to  $n.GetEnqueue(b, i)$  returns the element of the  $i$ th enqueue in the  $b$ th block of  $n$ . The range of subblocks of a block is determined using the  $end_{left}$  and  $end_{right}$  fields of the block and its previous block. Then, the subblock is found using binary search on the  $sum_{enq}$  field (Lines 99 and 103).

### 3.2 Pseudocode

We present our algorithm in pseudocode. page 22 contains the description of the fields in the tree nodes and the blocks. The value of any uninitialized field is null. page 23 contains major routines and the rest of this section consists of the auxiliary routines. The abbreviations below are used in the pseudocode and the proof of correctness.

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

## 4 PROOF OF CORRECTNESS

We adopt linearizability as our definition of correctness. In our case, where we create the linearization ordering in the root, we need to prove (1) the ordering is legal, i.e, for every execution on our queue if operation  $op_1$  terminates before operation  $op_2$  then  $op_1$  is linearized before operation  $op_2$  and (2) if we do operations sequentially in their linearization order, operations get the same results as in our queue. The proof is structured like this. First, we define and prove some facts about blocks and the node's head field. Then, we introduce the linearization ordering formally. Next, we prove double Refresh on a node is enough to propagate its children's new operations up to the

---

**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.
  - Node
    - \*Node left, right, parent: Initialized when creating the tree.
    - Block[] blocks: Initially blocks[0] contains an empty block with all fields equal to 0.
    - int head= 1: #blocks in blocks. blocks[0] is a block with all integer fields equal to zero.
  - Block
    - int super: approximate index of the superblock, read from parent.head when appending the block to the node
  - InternalBlock extends Block
    - int end<sub>left</sub>, end<sub>right</sub>: indices of the last subblock of the block in the left and right child
    - int sum<sub>enq-left</sub>: #enqueues in left.blocks[1..end<sub>left</sub>]
    - int sum<sub>deq-left</sub>: #dequeues in left.blocks[1..end<sub>left</sub>]
    - int sum<sub>enq-right</sub>: #enqueues in right.blocks[1..end<sub>right</sub>]
    - int sum<sub>deq-right</sub>: #dequeues in right.blocks[1..end<sub>right</sub>]
  - 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<sub>enq</sub>, sum<sub>deq</sub>: # enqueue, dequeue operations in this block and its previous blocks in the leaf
  - RootBlock extends InternalBlock
    - int size: size of the queue after performing all operations in this block and its previous blocks in the root
- 

---

**Algorithm** Queue
 

---

```

1: void Enqueue(Object e)                                ▶ Creates a block with element e and adds it to the tree.
2:   block newBlock= new(LeafBlock)
3:   newBlock.element= e
4:   newBlock.sumenq= leaf.blocks[leaf.head].sumenq+1
5:   newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq
6:   leaf.Append(newBlock)
7: end Enqueue

▶ Creates a block with null value element, appends it to the tree and returns its response.
8: Object Dequeue()
9:   block newBlock= new(LeafBlock)
10:  newBlock.element= null
11:  newBlock.sumenq= leaf.blocks[leaf.head].sumenq
12:  newBlock.sumdeq= leaf.blocks[leaf.head].sumdeq+1
13:  leaf.Append(newBlock)
14:  <b, i>= IndexDequeue(leaf.head, 1)
15:  output= FindResponse(b, i)
16:  return output
17: end Dequeue

▶ Returns the response to  $D_i(\text{root}, b)$ , the  $i$ th Dequeue in root.blocks[b].
18: element FindResponse(int b, int i)
19:   if root.blocks[b-1].size + root.blocks[b].numenq - i < 0 then    ▶ Check if the queue is empty.
20:     return null
21:   else                                                                ▶ The response is  $E_e(\text{root})$ , the  $e$ th Enqueue in the root.
22:     e= i + (root.blocks[b-1].sumenq-root.blocks[b-1].size)
23:     return root.GetEnqueue(root.DoublingSearch(e, b))
24:   end if
25: end FindResponse

```

---

---

**Algorithm Node**


---

$\leadsto$  Precondition:  $\text{blocks}[\text{start}..\text{end}]$  contains a block with  $\text{sum}_{\text{enq}}$  greater than or equal to  $x$   
 $\triangleright$  Does a binary search for the value  $x$  of  $\text{sum}_{\text{enq}}$  field and returns the index of the leftmost block in  $\text{blocks}[\text{start}..\text{end}]$  whose  $\text{sum}_{\text{enq}}$  is  $\geq x$ .

```

26: int BinarySearch(int x, int start, int end)
27:   while start < end do
28:     int mid = floor((start+end)/2)
29:     if blocks[mid].sumenq < x then
30:       start = mid+1
31:     else
32:       end = mid
33:     end if
34:   end while
35:   return start
36: end BinarySearch
  
```

---



---

**Algorithm Root**


---

$\leadsto$  Precondition:  $\text{root.blocks}[\text{end}].\text{sum}_{\text{enq}} \geq e$   
 $\triangleright$  Returns  $\langle b, i \rangle$  such that  $E_e(\text{root}) = E_i(\text{root}, b)$ , i.e., the  $e$ th Enqueue in the root is the  $i$ th Enqueue within  $\triangleright$  the  $b$ th block in the root.

```

37: <int, int> DoublingSearch(int e, int end)
38:   start = end-1
39:   while root.blocks[start].sumenq >= e do
40:     start = max(start-(end-start), 0)
41:   end while
42:   b = root.BinarySearch(e, start, end)
43:   i = e - root.blocks[b-1].sumenq
44:   return <b, i>
45: end DoublingSearch
  
```

---



---

**Algorithm Leaf**


---

```

46: void Append(block B)
47:   blocks[head] = B
48:   head = head+1
49:   parent.Propagate()
50: end Append
  
```

$\triangleright$  Only called by the owner of the leaf.

---

node, which is used to prove (1). After this, we prove some claims about the size and operations of each block, which we use to prove the correctness of `DoublingSearch()`, `GetEnqueue()` and `IndexDequeue()`. Finally, we prove the correctness of the way we compute the response of a dequeue, which establishes (2).

#### 4.1 Basic Properties

In this subsection, we talk about some properties of blocks and fields of the tree nodes.

A block is an object storing some statistics, as described in Algorithm Queue. A block in a node implicitly represents a set of operations.

**Definition 1** (Ordering of a block in a node). Let  $B$  be  $n.\text{blocks}[b]$  and  $B'$  be  $n.\text{blocks}[j]$ . We call  $i$  the *index* of block  $B$ . Block  $B$  is *before* block  $B'$  in node  $n$  if and only if  $i < j$ .

---

**Algorithm Node**


---

► *n*.Propagate propagates operations in this.children up to this when it terminates.

```

51: void Propagate()
52:   if not Refresh() then
53:     Refresh()
54:   end if
55:   if this is not root then
56:     parent.Propagate()
57:   end if
58: end Propagate

► Creates a block containing new operations of this.children, and then tries to append it to this.
59: boolean Refresh()
60:   h= head
61:   for each dir in {left, right} do
62:     hdir= dir.head
63:     if dir.blocks[hdir]!=null then
64:       dir.Advance(hdir)
65:     end if
66:   end for
67:   new= CreateBlock(h)
68:   if new.num==0 then return true
69:   end if
70:   result= blocks[h].CAS(null, new)
71:   this.Advance(h)
72:   return result
73: end Refresh

```

---

**Algorithm Node**


---

```

74: void Advance(int h)                                ► Sets blocks[h].super and increments head from h to h+1.
75:   hp= parent.head
76:   blocks[h].super.CAS(null, hp)
77:   head.CAS(h, h+1)
78: end Advance

79: Block CreateBlock(int i)                            ► Creates and returns the block to be installed in blocks[i].
80:   block new= new(InternalBlock)
81:   for each dir in {left, right} do
82:     indexprev= blocks[i-1].enddir
83:     new.enddir= dir.head-1                            ► new contains dir.blocks[blocks[i-1].enddir..dir.head-1].
84:     blockprev= dir.blocks[indexprev]
85:     blocklast= dir.blocks[new.enddir]
86:     new.sumenq-dir= blocks[i-1].sumenq-dir + blocklast.sumenq - blockprev.sumenq
87:     new.sumdeq-dir= blocks[i-1].sumdeq-dir + blocklast.sumdeq - blockprev.sumdeq
88:   end for
89:   if this is root then
90:     new.type= InternalBlock->RootBlock
91:     new.size= max(root.blocks[i-1].size + new.numenq- new.numdeq, 0)
92:   end if
93:   return new
94: end CreateBlock

```

---

---

**Algorithm Node**


---

```

    ~ Precondition: blocks[b].numenq ≥ i ≥ 1
95: element GetEnqueue(int b, int i)                                ▷ Returns the element of  $E_i(\text{this}, b)$ .
96:   if this is leaf then
97:     return blocks[b].element
98:   else if i ≤ blocks[b].numenq-left then                        ▷  $E_i(\text{this}, b)$  is in the left child of this node.
99:     subblockIndex = left.BinarySearch(i + blocks[b-1].sumenq-left, blocks[b-1].endleft + 1,
                                     blocks[b].endleft)
100:    return left.GetEnqueue(subblockIndex, i)
101:   else
102:     i = i - blocks[b].numenq-left
103:     subblockIndex = right.BinarySearch(i + blocks[b-1].sumenq-right, blocks[b-1].endright + 1,
                                     blocks[b].endright)
104:    return right.GetEnqueue(subblockIndex, i)
105:   end if
106: end GetEnqueue

    ~ Precondition: bth block of the node has propagated up to the root and blocks[b].numdeq ≥ i.
107: <int, int> IndexDequeue(int b, int i)                            ▷ Returns <x, y> if  $D_i(\text{this}, b) = D_y(\text{root}, x)$ .
108:   if this is root then
109:     return <b, i>
110:   else
111:     dir = (parent.left == n ? left : right)
112:     superblockIndex = parent.blocks[blocks[b].super].sumdeq-dir > blocks[b].sumdeq ?
                                     blocks[b].super : blocks[b].super + 1
113:   if dir is left then
114:     i += blocks[b-1].sumdeq-parent.blocks[superblockIndex-1].sumdeq-left
115:   else
116:     i += blocks[b-1].sumdeq-parent.blocks[superblockIndex-1].sumdeq-right
117:     i += parent.blocks[superblockIndex].numdeq-left
118:   end if
119:   return this.parent.IndexDequeue(superblockIndex, i)
120: end if
121: end IndexDequeue

```

---

Next, we show that the value of head in a node can only be increased. By the termination of a Refresh, head has been incremented by the process doing the Refresh or by another process.

**OBSERVATION 2.** *For each node  $n$ ,  $n.\text{head}$  is non-decreasing over time.*

**PROOF.** The claim follows trivially from the code since head is only changed by incrementing in Line 77 of Advance.  $\square$

**LEMMA 3.** *Let  $R$  be an instance of Refresh on a node  $n$  that creates a non-empty block new (new.sum  $\neq 0$ ). After  $R$  terminates,  $n.\text{head}$  is greater than the value read in Line 60 of  $R$ .*

**PROOF.** If the CAS in Line 77 is successful, then the claim holds. Otherwise,  $n.\text{head}$  has changed from the value read in Line 60. By Observation 2 this means another process has incremented  $n.\text{head}$ .  $\square$

Now we show  $n.\text{blocks}[n.\text{head}]$  is either the last block written into node  $n$  or the first empty block in  $n$ .

**Invariant 4** (headPosition). If the value of  $n.\text{head}$  is  $h$  then  $n.\text{blocks}[i] = \text{null}$  for  $i > h$  and  $n.\text{blocks}[i] \neq \text{null}$  for  $0 \leq i < h$ .

PROOF. Initially the invariant is true since  $n.\text{head} = 1$ ,  $n.\text{blocks}[0] \neq \text{null}$  and  $n.\text{blocks}[x] = \text{null}$  for every  $x > 0$ . The truth of the invariant may be affected by writing into  $n.\text{blocks}$  or incrementing  $n.\text{head}$ . We show that if the invariant holds before such a change, then it still holds after the change.

