



# A Wait-Free Deque With Polylogarithmic Step Complexity

Shalom M. Asbell  

Department of Electrical Engineering and Computer Science, York University, Toronto, Canada

Eric Ruppert   

Department of Electrical Engineering and Computer Science, York University, Toronto, Canada

---

## Abstract

The amortized step complexity of operations on all previous lock-free implementations of double-ended queues is linear in the number of processes. This paper presents the first concurrent double-ended queue where the amortized step complexity of each operation is polylogarithmic. Since a stack is a special case of a double-ended queue, this is also the first concurrent stack with polylogarithmic step complexity. The implementation is wait-free and the amortized step complexity is  $O(\log^2 p + \log q)$  per operation, where  $p$  is the number of processes and  $q$  is the size of the double-ended queue.

**2012 ACM Subject Classification** Theory of computation → Data structures design and analysis; Theory of computation → Concurrent algorithms

**Keywords and phrases** Lock-Free, Wait-Free, Double-Ended Queue, Deque, Stack, Space-Bounded, Polylogarithmic, Linearizable

**Digital Object Identifier** 10.4230/LIPIcs.OPODIS.2023.17

**Funding** This research was funded by the Natural Sciences and Engineering Research Council of Canada and a Lassonde Undergraduate Research Award.

**Acknowledgements** We thank the anonymous reviewers for their detailed comments. This research was funded by the Natural Sciences and Engineering Research Council of Canada and a Lassonde Undergraduate Research Award.

## 1 Introduction

This paper describes a linearizable, wait-free implementation of a double-ended queue (deque) [20], which allows items to be added or removed at both ends. As an example of the concurrent deque’s ubiquity, the class `ConcurrentLinkedDeque` is part of the Java standard library. Unlike previous lock-free deques, the amortized number of steps per operation in ours is polylogarithmic. To achieve this, we build on the recent wait-free queue of Naderibeni and Ruppert (N&R) [26], which was the first lock-free queue with polylogarithmic step complexity. All prior lock-free queues shared by  $p$  processes had  $\Omega(p)$  amortized step complexity due to what Morrison and Afek called the *CAS retry problem* [25], which occurs when all contending update operations must repeatedly perform compare-and-swap (CAS) instructions on one location until they succeed. A successful CAS may thwart an attempt of each other process, yielding the  $\Omega(p)$  amortized bound. Moreover, operations may starve.

To avoid the CAS retry problem and achieve polylogarithmic step complexity, the N&R queue uses a helping technique that was introduced by Afek, Dauber and Touitou [1] and used in Jayanti and Petrovic’s single-dequeue queue [18]. Each process is assigned a leaf in a static binary tree called an *ordering tree*. A process inserts its operation  $op$  into its leaf and then propagates  $op$  to the tree’s root. At each node along the path to the root,  $op$  collects non-propagated operations from the node’s children and attempts to CAS the *set* of all these operations into the node. If  $op$  fails its CAS on a node twice, some other process must have



© Shalom M. Asbell and Eric Ruppert;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Principles of Distributed Systems (OPODIS 2023).

Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, and Yukiko Yamauchi; Article No. 17; pp. 17:1–17:22



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

propagated  $op$  to this node, so  $op$  can continue up the tree. Operations are linearized in the order they reach the tree's root. A key innovation of the N&R queue is their implicit representation of sets of operations designed so that sets can be efficiently collected and propagated, while still supporting efficient retrieval of each dequeue's result.

Since it is unclear that the implicit representation of sets of operations in the N&R queue can be extended to deques (or stacks), we design a very different explicit representation that stores the sets of operations in balanced search trees. In addition to allowing us to gather and propagate sets of operations up the tree quickly, our new representation allows us to apply these batches of operations when they reach the root to obtain an explicit representation of the current state of the deque. As a byproduct, we also get an explicit representation of the results of all dequeue operations in a batch of operations. This contrasts with the implicit representation used by the N&R queue, where the result of a dequeue had to be reconstructed by searching through the history of all operations applied to the queue.

Using our new representation, we obtain a wait-free, linearizable deque for  $p$  processes with amortized step complexity  $O(\log^2 p + \log q)$  per operation, where  $q$  is the number of items in the deque. This matches the step complexity of the more restricted queue of N&R and is the first time sublinear amortized step complexity has been obtained for a lock-free deque (or stack, which is a restricted form of a deque). We unlink unneeded objects from our data structure to ensure it does not grow too large, which would slow operations down. However, we leave the orthogonal problem of reclaiming memory of unlinked items to a garbage collector, such as the highly optimized one provided by Java. Our ordering tree data structure uses  $O(q + p^2 \log p)$  space, improving on the N&R queue by a factor of  $p$ .

## 2 Related Work

**Dequeues.** The deque is a classical data structure [20], and lock-free implementations of deques have been studied for decades. Although it is a fairly simple data structure, implementing it in a concurrent setting can be tricky. For example, Koval et al. [22] recently reported a linearizability bug in the `ConcurrentLinkedDeque` of the Java standard library. Early lock-free deques [2, 6, 10, 23] used double-word CAS instructions, which are generally not provided by hardware. Michael [24] introduced the first lock-free deque based on single-word CAS. It uses a doubly-linked list. Each operation uses a CAS to update an *Anchor* object, which has pointers to both ends of the list. If the CAS fails, the operation tries again. This CAS retry problem yields  $\Omega(p)$  amortized step complexity per operation, where  $p$  is the number of processes. Ideally, operations on opposite ends of the deque should not interfere with each other, but they do in the Michael deque since they are serialized by their accesses to the Anchor. Sundell and Tsigas [28] gave a lock-free deque, also based on a doubly-linked list, where operations at opposite ends of the deque do not interfere with each other, unless the deque is empty or nearly empty. This deque also suffers from the CAS retry problem.

Herlihy, Luchangco and Moir [16] gave an array-based deque satisfying the weaker progress condition of obstruction-freedom, where operations terminate if they run without interference from other processes for sufficiently long. Graichen, Izraelevitz and Scott [9] described how to use a doubly-linked list of these array-based deques to get an obstruction-free deque with unlimited capacity. Both implementations have unbounded amortized step complexity.

**Restricted Deques.** Both FIFO queues and LIFO stacks are restricted versions of deques: each provides only two of the four deque operations. Naderibeni and Ruppert [26] survey the extensive previous literature on lock-free queues, all having amortized step complexity  $\Omega(p)$ , before providing their wait-free queue with polylogarithmic step complexity. An

unbounded-space version of their queue achieves a step complexity of  $O(\log p)$  per enqueue and  $O(\log^2 p + \log q)$  per dequeue, where  $q$  is the size of the queue. A space-bounded version of their queue has amortized step complexity of  $O(\log p \log(p + q))$