In the algorithm,  $n.\text{blocks}$  is modified only on Line 70, which updates  $n.\text{blocks}[h]$  where  $h$  is the value read from  $n.\text{head}$  in Line 60. Since the CAS in Line 70 is successful, it means  $n.\text{head}$  has not changed from  $h$  before doing the CAS: if  $n.\text{head}$  had changed before the CAS then it would be greater than  $h$  by Observation 2 and hence  $n.\text{blocks}[h] \neq \text{null}$  and by the induction hypothesis, so the CAS would fail. Writing into  $n.\text{blocks}[h]$  when  $h = n.\text{head}$  preserves the invariant, since the claim does not talk about the content of  $n.\text{blocks}[n.\text{head}]$ .

The value of  $n.\text{head}$  is modified only in Line 77 of Advance. If  $n.\text{head}$  is incremented to  $h + 1$  it is sufficient to show  $n.\text{blocks}[h] \neq \text{null}$ . Advance is called in Lines 64 and 71. For Line 64,  $n.\text{blocks}[h] \neq \text{null}$  because of the if condition in Line 63. For Line 71, Line 70 was finished before doing Line 71. Whether Line 70 is successful or not,  $n.\text{blocks}[h] \neq \text{null}$  after the  $n.\text{blocks}[h].\text{CAS}$ .  $\square$

We define the subblocks of a block recursively.

**Definition 5** (Subblock). A block is a *direct subblock* of the  $i$ th block in node  $n$  if it is in

$$n.\text{left}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1 \cdots n.\text{blocks}[i].\text{end}_{\text{left}}]$$

or in

$$n.\text{right}.\text{blocks}[n.\text{blocks}[i-1].\text{end}_{\text{right}}+1 \cdots n.\text{blocks}[i].\text{end}_{\text{right}}].$$

Block  $B$  is a *subblock* of block  $C$  if  $B$  is a direct subblock of  $C$  or a subblock of a direct subblock of  $C$ . We say block  $B$  has been *propagated* to node  $n$  if  $B$  is in  $n.\text{blocks}$  or is a subblock of a block in  $n.\text{blocks}$ .

The following lemma is used to prove that subblocks of two blocks in a node are disjoint.

LEMMA 6. If  $n.\text{blocks}[b] \neq \text{null}$  ( $b > 0$ ) then  $n.\text{blocks}[i].\text{end}_{\text{left}} \geq n.\text{blocks}[i-1].\text{end}_{\text{left}}$  and  $n.\text{blocks}[i].\text{end}_{\text{right}} \geq n.\text{blocks}[i-1].\text{end}_{\text{right}}$ .

PROOF. Consider the block  $B$  written into  $n.\text{blocks}[b]$  by CAS at Line 70. Block  $B$  is created by the  $\text{CreateBlock}(b)$  called at Line 67. Prior to this call to  $\text{CreateBlock}(b)$ ,  $n.\text{head} = b$  at Line 60, so  $n.\text{blocks}[b-1]$  is already a non-null value  $B'$  by Invariant 4. Thus, the  $\text{CreateBlock}(b-1)$  that created  $B'$  has terminated before the invocation of  $\text{CreateBlock}(b)$  that created  $B$ . The value written into  $B.\text{end}_{\text{left}}$  at Line 83 of  $\text{CreateBlock}(b)$  was one less than the value of  $n.\text{left}.\text{head}$  read at Line 83 of  $\text{CreateBlock}(b)$ . Similarly, the value in  $n.\text{blocks}[b-1].\text{end}_{\text{left}}$  was one less than the value read from  $n.\text{left}.\text{head}$  during the call to  $\text{CreateBlock}(b-1)$ . By Observation 2,  $n.\text{left}.\text{head}$  is non-decreasing, so  $B'.\text{end}_{\text{left}} \leq B.\text{end}_{\text{left}}$ . The proof for  $\text{end}_{\text{right}}$  is similar.  $\square$

LEMMA 7. The sets of subblocks of any two blocks in a node are disjoint.

PROOF. We are going to prove the lemma by contradiction. Consider the lowest node  $n$  in the tree that violates the claim. Then, subblocks of  $n.\text{blocks}[i]$  and  $n.\text{blocks}[j]$  overlap for some  $i < j$ . Since  $n$  is the lowest node in the tree violating the claim, direct subblocks of blocks of  $n.\text{blocks}[i]$  and  $n.\text{blocks}[j]$  have to overlap. Without loss of generality, assume left child subblocks of  $n.\text{blocks}[i]$  overlap with the left child subblocks of  $n.\text{blocks}[j]$ . By Lemma 6 we have  $n.\text{blocks}[i].\text{end}_{\text{left}} \leq n.\text{blocks}[j-1].\text{end}_{\text{left}}$ , so the range  $[n.\text{blocks}[i-1].\text{end}_{\text{left}}+1 \cdots n.\text{blocks}[i].\text{end}_{\text{left}}]$  cannot have overlap with the range  $[n.\text{blocks}[j-1].\text{end}_{\text{left}}+1 \cdots$



$n.\text{blocks}[j].\text{end}_{\text{left}}$ . Therefore, direct subblocks of  $n.\text{blocks}[i]$  and  $n.\text{blocks}[j]$  cannot overlap, which is in contradiction with the assumption.  $\square$

**Definition 8** (Superblock). Block  $B$  is *superblock* of block  $C$  if  $C$  is a direct subblock of  $B$ .

COROLLARY 9. *Every block has at most one superblock.*

PROOF. A block having more than one superblock contradicts Lemma 7.  $\square$

Now we can define the operations of a block using the definition of subblocks.

**Definition 10** (Operations of a block). A block  $B$  in a leaf represents an Enqueue if  $B.\text{element} \neq \text{null}$ . Otherwise, if  $B.\text{element} = \text{null}$ ,  $B$  represents a Dequeue. The set of operations of block  $B$  is the union of the operations in leaf subblocks of  $B$ . We denote the set of operations of block  $B$  by  $\text{ops}(B)$  and the union of operations of a set of blocks  $\mathcal{B}$  by  $\text{ops}(\mathcal{B})$ . We also say  $B$  contains  $op$  if  $op \in \text{ops}(B)$ .

The next lemma proves that each operation appears at most once in the blocks of a node.

LEMMA 11. *For any node  $n$ , if  $op$  is in  $n.\text{blocks}[i]$  then there is no  $j \neq i$  such that  $op$  is in  $n.\text{blocks}[j]$ .*

PROOF. We prove this claim by contradiction using Lemma 7. Assume  $op$  is in the subblocks of both  $n.\text{blocks}[i]$  and  $n.\text{blocks}[j]$ . From Lemma 7 we know that the subblocks of these blocks are different, so there are two leaf blocks containing  $op$ . Since each process puts each operation in only one block of its leaf,  $op$  cannot be in two leaf blocks. This is a contradiction.  $\square$

**Definition 12.**  $n.\text{blocks}[i]$  is *established* if  $n.\text{head} > i$ . An operation is *established* in node  $n$  if it is in an established block of  $n$ .  $EST_n^t$  is the set of established operations in node  $n$  at time  $t$ .

Now we want to say that blocks of a node grow over time.

OBSERVATION 13. *If time  $t < \text{time } t'$  ( $t$  is before  $t'$ ), then  $\text{ops}(n.\text{blocks})$  at time  $t$  is a subset of  $\text{ops}(n.\text{blocks})$  at time  $t'$ .*

PROOF. Blocks are only appended (not modified) with CAS to  $n.\text{blocks}[n.\text{head}]$ , so the set of the blocks of a node after the CAS contains the set of the blocks before the CAS.  $\square$

## 4.2 Ordering Operations

Now we define the ordering of operations stored in each node. In the non-root nodes, we only need to order operations of a type among themselves (that is, we order the Enqueues in the node and order the Dequeues in the node separately). Processes are numbered from 1 to  $p$ , and leaves of the tree are assigned from left to right. We will show in Lemma 27 that there is at most one operation from each process in a given block.

**Definition 14** (Ordering of operations inside the nodes).

- $E(n, b)$  is the sequence of enqueue operations in  $\text{ops}(n.\text{blocks}[b])$  defined recursively as follows.  $E(\text{leaf}, b)$  is the single enqueue operation in  $\text{ops}(\text{leaf}.\text{blocks}[b])$  or an empty sequence if  $\text{leaf}.\text{blocks}[b]$  represents a dequeue operation. If  $n$  is an internal node, then

$$E(n, b) = E(n.\text{left}, n.\text{blocks}[b-1].\text{end}_{\text{left}} + 1) \cdots E(n.\text{left}, n.\text{blocks}[b].\text{end}_{\text{left}}).$$

$$E(n.\text{right}, n.\text{blocks}[b-1].\text{end}_{\text{right}} + 1) \cdots E(n.\text{right}, n.\text{blocks}[b].\text{end}_{\text{right}}).$$

- $E_i(n, b)$  is the  $i$ th enqueue in  $E(n, b)$ .
- The order of the enqueue operations in the node  $n$  is  $E(n) = E(n, 1) \cdot E(n, 2) \cdot E(n, 3) \cdots$

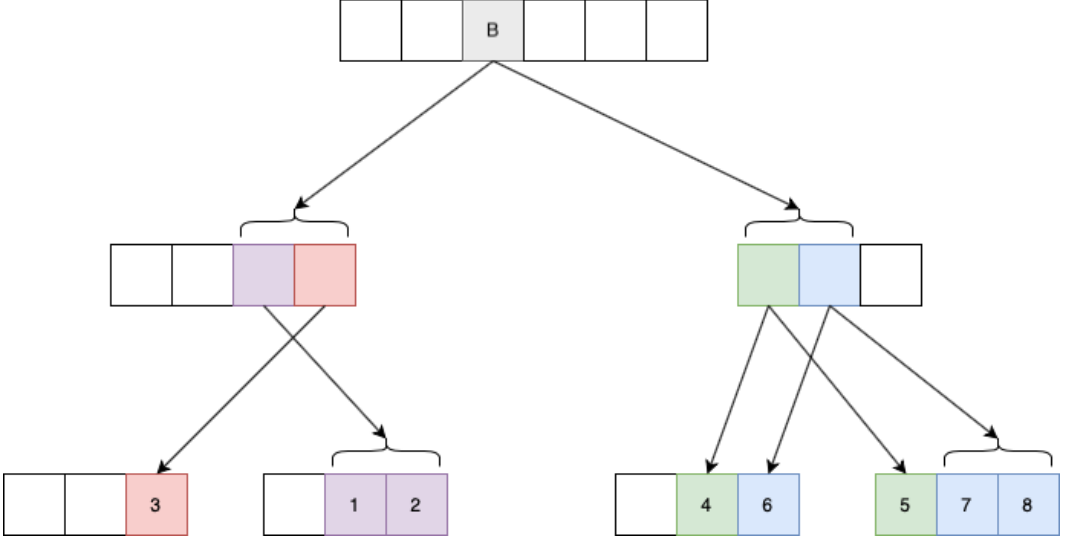


Fig. 7. Order of operations in the block  $B$ . Operations in the leaves are ordered in the numerical order shown in the drawing.

- $E_i(n)$  is the  $i$ th 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  $leaf.blocks[b]$  represents an enqueue operation. If  $n$  is an internal node, then

$$D(n, b) = D(n.left, n.blocks[b-1].end_{left} + 1) \cdots D(n.left, n.blocks[b].end_{left}) \cdot \\ D(n.right, n.blocks[b-1].end_{right} + 1) \cdots D(n.right, n.blocks[b].end_{right}).$$

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

The linearization ordering is given by the order in which operations appear in the blocks in the root.

**Definition 15** (Linearization).

$$L = E(root, 1) \cdot D(root, 1) \cdot E(root, 2) \cdot D(root, 2) \cdot E(root, 3) \cdot D(root, 3) \cdots$$

The following observation follows from the Definition of  $num_x$  on page 22.

OBSERVATION 16. For any node  $n$  and indices  $i < j$  of blocks in  $n$ , we have

$$n.blocks[j].sum_x - n.blocks[i].sum_x = \sum_{k=i+1}^j n.blocks[k].num_x$$

where  $x \in \{\text{enq}, \text{deq}, \text{enq-left}, \text{enq-right}, \text{deq-left}, \text{deq-right}\}$ .

The next claim is also valid if we replace enq with deq and  $E$  with  $D$ .

LEMMA 17. Let  $B$  and  $B'$  be  $n.blocks[b]$  and  $n.blocks[b-1]$ , respectively. If  $n$  is an internal node, then

$$(1) B.\text{num}_{\text{enq-left}} = \left| E(n.\text{left}, B'.\text{end}_{\text{left}} + 1) \cdots E(n.\text{left}, B.\text{end}_{\text{left}}) \right|.$$

$$(2) B.\text{num}_{\text{enq-right}} = \left| E(n.\text{right}, B'.\text{end}_{\text{right}} + 1) \cdots E(n.\text{right}, B.\text{end}_{\text{right}}) \right|.$$

And for every node  $n$ , we have

$$(3) B.\text{num}_{\text{enq}} = \left| E(n, b) \right|.$$

PROOF. We prove the claim by induction on the height of node  $n$ . For the base case when  $n$  is a leaf, statement (3) is trivial, and (1) and (2) are vacuously true. Supposing the claim is true for  $n$ 's children, we prove the claim for  $n$ .

$$\begin{aligned} B.\text{num}_{\text{enq-left}} &= B.\text{sum}_{\text{enq-left}} - B'.\text{sum}_{\text{enq-left}} \\ &= B'.\text{sum}_{\text{enq-left}} + n.\text{left.blocks}[B.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} \\ &\quad - n.\text{left.blocks}[B'.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} - B'.\text{sum}_{\text{enq-left}} \\ &= n.\text{left.blocks}[B.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} - n.\text{left.blocks}[B'.\text{end}_{\text{left}}].\text{sum}_{\text{enq}} \\ &= \sum_{i=B'.\text{end}_{\text{left}}+1}^{B.\text{end}_{\text{left}}} n.\text{left.blocks}[i].\text{num}_{\text{enq}} \\ &= \left| E(n.\text{left}, B'.\text{end}_{\text{left}} + 1) \cdots E(n.\text{left}, B.\text{end}_{\text{left}}) \right| \end{aligned}$$

The first line follows from the Definition of  $\text{num}_{\text{enq}}$ . The second line is similar to the way  $\text{sum}_{\text{enq-left}}$  is computed in the `CreateBlock` routine. Observation 16 implies the third line and the last line holds because of the induction hypothesis (3). (2) is similar to (1). Now we prove (3) starting from the definition of  $E(n, b)$ .

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

By (1) and (2) we have  $\left| E(n, b) \right| = B.\text{num}_{\text{enq-left}} + B.\text{num}_{\text{enq-right}} = B.\text{num}_{\text{enq}}$ .  $\square$

The next claim is also true if we replace `enq` with `deq` and  $E$  with  $D$ .

COROLLARY 18. Let  $B$  be  $n.\text{blocks}[b]$ .

$$(1) \text{ If } n \text{ is an internal node then } B.\text{sum}_{\text{enq-left}} = \left| E(n.\text{left}, 1) \cdots E(n.\text{left}, B.\text{end}_{\text{left}}) \right|.$$

$$(2) \text{ If } n \text{ is an internal node then } B.\text{sum}_{\text{enq-right}} = \left| E(n.\text{right}, 1) \cdots E(n.\text{right}, B.\text{end}_{\text{right}}) \right|.$$

$$(3) B.\text{sum}_{\text{enq}} = \left| E(n, 1) \cdot E(n, 2) \cdots E(n, b) \right|.$$

PROOF. Result (1) can be proved using the previous lemma.

$$\begin{aligned} B.\text{sum}_{\text{enq-left}} &= n.\text{blocks}[1].\text{num}_{\text{enq-left}} + \cdots + n.\text{blocks}[b].\text{num}_{\text{enq-left}} \\ &= \left| E(n.\text{left}, 1) \cdots E(n.\text{left}, n.\text{blocks}[1].\text{end}_{\text{left}}) \right| + \\ &\quad \vdots \\ &\quad + \left| E(n.\text{left}, n.\text{blocks}[b-1].\text{end}_{\text{left}}) \cdots E(n.\text{left}, n.\text{blocks}[b].\text{end}_{\text{left}}) \right| \\ &= \left| E(n.\text{left}, 1) \cdots E(n.\text{left}, B.\text{end}_{\text{left}}) \right| \end{aligned}$$

We can prove (2) and (3) the same as (1).  $\square$

### 4.3 Propagating Operations to the Root

This section explains why two Refreshes are enough to propagate a node's operations to its parent.

**Definition 19.** Let  $t^{op}$  be the time  $op$  is invoked,  $^{op}t$  be the time  $op$  terminates,  $t_l^{op}$  be the time immediately before running Line  $l$  of operation  $op$  and  $^{op}_l t$  be the time immediately after running Line  $l$  of operation  $op$ . We sometimes suppress  $op$  and write  $t_l$  or  ${}_l t$  if  $op$  is clear from the context. In the text,  $v_l$  is the value of variable  $v$  immediately after line  $l$  for the process we are talking about and  $v_t$  is the value of variable  $v$  at time  $t$ .

**Definition 20** (Successful Refresh). An instance of Refresh is *successful* if its CAS in Line 70 returns true.

In the next two results, we show that for every successful Refresh, all the operations established in the children before the Refresh are in the parent after the Refresh's successful CAS at Line 70.

**LEMMA 21.** *If  $R$  is a successful instance of  $n$ .Refresh, then we have  $EST_{n.\text{left}}^{t^R} \cup EST_{n.\text{right}}^{t^R} \subseteq ops(n.\text{blocks}_{70})$ .*

**PROOF.** We show  $EST_{n.\text{left}}^{t^R} = ops(n.\text{left.blocks}[0..n.\text{left.head}_{t^R} - 1])$   
 $\subseteq ops(n.\text{blocks}_{70}) = ops(n.\text{blocks}[0..n.\text{head}_{70}])$ .

In every node,  $\text{blocks}[0]$  is an empty block without any operations. Line 70 stores a block new in  $n$  that has  $\text{end}_{\text{left}} = n.\text{left.head}_{83} - 1$ . Therefore, by Definition 5, after the successful CAS in Line 70 we know all blocks in  $n.\text{left.blocks}[1 \dots n.\text{left.head}_{83} - 1]$  are sub-blocks of  $n.\text{blocks}[1 \dots n.\text{head}_{60}]$ . Because of Observation 2 we have  $n.\text{left.head}_{t^R} - 1 \leq n.\text{left.head}_{83} - 1$  and  $n.\text{head}_{60} \leq n.\text{head}_{70}$ . From Observation 13 the claim follows. The proof for the right child is the same.  $\square$

**COROLLARY 22.** *If  $R$  is a successful instance of  $n$ .Refresh that terminates, then we have*

$$EST_{n.\text{left}}^{t^R} \cup EST_{n.\text{right}}^{t^R} \subseteq EST_n^{Rt}.$$

**PROOF.** The left-hand side is the same as Lemma 21, so it is sufficient to show when  $R$  terminates the established blocks in  $n$  are a superset of  $n.\text{blocks}_{70}$ . Line 70 writes the block new in  $n.\text{blocks}[h]$  where  $h$  is value of  $n.\text{head}$  read at Line 60. Because of Lemma 3 we are sure that  $n.\text{head} > h$  when  $R$  terminates. So the block new appended to  $n$  at Line 70 is established at  ${}^R t$ .  $\square$

In the next lemma, we show that if two consecutive instances of Refresh by the same process on node  $n$  fail, then the blocks established in the children of  $n$  before the first Refresh are guaranteed to be in  $n$  after the second Refresh.

**LEMMA 23.** *Consider two consecutive terminating instances  $R_1, R_2$  of Refresh by a process on an internal node  $n$ . If neither  $R_1$  nor  $R_2$  is a successful Refresh, then we have  $EST_{n.\text{left}}^{t^{R_1}} \cup EST_{n.\text{right}}^{t^{R_1}} \subseteq EST_n^{R_2 t}$ .*

**PROOF.** Let  $R_1$  read  $i$  from  $n.\text{head}$  at Line 60. By Lemma 3,  $R_1$  and  $R_2$  cannot both read the same value  $i$ . By Observation 2,  $R_2$  reads a larger value of  $n.\text{head}$  than  $R_1$ .

Consider the case where  $R_1$  reads  $i$  and  $R_2$  reads  $i + 1$  from Line 60. As  $R_2$ 's CAS in Line 70 returns false, there is another successful instance  $R'_2$  of  $n$ .Refresh that has done a CAS successfully into  $n.\text{blocks}[i + 1]$  before  $R_2$  tries to CAS.  $R'_2$  creates its block new after reading the value  $i + 1$  from  $n.\text{head}$  (Line 60) and  $R_1$  reads the value  $i$  from  $n.\text{head}$ . By Observation 2 we have  ${}^{R_1} t < t^{R_1}_{60} < t^{R'_2}_{60}$  (see Figure 8). By Lemma 21 we have  $EST_{n.\text{left}}^{t^{R_1}} \cup EST_{n.\text{right}}^{t^{R_1}} \subseteq ops(n.\text{blocks}_{\frac{R'_2}{70} t})$ . Also by Lemma

3 on  $R_2$ , the value of  $n.head$  is more than  $i + 1$  after  $R_2$  terminates, so the block appended by  $R'_2$  into  $n.blocks[i]$  is established by the time  $R_2$  terminates. To summarize,  $R_1 t$  is before  $R'_2$ 's read of  $n.head$  ( $t_{60}^{R'_2}$ ), so we have  $EST_{n.left}^{t_{R_1}} \cup EST_{n.right}^{t_{R_1}} \subseteq ops(n.blocks_{t_{70}^{R'_2}})$ .  $R'_2$ 's successful CAS ( $t_{70}^{R'_2}$ ) is before  $R_2$ 's termination ( $t_{R_2}$ ), so by Lemma 3  $n.head$  has been incremented when  $R_2$  terminates and the block  $R'_2$  put into  $n$  is established by then. So we have  $ops(n.blocks_{t_{70}^{R'_2}}) \subseteq EST_n^{t_{R_2}}$ .

If  $R_2$  reads some value greater than  $i + 1$  in Line 60 it means  $n.head$  has been incremented at least two times since  $R_1 t$ . By Invariant 4, when  $n.head$  is incremented from  $i + 1$  to  $i + 2$ ,  $n.blocks[i + 1]$  is non-null. Let  $R_3$  be the Refresh on  $n$  that has put the block in  $n.blocks[i + 1]$ .  $R_3$  read  $n.head = i + 1$  at Line 60 and has put its block in  $n.blocks[i + 1]$  before  $R_2$ 's read of  $n.head$  at Line 60. So we have  $t_{60}^{R_1} <_{60}^{R_3} t <_{70}^{R_3} t <_{60}^{R_2} t <_{R_2} t$ . From Observation 13 on the operations before and after  $R_3$ 's CAS and Lemmas 21 and 3 on  $R_3$  the claim holds.  $\square$

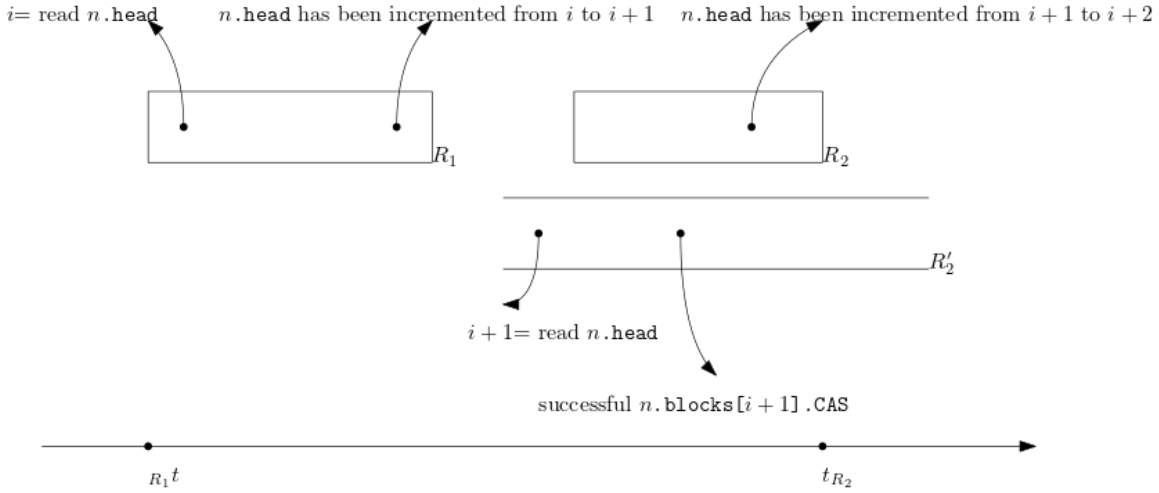


Fig. 8.  $R_1 t < t_{60}^{R_1} < \text{incrementing } n.head \text{ from } i \text{ to } i + 1 < t_{60}^{R'_2} < t_{70}^{R'_2} < \text{incrementing } n.head \text{ from } i + 1 \text{ to } i + 2 < t_{R_2}$

COROLLARY 24.  $EST_{n.left}^{t_{52}} \cup EST_{n.right}^{t_{52}} \subseteq EST_n^{t_{53}}$

PROOF. If the first Refresh in line 52 returns true, then by Corollary 22 the claim holds. If the first Refresh failed and the second Refresh succeeded, the claim still holds by Corollary 22. Otherwise, both failed, and the claim is implied by Lemma 23.  $\square$

Now we show that after  $\text{Append}(B)$  on a leaf finishes, the operation contained in  $B$  will be established in root.

COROLLARY 25. For  $A = l.$   $\text{Append}(B)$  we have  $ops(b) \subseteq EST_n^{t_A}$  for each node  $n$  in the path from  $l$  to root.

PROOF.  $A$  adds  $B$  to the assigned leaf of the process, establishes it at Line 48 and then calls  $\text{Propagate}$  on the parent of the leaf where it appended  $B$ . For every node  $n$ ,  $n.\text{Propagate}$  appends  $B$  to  $n$ , establishes it in  $n$  by Corollary 24 and then calls  $n.\text{parent}.\text{Propagate}$  until  $n$  is root.  $\square$

COROLLARY 26. *After  $l.\text{Append}(B)$  finishes,  $B$  is a subblock of exactly one block in each node along the path from  $l$  to the root.*

PROOF. By the previous corollary and Lemma 11 there is exactly one block in each node containing  $B$ .  $\square$

#### 4.4 Correctness of GetEnqueue

First, we prove some claims about the size and operations of a block. These lemmas will be used later for the correctness and analysis of GetEnqueue().

LEMMA 27. *Each block contains at most one operation of each process.*

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  $op_2$ . Process  $P$  cannot invoke more than one operation concurrently, so  $op_1$  has to be finished before  $op_2$  begins. By Corollary 26, before  $op_2$  calls Append,  $op_1$  exists in every node of the tree on the path from  $P$ 's leaf to the root. Since  $b$  contains  $op_2$ , it must be created after  $op_2$  is invoked. The fact that  $op_2.\text{Append}$  is invoked after  $op_1.\text{Append}$  terminated means that there is some block  $B'$  in  $n$  before  $B$  that contains  $op_1$ . The existence of  $op_1$  in  $B$  and  $B'$  contradicts Lemma 11.  $\square$

LEMMA 28. *Each block contains at most  $c$  operations, where  $c$  is the maximum number of concurrent operations at any time in the whole execution ( $c \leq p$ ).*

PROOF. There is a time that all the operations in a block are concurrent, because otherwise if there is an operation in a block that has ended before another operation in that block starts, then by Corollary 26 these two operations couldn't be in the same block. From the definition of  $c$  we know at any time in the execution there cannot be more than  $c$  concurrent operations, and from the previous lemma we know a process has at most one operation in a block, so there cannot be a block with more than  $c$  operations.  $\square$

LEMMA 29. *Each block has at most  $c$  direct subblocks, where  $c$  is the maximum number of concurrent operations at any time in the whole execution ( $c \leq p$ ).*

PROOF. From Definition 10 we know the operations in a block are the union of the operations in the direct subblocks of the block. We can see that each block appended to an internal node contains at least one operation due to the test on Line 68. Also, blocks in the leaves contain only one Enqueue or Dequeue operation. By Lemma 28 each block in an internal node contains at most  $c$  operations and each one of its direct subblocks has at least one operation, so by pigeonhole principle the number of direct subblocks in a block is at most  $c$ .  $\square$

DoublingSearch( $e$ ,  $end$ ) returns a pair  $\langle b, i \rangle$  such that the  $i$ th Enqueue in the  $b$ th block of the root is the  $e$ th Enqueue in the sequence stored in the root.

LEMMA 30 (DoublingSearch CORRECTNESS). *If  $1 \leq e \leq \text{root.blocks}[end].\text{sum}_{\text{enq}}$ , then DoublingSearch( $e$ ,  $end$ ) returns  $\langle b, i \rangle$  such that  $E_i(\text{root}, b) = E_e(\text{root})$ .*

PROOF. From Lines 86 and 87 we know the  $\text{sum}_{\text{enq-left}}$ , and  $\text{sum}_{\text{enq-right}}$  fields of blocks in each node are sorted in non-decreasing order. Since  $\text{sum}_{\text{enq}} = \text{sum}_{\text{enq-left}} + \text{sum}_{\text{enq-right}}$ , the  $\text{sum}_{\text{enq}}$  values of  $\text{root.blocks}[0 \cdot \text{end}]$  are also non-decreasing. By Corollary 18 we know that the  $\text{sum}_{\text{enq}}$  field in a block is the sum of the number of Enqueue operations in that block and the all blocks before that block in the node. Furthermore, since  $\text{root.blocks}[0].\text{sum}_{\text{enq}} = 0$  and  $\text{root.blocks}[end].\text{sum}_{\text{enq}} \geq e$ , there is a  $b$  such that  $\text{root.blocks}[b-1].\text{sum}_{\text{enq}} < e$  and  $e \leq \text{root.blocks}[b].\text{sum}_{\text{enq}}$ . Block  $\text{root.blocks}[b]$  contains  $E_i(\text{root}, b)$ . Lines 38-41 doubles the

search range in Line 40 and will eventually reach `start` such that  $\text{root.blocks}[\text{start}].\text{sum}_{\text{enq}} \leq e \leq \text{root.blocks}[\text{end}].\text{sum}_{\text{enq}}$ . Then, in Line 42, the binary search finds the  $b$  such that  $\text{root.blocks}[b-1].\text{sum}_{\text{enq}} < e \leq \text{root.blocks}[b].\text{sum}_{\text{enq}}$ . By Corollary 18,  $\text{root.blocks}[b]$  is the block that contains  $E_e(\text{root})$ . Finally,  $i$  is computed using the definition of  $\text{sum}_{\text{enq}}$  and Corollary 18.  $\square$

**LEMMA 31 (GetEnqueue CORRECTNESS).** *If  $1 \leq i \leq n.\text{blocks}[b].\text{num}_{\text{enq}}$  then  $n.\text{GetEnqueue}(b, i)$  returns  $E_i(n, b).\text{element}$ .*

**PROOF.** We will prove this lemma by induction on the node  $n$ 's height. For the base case, suppose  $n$  is a leaf. Leaf blocks each contain exactly one operation,  $n.\text{blocks}[b].\text{sum}_{\text{enq}} \leq 1$ , which means only  $n.\text{GetEnqueue}(b, 1)$  can be called when  $n$  is a leaf and  $n.\text{blocks}[b]$  must contain an Enqueue operation. Line 97 of  $n.\text{GetEnqueue}(b, 1)$  returns the element of the Enqueue operation stored in the  $b$ th block of leaf  $n$ , as required.

For the induction step, we prove if  $n.\text{dir}.\text{GetEnqueue}(b', i)$  returns  $E_i(n.\text{dir}, b')$  then  $n.\text{GetEnqueue}(b, i)$  returns  $E_i(n, b)$ . From Definition 14 of  $E(n, b)$ , we know that operations from the left subblocks come before the operations from the right subblocks in a block (see Figure 9). By Lemma 17, the  $\text{num}_{\text{enq-left}}$  field in  $n.\text{blocks}[b]$  is the number of Enqueue operations from the blocks' subblocks in the left child of  $n$ . So the  $i$ th Enqueue operation in  $n.\text{blocks}[b]$  is propagated from the right child if and only if  $i$  is greater than  $n.\text{blocks}[b].\text{num}_{\text{enq-left}}$ . Line 98 decides whether the  $i$ th enqueue in the  $b$ th block of internal node  $n$  is in the left child or right child subblocks of  $n.\text{blocks}[b]$ . By Definitions 5 and 10, to find an operation in the subblocks of  $n.\text{blocks}[b]$  we need to search in the range

$$\begin{aligned} & n.\text{left}.\text{blocks}[n.\text{blocks}[b-1].\text{end}_{\text{left}}+1..n.\text{blocks}[b].\text{end}_{\text{left}}] \text{ or} \\ & n.\text{right}.\text{blocks}[n.\text{blocks}[b-1].\text{end}_{\text{right}}+1..n.\text{blocks}[b].\text{end}_{\text{right}}]. \end{aligned}$$

First, we consider the case where the Enqueue we are looking for is in the left child. There are  $eb = n.\text{blocks}[b-1].\text{sum}_{\text{enq-left}}$  Enqueues in the blocks of  $n.\text{left}$  before the left subblocks of  $n.\text{blocks}[b]$ , so  $E_i(n, b)$  is  $E_{i+eb}(n.\text{left})$  which is  $E_{i'}(n.\text{left}, b')$  for some  $b'$  and  $i'$ . We can compute  $b'$  and then search for the  $i'$ th Enqueue in  $n.\text{left}.\text{blocks}[b']$ , where  $i'$  is  $i + eb - n.\text{left}.\text{blocks}[b'-1].\text{sum}_{\text{enq}}$ . The parameters in Line 99 are for searching  $E_{i+eb}(n.\text{left})$  in  $n.\text{left}.\text{blocks}$  in the range of left subblocks of  $n.\text{blocks}[b]$ , so this BinarySearch returns the index of the subblock containing  $E_i(n, b)$ .

Otherwise, the Enqueue we are looking for is in the right child. Because Enqueues from the left subblocks are ordered before the ones from the right subblocks, there are  $n.\text{blocks}[b].\text{num}_{\text{enq-left}}$  enqueues ahead of  $E_i(n, b)$  from the left child. So we need to search for  $i - n.\text{blocks}[b].\text{num}_{\text{enq-left}} + n.\text{blocks}[b-1].\text{sum}_{\text{enq-right}}$  in the right child (Line 103). Other parameters for the right child are chosen similarly to the left child.

So, in both cases, the direct subblock containing  $E_i(n, b)$  is computed in Line 99 or 103. *subblockIndex* is the index of the block in  $n.\text{dir}$  containing  $E_i(n, b)$ . Finally,  $n.\text{child}.\text{GetEnqueue}(\text{subblockIndex}, i)$  is invoked and it returns  $E_i(n, b).\text{element}$  by the hypothesis of the induction.  $\square$

#### 4.5 Correctness of IndexDequeue

The next few results show that the super field of a block is accurate within one of the actual index of the block's superblock in the parent node. Then we explain how it is used to compute the rank of a given Dequeue in the root.

**Definition 32.** If a Refresh instance  $R_1$  does its CAS at Line 70 earlier than Refresh instance  $R_2$  we say  $R_1$  has *happened before*  $R_2$ .

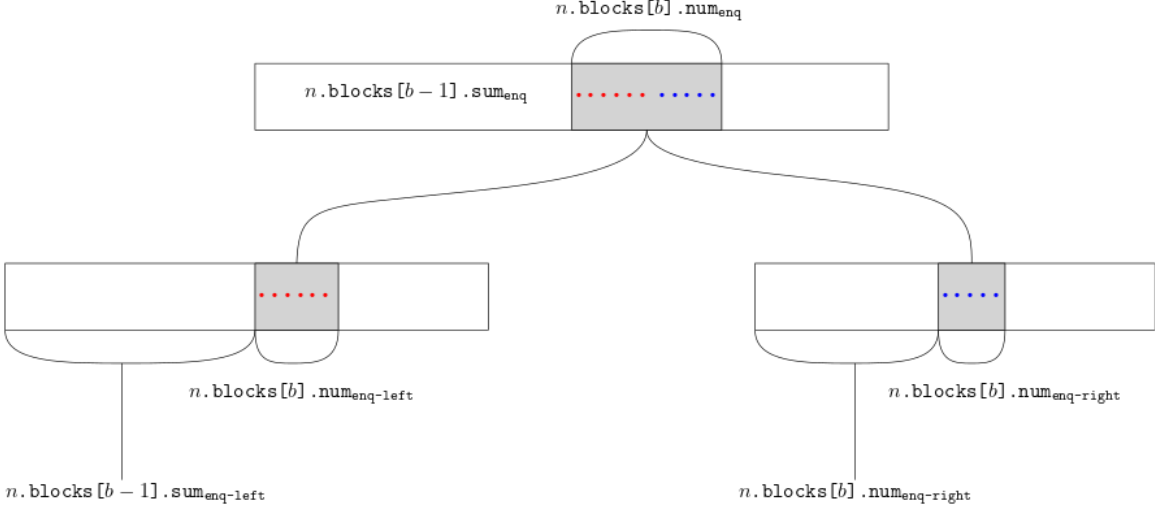


Fig. 9. The number and order of the Enqueue operations propagated from the left and the right child to  $n.blocks[b]$ . Both  $n.blocks[b]$  and its subblocks are shown in grey. Enqueue operations from the left child (colored red), are ordered before the Enqueue operations from the right child (colored blue).

**OBSERVATION 33.** *After  $n.blocks[i].CAS(\text{null}, B)$  succeeds,  $n.head$  cannot increase from  $i$  to  $i + 1$  until  $B.super$  is set.*

**PROOF.** From Observation 2 we know that  $n.head$  changes only by the increment on Line 77. Before an instance of Advance increments  $n.head$  on Line 77, Line 76 ensures that  $n.blocks[head].super$  was set at Line 76.  $\square$

**COROLLARY 34.** *If  $n.blocks[i].super$  is null, then  $n.head \leq i$  and  $n.blocks[i + 1]$  is null.*

**PROOF.** By Invariant 4 and Observation 33.  $\square$

Now let us consider how the Refreshes that took place on the parent of node  $n$  after block  $B$  was stored in  $n$  will help to set  $B.super$  and propagate  $B$  to the parent.

**OBSERVATION 35.** *If the block created by an instance  $R_p$  of  $n.parent.Refresh$  contains block  $B = n.blocks[b]$  then  $R_p$  reads a value greater than  $b$  from  $n.head$  in Line 83.*

**LEMMA 36.** *If  $B = n.blocks[b]$  is a direct subblock of  $n.parent.blocks[superblock]$  then  $B.super \leq superblock$ .*

**PROOF.** Let  $R_p$  be the instance of  $n.parent.Refresh$  that does a successful CAS (Line 70) and puts the superblock of  $B$  which is  $n.parent.blocks[superblock]$  into  $n.parent$ . By Observation 35 if  $R_p$  propagates  $B$  it has to read a greater value than  $b$  from  $n.head$ , which means  $n.head$  was incremented from  $b$  to  $b + 1$  in Line 77. By Observation 33  $B.super$  was already set in Line 76. The value written in  $B.super$ , was read in Line 75 before the CAS that sets  $B.super$  in Line 76. From Observation 2 we know  $n.parent.head$  is non-decreasing so  $B.super \leq superblock$ , since  $n.parent.head$  is still equal to  $superblock$  when  $R_p$  executes its CAS at Line 70 by Lemma 6. The reader may wonder when the case  $b.super = superblock$  happens. This can happen when  $n.parent.blocks[B.super] = \text{null}$  when  $B.super$  is written and  $R_p$  puts its created block into  $n.parent.blocks[B.super]$  afterwards.  $\square$



LEMMA 37. *Let  $R_n$  be a Refresh that puts  $B$  in  $n.\text{blocks}[b]$  at Line 70. Then, the block created by one of the next two successful  $n.\text{parent}.\text{Refreshes}$  according to Definition 32 contains  $B$  and  $B.\text{super}$  is set when the second successful  $n.\text{parent}.\text{Refresh}$  reaches Line 67.*

PROOF. Let  $R_{p_1}$  and  $R_{p_2}$  be the next two successful  $n.\text{parent}.\text{Refreshes}$  after  $R_n$ . To derive a contradiction assume  $B$  was neither propagated to  $n.\text{parent}$  by  $R_{p_1}$  nor by  $R_{p_2}$ .

Since  $R_{p_2}$ 's created block does not contain  $B$ , by Observation 35 the value  $R_{p_2}$  reads from  $n.\text{head}$  in Line 83 is at most  $b$ . From Observation 2 the value  $R_{p_2}$  reads in Line 62 is also at most  $b$ .

$R_n$  puts  $B$  into  $n.\text{blocks}[b]$  so  $R_n$  reads the value  $b$  from  $n.\text{head}$ . Since  $R_{p_2}$ 's CAS into  $n.\text{parent}.\text{blocks}$  is successful there should be a Refresh instance  $R'_p$  on  $n.\text{parent}$  that increments  $n.\text{parent}.\text{head}$  (Line 77) after  $R_{p_1}$ 's Line 70 and before  $R_{p_2}$ 's Line 60. We assumed  $t_{70}^{R_n} < t_{70}^{R_{p_1}} < t_{70}^{R_{p_2}}$  by Definition 32. Finally, Line 62 is after Line 60 and  $R_{p_2}$ 's Line 60 is after  $R'_p$ 's Line 77, which is after  $R_n$ 's  $n.\text{blocks}.\text{CAS}$ .

$$\left. \begin{array}{l} R_n t_{70} < R_{p_1} t_{70} \\ R_{p_1} t_{70} < R_{p_2} t_{60} \\ R_{p_2} t_{60} < R_{p_2} t_{62} \end{array} \right\} \Rightarrow R_n t_{70} < R_{p_2} t_{62}$$

So  $R_{p_2}$  reads a value greater than or equal to  $b$  for  $n.\text{head}$  by Observation 2.

Therefore  $R_{p_2}$  reads  $n.\text{head} = b$ .  $R_{p_2}$  calls  $n.\text{Advance}$  at Line 64, which ensures  $n.\text{head}$  is incremented from  $b$ . So the value  $R_{p_2}$  reads in Line 83 of CreateBlock is greater than  $b$  and  $R_{p_2}$ 's created block contains  $B$ . This is in contradiction with our hypothesis.

Furthermore, if  $B.\text{super}$  was not set earlier, it is set by  $R_{p_2}$ 's call to  $n.\text{Advance}$  invoked from Line 64.  $\square$

COROLLARY 38. *If  $B = n.\text{blocks}[b]$  is propagated to  $n.\text{parent}$ , then  $B.\text{super}$  is equal to or one less than the index of the superblock of  $B$ .*

PROOF. Let  $R_n$  be the  $n.\text{Refresh}$  that put  $B$  in  $n.\text{blocks}$  and let  $R_{p_1}$  be the first successful  $n.\text{parent}.\text{Refresh}$  after  $R_n$  and  $R_{p_2}$  be the second next successful  $n.\text{parent}.\text{Refresh}$ . Before  $B$  can be propagated to  $n$ 's parent,  $n.\text{head}$  must be greater than  $b$ , so by Observation 33  $B.\text{super}$  is set. From Lemma 37 we know that  $B$  is propagated by the second next successful Refresh's CAS on  $n.\text{parent}.\text{blocks}$ . To summarize, we have  $n.\text{parent}.\text{head}_{R_{p_2} t_{70}} = n.\text{parent}.\text{head}_{R_{p_1} t_{70}} + 1$  and  $n.\text{parent}.\text{head}_{R_{p_1} t_{70}} \leq n.\text{parent}.\text{head}_{R_n t_{70}}$  from Definition 32 and Observation 2. The value that is set in  $B.\text{super}$  is read from  $n.\text{parent}.\text{head}$  after  $R_n t_{70}$ . So  $B.\text{super}$  is equal to or one less than the index of the superblock of  $B$ .  $\square$

We prove IndexDequeue's correctness using Corollary 38 on each step of the IndexDequeue.

LEMMA 39 (IndexDequeue CORRECTNESS). *If  $1 \leq i \leq n.\text{blocks}[b].\text{num}_{\text{deq}}$  then  $n.\text{IndexDequeue}(b, i)$  returns  $\langle x, y \rangle$  such that  $D_i(n, b) = D_y(\text{root}, x)$ .*

PROOF. We will prove this by induction on the distance of  $n$  from the root. The base case where  $n$  is root is trivial (see Line 109). For the non-root nodes  $n$ ,  $\text{IndexDequeue}(b, i)$  computes  $\text{superblockIndex}$ , the index of the superblock of the  $b$ th block in  $n$ , in Line 112 by Corollary 38. After that, the position of  $D_i(n, b)$  in  $D(n.\text{parent}, \text{superblockIndex})$  is computed in Lines 113–118. By Definition 14, Dequeues in a block are ordered based on the order of its subblocks from left to right. If  $D_i(n, b)$  was propagated from the left child, the number of dequeues in the left subblocks of  $n.\text{parent}.\text{blocks}[\text{superblockIndex}]$  before  $n.\text{blocks}[b]$  is considered in Line 114 (see Figure 10). Otherwise, if  $D_i(n, b)$  was propagated from the right child, the number of dequeues in the subblocks from the left child is considered to be ahead of the computed index (Line 115) (see Figure

11). Finally, IndexDeque is called on  $n.parent$  recursively, and it returns the correct response by the induction hypothesis.  $\square$

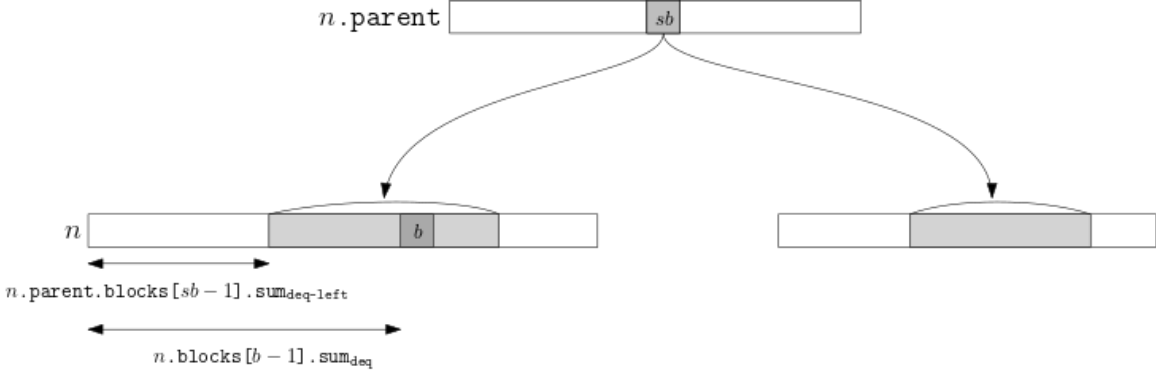


Fig. 10. The number of Dequeue operations before  $D_i(n, b)$  shown in the case where  $n$  is a left child. The index of the superbloc is shown with  $sb$ .

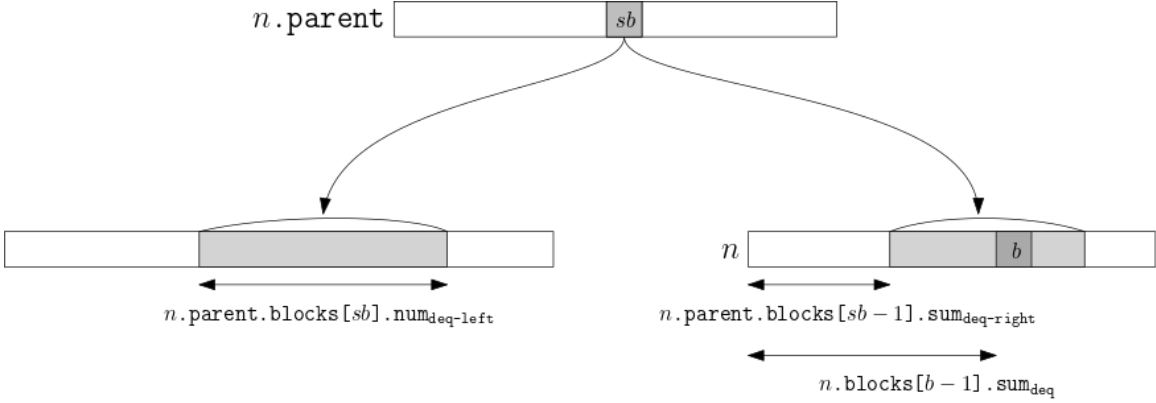


Fig. 11. The number of Dequeue operations before  $D_i(n, b)$  shown in the case where  $n$  is a right child. The index of the superbloc is shown with  $sb$ .

#### 4.6 Linearizability

We now prove the two properties needed for linearizability.

LEMMA 40.  $L$  is a legal linearization ordering.

PROOF. We must show for any execution that every operation that terminates is in  $L$  exactly once. Also, if  $op_1$  terminates before  $op_2$  starts in the execution, then  $op_1$  is before  $op_2$  in the linearization. The first claim is directly reasoned from Corollary 26. For the latter, if  $op_1$  terminates before  $op_2$  starts,  $op_1$ . Append has terminated before  $op_2$ . Append started. From Corollary 25,  $op_1$  is in  $root.blocks$  before  $op_2$  starts to propagate. By definition of  $L$ ,  $op_1$  is linearized before  $op_2$ .  $\square$

Once some operations are aggregated in one block, they will get propagated up to the root together, and they can be linearized in any order among themselves. We have chosen to put Enqueues in a block before Dequeues (see Definition 14).

**Definition 41.** If a Dequeue operation returns null it is called a *null* Dequeue, otherwise it is called *non-null* Dequeue.

Next, we define the responses that Dequeues should return, according to the linearization.

**Definition 42.** Assume the operations in  $\text{root.blocks}$  are applied sequentially on an empty queue in the order of  $L$ .  $\text{Resp}(d) = e.\text{element}$  if the element of Enqueue  $e$  is the response to Dequeue  $d$ . Otherwise if  $d$  is a null Dequeue then  $\text{Resp}(d) = \text{null}$ .

In the next lemma, we show that the size field in each root block is computed correctly.

**LEMMA 43.**  $\text{root.blocks}[b].\text{size}$  is the size of the queue after the operations in  $\text{root.blocks}[0 \dots b]$  are applied in the order of  $L$ .

**PROOF.** We prove the claim by induction on  $b$ . The base case when  $b = 0$  is trivial since the queue is initially empty and  $\text{root.blocks}[0]$  contains an empty block with size field equal to 0. We are going to show the correctness when  $b = i$  assuming correctness when  $b = i - 1$ . By Definition 14 Enqueue operations come before Dequeue operations in a block in  $L$ . By Lemma 17  $\text{num}_{\text{enq}}$  and  $\text{num}_{\text{deq}}$  fields in a block show the number of Enqueue and Dequeue operations in it. If there are more than  $\text{root.blocks}[i - 1].\text{size} + \text{root.blocks}[i].\text{num}_{\text{enq}}$  dequeue operations in  $\text{root.blocks}[i]$  then the queue would become empty after  $\text{root.blocks}[i]$ . Otherwise, the size of the queue after the  $b$ th block in the root is  $\text{root.blocks}[b - 1].\text{size} + \text{root.blocks}[b].\text{num}_{\text{enq}} - \text{root.blocks}[b].\text{num}_{\text{deq}}$ . In both cases, this is the same as the assignment on Line 91.  $\square$

The next lemma is useful to compute the number of non-null dequeues.

**LEMMA 44.** If operations in the root are applied in the order of  $L$ , the number of non-null Dequeues in  $\text{root.blocks}[0 \dots b]$  is  $\text{root.blocks}[b].\text{sum}_{\text{enq}} - \text{root.blocks}[b].\text{size}$ .

**PROOF.** There are  $\text{root.blocks}[b].\text{sum}_{\text{enq}}$  Enqueue operations in  $\text{root.blocks}[0 \dots b]$  by Corollary 18. The size of the queue after doing  $\text{root.blocks}[0 \dots b]$  in the order of  $L$  is the number of enqueues in  $\text{root.blocks}[0 \dots b]$  minus the number of non-null Dequeues in  $\text{root.blocks}[0 \dots b]$ . By the correctness of the size field from Lemma 43 and  $\text{sum}_{\text{enq}}$  field from Lemma 17, the number of non-null Dequeues is  $\text{root.blocks}[b].\text{sum}_{\text{enq}} - \text{root.blocks}[b].\text{size}$ .  $\square$

**COROLLARY 45.** If operations in the root are applied in the order of  $L$ , the number of non-null dequeues in  $\text{root.blocks}[b]$  is  $\text{root.blocks}[b].\text{num}_{\text{enq}} - \text{root.blocks}[b].\text{size} + \text{root.blocks}[b - 1].\text{size}$ .

**LEMMA 46.**  $\text{Resp}(D_i(\text{root}, b))$  is null iff  $\text{root.blocks}[b - 1].\text{size} + \text{root.blocks}[b].\text{num}_{\text{enq}} - i < 0$ .

**PROOF.** The claim follows immediately from Corollary 45 and Lemma 17.  $\square$

**LEMMA 47.**  $\text{FindResponse}(b, i)$  returns  $\text{Resp}(D_i(\text{root}, b))$ .

**PROOF.**  $D_i(\text{root}, b)$  is  $D_{\text{root.blocks}[b - 1].\text{sum}_{\text{deq}} + i}(\text{root})$  by Definition 14 and Lemma 18.  $D_i(\text{root}, b)$  returns null at Line 20 if  $\text{root.blocks}[b - 1].\text{size} + \text{root.blocks}[b].\text{num}_{\text{enq}} - i < 0$  and  $\text{Resp}(D_i(\text{root}, b)) = \text{null}$  in this case by Lemma 46. Otherwise, if  $D_i(\text{root}, b)$  is the  $i$ th non-null Dequeue in  $L$  it should return the  $i$ th enqueued value. By Lemma 44 there are  $\text{root.blocks}[b - 1].\text{sum}_{\text{enq}} - \text{root.blocks}[b - 1].\text{size}$  non-null Dequeue operations in  $\text{root.blocks}[0 \dots b - 1]$ .

The Dequeues in  $\text{root.blocks}[b]$  before  $D_i(\text{root}, b)$  are non-null Dequeues. So  $D_i(\text{root}, b)$  is the  $e$ th non-null Dequeue where  $e = i + \text{root.blocks}[b-1].\text{sum}_{\text{deq}} - \text{root.blocks}[b-1].\text{size}$  (Line 22). See Figure 12.

After computing  $e$  at Line 22, the code finds  $b, i$  such that  $E_i(\text{root}, b) = E_e(\text{root})$  using  $\text{DoublingSearch}$  and then finds its element using  $\text{GetEnqueue}$  (Line 23). Correctness of  $\text{DoublingSearch}$  and  $\text{GetEnqueue}$  routines are shown in Lemmas 30 and 31.  $\square$

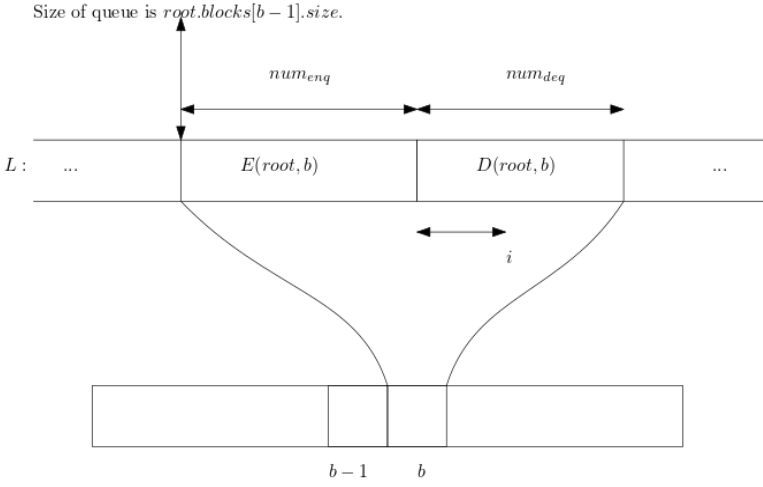


Fig. 12. The position of  $D_i(\text{root}, b)$ .

LEMMA 48. *The responses to operations in our algorithm are the same as in the sequential execution in the order given by  $L$ .*

PROOF. Enqueue operations do not return any value. By Lemma 47, the response of a Dequeue in our algorithm is the same as its response in the sequential execution of  $L$ .  $\square$

THEOREM 49 (MAIN). *The queue implementation is linearizable.*

PROOF. The theorem follows from Lemmas 40 and 48.  $\square$

*Remark.* In fact our algorithm is strongly linearizable as defined in [? ]. By Definition 14 the linearization ordering of operations will not change as blocks containing new operations are appended to the root.

## 5 ANALYSIS

In this section, we analyze the number of CAS invocations and the time complexity of our algorithm.

**Proposition 50.** An Enqueue or Dequeue operation does at most  $14 \log p$  CAS operations.

PROOF. In each level of the tree Refresh is invoked at most two times, and every Refresh invokes at most seven CASes, one in Line 70 and two from each Advance in Line 64 or 71.  $\square$

LEMMA 51 (DoublingSearch ANALYSIS). *If the element enqueued by  $E_i(\text{root}, b) = E_e(\text{root})$  is the response to some Dequeue operation in  $\text{root.blocks}[\text{end}]$ , then  $\text{DoublingSearch}(e, \text{end})$  takes  $O(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$  steps.*

PROOF. First we show  $\text{end} - b - 1 \leq 2 \times \text{root.blocks}[b - 1].\text{size} + \text{root.blocks}[\text{end}].\text{size}$ . There can be at most  $\text{root.blocks}[b].\text{size}$  Dequeues in  $\text{root.blocks}[b + 1 \dots \text{end} - 1]$ ; otherwise all elements enqueued by  $\text{root.blocks}[b]$  would be dequeued before  $\text{root.blocks}[\text{end}]$ . Furthermore, in the execution of queue operations in the linearization ordering, the size of the queue becomes  $\text{root.blocks}[\text{end}].\text{size}$  after the operations of  $\text{root.blocks}[\text{end}]$ . The final size of the queue after  $\text{root.blocks}[1 \dots \text{end}]$  is  $\text{root.blocks}[\text{end}].\text{size}$ . After an execution on a queue, the size of the queue is greater than or equal to  $\# \text{enqueues} - \# \text{dequeues}$  in the execution. We know the number of dequeues in  $\text{root.blocks}[b + 1 \dots \text{end} - 1]$  is less than  $\text{root.blocks}[b].\text{size}$ , therefore in  $\text{root.blocks}[b + 1 \dots \text{end} - 1]$  there cannot be more than  $\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}$  Enqueues. Overall there can be at most  $2 \times \text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}$  operations in  $\text{root.blocks}[b + 1 \dots \text{end} - 1]$  and since from Line 68 we know that the num field of every block in the tree is greater than 0, each block has at least one operation, so there are at most  $2 \times \text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}$  blocks in between  $\text{root.blocks}[b]$  and  $\text{root.blocks}[\text{end}]$ . So,  $\text{end} - b - 1 \leq 2 \times \text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}$ .

Thus, the doubling search reaches start such that the  $\text{root.blocks}[\text{start}].\text{sum}_{\text{enq}}$  is less than  $e$  in  $O(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$  steps. See Figure 13. After Line 41, the binary search that finds  $b$  also takes  $O(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$ . Next,  $i$  is computed via the definition of  $\text{sum}_{\text{enq}}$  in constant time (Line 43).  $\square$

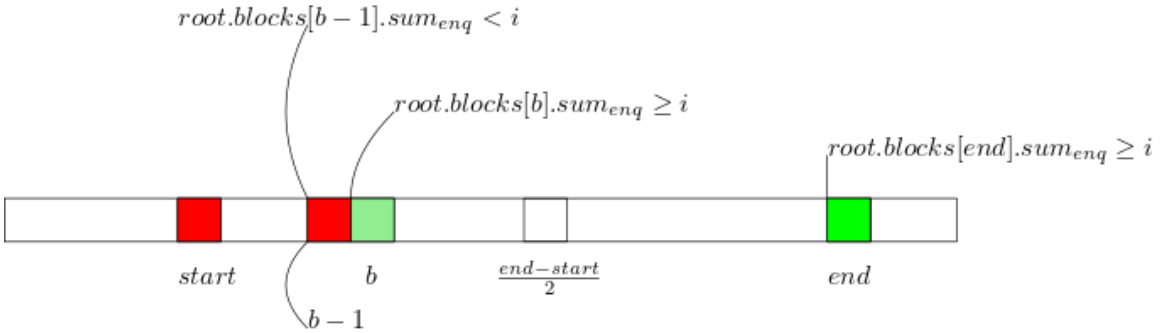


Fig. 13. Distance relations between start,  $b$ , end.

LEMMA 52 (WORST CASE TIME ANALYSIS). *The worst case number of steps for an Enqueue is  $O(\log^2 p)$  and for a Dequeue, is  $O(\log^2 p + \log q_e + \log q_d)$ , where  $q_d$  is the size of the queue when the Dequeue is linearized and  $q_e$  is the size of the queue at the time the response of the Dequeue is linearized.*

PROOF. Enqueue consists of creating a block and appending it to the tree. The first part takes constant time. To propagate the operation to the root the algorithm tries at most two Refreshes in each node of the path from the leaf to the root (Lines 52, 53). We can see from the code that each Refresh takes a constant number of steps and does  $O(1)$  CASes. Since the height of the tree is  $\Theta(\log p)$ , Enqueue takes  $O(\log p)$  steps.

A Dequeue creates a block whose element is null, appends it to the tree, computes its rank among non-null dequeues, finds the corresponding enqueue and returns the response. The first two parts are similar to an Enqueue operation and take  $O(\log p)$  steps. To compute the rank of a Dequeue in  $D(n)$ , the Dequeue calls IndexDequeue(). IndexDequeue does  $O(1)$  steps in each level which takes  $O(\log p)$  steps. If the response to the Dequeue is null, FindResponse returns null in  $O(1)$  steps. Otherwise, if the response to a dequeue in  $\text{root.blocks}[\text{end}]$  is in  $\text{root.blocks}[b]$  the DoublingSearch takes  $\Theta(\log(\text{root.blocks}[b].\text{size} + \text{root.blocks}[\text{end}].\text{size}))$  by Lemma 51, which is  $O(\log q_e + \log q_d)$ . Each search in GetEnqueue() takes  $O(\log p)$  steps since there are at most  $p$  subblocks in a block (Lemma 29), so GetEnqueue() takes  $O(\log^2 p)$  steps.  $\square$

LEMMA 53 (AMORTIZED WORST-CASE ANALYSIS). *The amortized number of steps for an Enqueue or Dequeue is  $O(\log^2 p + \log q)$ , where  $q$  is the size of the queue when the operation is linearized.*

PROOF. If we split the DoublingSearch time cost between the corresponding Enqueue and Dequeue, each operation takes  $O(\log^2 p + q)$  steps.  $\square$

OBSERVATION 54. *If the maximum number of concurrent processes at any time in an execution is  $c$ , then the amortized worst-case step complexity is  $O(\log p \log c + \log q)$  per operations. Furthermore, in a sequential, execution where  $c = 1$ , the step complexity of our algorithm is  $\Theta(\log p + \log q)$  per operation.*

PROOF. The analysis is similar to the two previous Lemmas, but by Lemma 29 each BinarySearch in each call of GetEnqueue takes  $O(\log c)$  steps.  $\square$

THEOREM 55. *The queue implementation is wait-free.*

PROOF. To prove the claim, it is sufficient to show that every Enqueue and Dequeue operation terminates after a finite number of its own steps. This is directly concluded from Lemma 52.  $\square$

## 6 FUTURE DIRECTIONS

We designed a tree to achieve agreement on a linearization of operations invoked by  $p$  processes in an asynchronous model, which we will call a *block tree*. We implemented two queries to compute information about the ordering agreed in the block tree. Then we used the tree to implement a queue where the number of steps per operation is poly-logarithmic with respect to the size of the queue and the number of processes. Block trees can be used as a mechanism to achieve agreement among processes to construct more poly-logarithmic wait-free linearizable objects. In the next paragraphs, we talk about possible improvements on block trees and the data structures that we can implement with block trees.

*Reducing Space Usage.* The blocks arrays defined in our algorithm are unbounded. To use  $O(n)$  space in each node where  $n$  is the total number of operations, instead of unbounded arrays, we could use the memory model of the wait-free vector introduced by Feldman, Valera-Leon and Damian [?]. We can create an array called `arr` of pointers to array segments (see Figure 14). When a process wishes to write into location `head` it checks whether `arr[⌊log head⌋]` points to an array or not. If not, it creates a shared array of size  $2^{\lceil \log \text{head} \rceil}$  and tries to CAS a pointer to the created array into `arr[⌊log head⌋]`. Whether the CAS is successful or not, `arr[⌊log head⌋]` points to an array. When a process wishes to access the  $i$ th element it looks up `arr[⌊log i⌋][i - 2^{\lceil \log i \rceil}]`, which takes  $O(1)$  steps. The CAS Retry Problem does not happen here because if  $n$  elements are appended to the array, then only  $O(p \times \log n)$  CAS steps have happened on the array `arr`. Furthermore, at most  $p$  arrays with size  $2^{\lceil \log i \rceil}$  are allocated by processes while processes try to do the CAS on `arr[i]`. Jayanti and Shun [?] present a way to initialize wait-free arrays in constant steps. The time taken to allocate arrays in an execution containing  $n$  operations is  $O(\frac{p \log n}{n})$  per operation, which is negligible if  $n \gg p$ . The vector implementation also has a mechanism for doubling `arr` when necessary, but this happens very rarely since increasing `arr` from  $s$  to  $2s$  increases the capacity of the vector from  $2^s$  to  $2^{2s}$ .

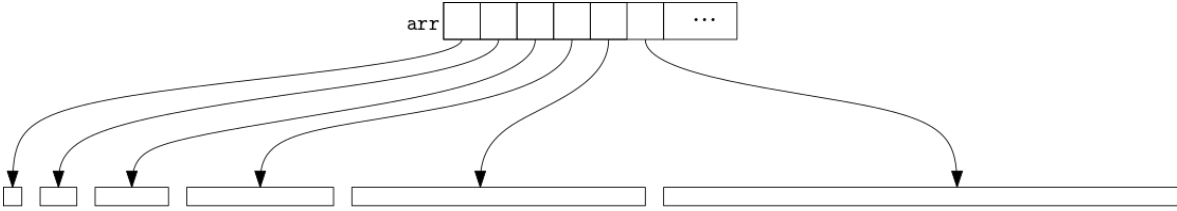


Fig. 14. Array segments.

*Garbage Collection.* We did not handle garbage collection: Enqueue operations remain in the nodes even after their elements have been dequeued. We can keep track of the blocks in the root whose operations are all terminated, i.e., all enqueues have been dequeued, and the responses of all dequeues have been computed. We call these blocks *finished blocks*. If we help the operations of all processes to compute their responses, then we can say if block  $B$  is finished, then all blocks before  $B$  are also finished. Knowing the most recent finished block in a node, we can reclaim the memory taken by finished blocks. We cannot use arrays (or vectors) to throw the garbage blocks away. We need a data structure that supports `tryAppend()`, `read(i)`, `write(i)` and `split(i)` operations in  $O(\log n)$  time, where `split(i)` removes all the indices less than  $i$ . If each process tries to do the garbage collection once every  $p^2$  operations on the queue, then the amortized complexity remains

the same. We can use a concurrent implementation of a persistent red-black trees for this [? ]. Bashari and Woelfel [? ] used persistent red-black trees in a similar way.

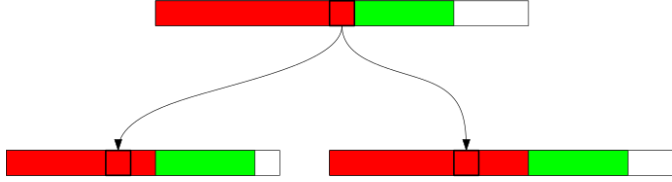


Fig. 15. Finished blocks are shown with red color and unfinished blocks are shown with green color. All the subblocks of a finished block are also finished.

*Poly-logarithmic Wait-free Sequences.* Consider a data structure storing a sequence that supports three operations `append(e)`, `get(i)` and `index(e)`. An `append(e)` adds `e` to the end of the sequence, a `get(i)` gets the  $i$ th element in the sequence and an `index(e)` computes the position of element `e` in the sequence. We can modify our queue to design such a data structure. An `append(e)` is implemented like `Enqueue(e)`, `get(i)` is done by calling `DoublingSearch` but with a `BinarySearch` on the entire `root.blocks` array and `index(e)` is done similarly to `IndexDequeue` (except operating on enqueues instead of dequeues). We achieve this with poly-logarithmic steps for each operation with respect to the number of appends done.

*Other Poly-log Wait-free Data structures.* There are two reasons the block tree worked well to implement a queue. Firstly, to respond to a `Dequeue` we do not need to look at the entire history of operations: if a `Dequeue` does not return null, we can compute the index of the `Enqueue` that is its response in  $O(\log n)$  time if we keep the number of enqueues and the size. Secondly, the operations we need to search to respond to the `Dequeue` are not very far from it in the sequence of operations: the distance is at most linear in the size of the queue. Creating a wait-free poly-logarithmic implementation of other objects whose operations satisfy these two conditions may be possible.

*Attiya–Fouren Lower Bound.* As discussed in Section 2.4 the Attiya–Fouren lower bound says that a concurrent implementation of queues using reads, writes and conditional operations like CAS has  $\Omega(c)$  amortized complexity [? ] when  $c$  the number of concurrent processes, is  $O(\log \log p)$ . Our amortized worst-case step complexity is  $\Theta(\log^2 p + \log q)$ . It is an open problem to reduce the gap between our algorithm and the Attiya–Fouren lower bound  $\Omega(\log \log p)$ .



## 7 TEMPLATE OVERVIEW

As noted in the introduction, the “acmart” document class can be used to prepare many different kinds of documentation — a double-blind initial submission of a full-length technical paper and more — all by selecting the appropriate *template style* and *template parameters*.

This document will explain the major features of the document class. For further information, the *L<sup>A</sup>T<sub>E</sub>X User’s Guide* is available from <https://www.acm.org/publications/proceedings-template>.

### 7.1 Template Styles

The primary parameter given to the “acmart” document class is the *template style* which corresponds to the kind of publication or SIG publishing the work. This parameter is enclosed in square brackets and is a part of the `\documentclass` command:

```
\documentclass[STYLE]{acmart}
```

Journals use one of three template styles. All but three ACM journals use the `acmsmall` template style:

- `acmsmall`: The default journal template style.
- `acmlarge`: Used by JOCCH and TAP.
- `acmtog`: Used by TOG.

The majority of conference proceedings documentation will use the `acmconf` template style.

- `acmconf`: The default proceedings template style.
- `sigchi`: Used for SIGCHI conference articles.
- `sigplan`: Used for SIGPLAN conference articles.

The template style used for PODC 2023 submissions is `acmsmall`.

### 7.2 Template Parameters

In addition to specifying the *template style* to be used in formatting your work, there are a number of *template parameters* which modify some part of the applied template style. A complete list of these parameters can be found in the *L<sup>A</sup>T<sub>E</sub>X User’s Guide*.

Frequently-used parameters, or combinations of parameters, include:

- `anonymous, review`: Suitable for a “double-blind” conference submission. Anonymizes the work and includes line numbers. Use with the `\printsubmissionid` command to print the submission’s unique ID on each page of the work.
- `authorversion`: Produces a version of the work suitable for posting by the author.
- `screen`: Produces colored hyperlinks.

This document uses the following string as the first command in the source file:

```
\documentclass[acmsmall,nonacm,anonymous]{acmart}
```

## 8 MODIFICATIONS

Modifying the template — including but not limited to: adjusting margins, typeface sizes, line spacing, paragraph and list definitions, and the use of the `\vspace` command to manually adjust the vertical spacing between elements of your work — is not allowed.

**Your document will be returned to you for revision if modifications are discovered.**

## 9 TYPEFACES

The “acmart” document class requires the use of the “Libertine” typeface family. Your T<sub>E</sub>X installation should include this set of packages. Please do not substitute other typefaces. The “lmodern” and “ltimes” packages should not be used, as they will override the built-in typeface families.

Table 3. Frequency of Special Characters

Non-English or Math	Frequency	Comments
Ø	1 in 1,000	For Swedish names
$\pi$	1 in 5	Common in math
\$	4 in 5	Used in business
$\Psi_1^2$	1 in 40,000	Unexplained usage

## 10 TITLE INFORMATION

The title of your work should use capital letters appropriately - <https://capitalizemytitle.com/> has useful rules for capitalization. Use the `title` command to define the title of your work. If your work has a subtitle, define it with the `subtitle` command. Do not insert line breaks in your title.

If your title is lengthy, you must define a short version to be used in the page headers, to prevent overlapping text. The `title` command has a “short title” parameter:

```
\title[short title]{full title}
```

## 11 CCS CONCEPTS AND USER-DEFINED KEYWORDS

Two elements of the “acmart” document class provide powerful taxonomic tools for you to help readers find your work in an online search.

The ACM Computing Classification System — <https://www.acm.org/publications/class-2012> — is a set of classifiers and concepts that describe the computing discipline. Authors can select entries from this classification system, via <https://dl.acm.org/ccs/ccs.cfm>, and generate the commands to be included in the  $\LaTeX$  source.

User-defined keywords are a comma-separated list of words and phrases of the authors’ choosing, providing a more flexible way of describing the research being presented.

CCS concepts and user-defined keywords are required for for all articles over two pages in length, and are optional for one- and two-page articles (or abstracts).

## 12 SECTIONING COMMANDS

Your work should use standard  $\LaTeX$  sectioning commands: `section`, `subsection`, `subsubsection`, and `paragraph`. They should be numbered; do not remove the numbering from the commands.

Simulating a sectioning command by setting the first word or words of a paragraph in boldface or italicized text is **not allowed**.

## 13 TABLES

The “acmart” document class includes the “booktabs” package — <https://ctan.org/pkg/booktabs> — for preparing high-quality tables.

Table captions are placed *above* the table.

Because tables cannot be split across pages, the best placement for them is typically the top of the page nearest their initial cite. To ensure this proper “floating” placement of tables, use the environment **table** to enclose the table’s contents and the table caption. The contents of the table itself must go in the **tabular** environment, to be aligned properly in rows and columns, with the desired horizontal and vertical rules. Again, detailed instructions on **tabular** material are found in the  $\LaTeX$  *User’s Guide*.

Immediately following this sentence is the point at which Table 3 is included in the input file; compare the placement of the table here with the table in the printed output of this document.

Table 4. Some Typical Commands

Command	A Number	Comments
<code>\author</code>	100	Author
<code>\table</code>	300	For tables
<code>\table*</code>	400	For wider tables

To set a wider table, which takes up the whole width of the page’s live area, use the environment **table\*** to enclose the table’s contents and the table caption. As with a single-column table, this wide table will “float” to a location deemed more desirable. Immediately following this sentence is the point at which Table 4 is included in the input file; again, it is instructive to compare the placement of the table here with the table in the printed output of this document.

Always use `midrule` to separate table header rows from data rows, and use it only for this purpose. This enables assistive technologies to recognise table headers and support their users in navigating tables more easily.

14 MATH EQUATIONS

You may want to display math equations in three distinct styles: inline, numbered or non-numbered display. Each of the three are discussed in the next sections.

14.1 Inline (In-text) Equations

A formula that appears in the running text is called an inline or in-text formula. It is produced by the **math** environment, which can be invoked with the usual `\begin . . . \end` construction or with the short form `$ . . . $`. You can use any of the symbols and structures, from  $\alpha$  to  $\omega$ , available in  $\text{\LaTeX}$  [?]; this section will simply show a few examples of in-text equations in context. Notice how this equation:  $\lim_{n \rightarrow \infty} x = 0$ , set here in in-line math style, looks slightly different when set in display style. (See next section).

14.2 Display Equations

A numbered display equation—one set off by vertical space from the text and centered horizontally—is produced by the **equation** environment. An unnumbered display equation is produced by the **displaymath** environment.

Again, in either environment, you can use any of the symbols and structures available in  $\text{\LaTeX}$ ; this section will just give a couple of examples of display equations in context. First, consider the equation, shown as an inline equation above:

$$\lim_{n \rightarrow \infty} x = 0$$

(1)

Notice how it is formatted somewhat differently in the **displaymath** environment. Now, we’ll enter an unnumbered equation:

$$\sum_{i=0}^{\infty} x + 1$$

and follow it with another numbered equation:

$$\sum_{i=0}^{\infty} x_i = \int_0^{\pi+2} f$$

(2)

just to demonstrate  $\text{\LaTeX}$ ’s able handling of numbering.

## 15 FIGURES

The “figure” environment should be used for figures. One or more images can be placed within a figure. If your figure contains third-party material, you must clearly identify it as such, as shown in the example below.



Fig. 16. 1907 Franklin Model D roadster. Photograph by Harris & Ewing, Inc. [Public domain], via Wikimedia Commons. (<https://goo.gl/VLCRBB>).

Your figures should contain a caption which describes the figure to the reader.

Figure captions are placed *below* the figure.

Every figure should also have a figure description unless it is purely decorative. These descriptions convey what’s in the image to someone who cannot see it. They are also used by search engine crawlers for indexing images, and when images cannot be loaded.

A figure description must be unformatted plain text less than 2000 characters long (including spaces). **Figure descriptions should not repeat the figure caption – their purpose is to capture important information that is not already provided in the caption or the main text of the paper.** For figures that convey important and complex new information, a short text description may not be adequate. More complex alternative descriptions can be placed in an appendix and referenced in a short figure description. For example, provide a data table capturing the information in a bar chart, or a structured list representing a graph. For additional information

regarding how best to write figure descriptions and why doing this is so important, please see <https://www.acm.org/publications/taps/describing-figures/>.

### 15.1 The “Teaser Figure”

A “teaser figure” is an image, or set of images in one figure, that are placed after all author and affiliation information, and before the body of the article, spanning the page. If you wish to have such a figure in your article, place the command immediately before the `\maketitle` command:

```
\begin{teaserfigure}
  \includegraphics[width=\textwidth]{sampleteaser}
  \caption{figure caption}
  \Description{figure description}
\end{teaserfigure}
```

## 16 CITATIONS AND BIBLIOGRAPHIES

The use of Bib $\TeX$  for the preparation and formatting of one’s references is strongly recommended. Authors’ names should be complete — use full first names (“Donald E. Knuth”) not initials (“D. E. Knuth”) — and the salient identifying features of a reference should be included: title, year, volume, number, pages, article DOI, etc.

The bibliography is included in your source document with these two commands, placed just before the `\end{document}` command:

```
\bibliographystyle{ACM-Reference-Format}
\bibliography{bibfile}
```

where “bibfile” is the name, without the “.bib” suffix, of the Bib $\TeX$  file.

Citations and references are numbered by default. A small number of ACM publications have citations and references formatted in the “author year” style; for these exceptions, please include this command in the **preamble** (before the command “`\begin{document}`”) of your  $\LaTeX$  source:

```
\citestyle{acmauthoryear}
```

Some examples. A paginated journal article [? ], an enumerated journal article [? ], a reference to an entire issue [? ], a monograph (whole book) [? ], a monograph/whole book in a series (see 2a in spec. document) [? ], a divisible-book such as an anthology or compilation [? ] followed by the same example, however we only output the series if the volume number is given [? ] (so Editor00a’s series should NOT be present since it has no vol. no.), a chapter in a divisible book [? ], a chapter in a divisible book in a series [? ], a multi-volume work as book [? ], a couple of articles in a proceedings (of a conference, symposium, workshop for example) (paginated proceedings article) [? ? ], a proceedings article with all possible elements [? ], an example of an enumerated proceedings article [? ], an informally published work [? ], a couple of preprints [? ? ], a doctoral dissertation [? ], a master’s thesis: [? ], an online document / world wide web resource [? ? ? ], a video game (Case 1) [? ] and (Case 2) [? ] and [? ] and (Case 3) a patent [? ], work accepted for publication [? ], ‘YYYYb’-test for prolific author [? ] and [? ]. Other cites might contain ‘duplicate’ DOI and URLs (some SIAM articles) [? ]. Boris / Barbara Beeton: multi-volume works as books [? ] and [? ]. A couple of citations with DOIs: [? ? ]. Online citations: [? ? ? ]. Artifacts: [? ] and [? ].

## 17 ACKNOWLEDGMENTS

Identification of funding sources and other support, and thanks to individuals and groups that assisted in the research and the preparation of the work should be included in an acknowledgment section, which is placed just before the reference section in your document.

This section has a special environment:

```
\begin{acks}
...
\end{acks}
```

so that the information contained therein can be more easily collected during the article metadata extraction phase, and to ensure consistency in the spelling of the section heading.

Authors should not prepare this section as a numbered or unnumbered `\section`; please use the “acks” environment.

## 18 APPENDICES

If your work needs an appendix, add it before the “`\end{document}`” command at the conclusion of your source document.

Start the appendix with the “appendix” command:

```
\appendix
```

and note that in the appendix, sections are lettered, not numbered. This document has two appendices, demonstrating the section and subsection identification method.

## 19 MULTI-LANGUAGE PAPERS

Papers may be written in languages other than English or include titles, subtitles, keywords and abstracts in different languages (as a rule, a paper in a language other than English should include an English title and an English abstract). Use `language=...` for every language used in the paper. The last language indicated is the main language of the paper. For example, a French paper with additional titles and abstracts in English and German may start with the following command

```
\documentclass[sigconf, language=english, language=german,
               language=french]{acmart}
```

The title, subtitle, keywords and abstract will be typeset in the main language of the paper. The commands `\translatedXXX`, `XXX` begin title, subtitle and keywords, can be used to set these elements in the other languages. The environment `translatedabstract` is used to set the translation of the abstract. These commands and environment have a mandatory first argument: the language of the second argument. See `sample-sigconf-i13n.tex` file for examples of their usage.

## A RESEARCH METHODS

### A.1 Part One

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi malesuada, quam in pulvinar varius, metus nunc fermentum urna, id sollicitudin purus odio sit amet enim. Aliquam ullamcorper eu ipsum vel mollis. Curabitur quis dictum nisl. Phasellus vel semper risus, et lacinia dolor. Integer ultricies commodo sem nec semper.

### A.2 Part Two

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.

## B ONLINE RESOURCES

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis

suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam interdum magna at lectus dignissim, ac dignissim lorem rhoncus. Maecenas eu arcu ac neque placerat aliquam. Nunc pulvinar massa et mattis lacinia.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009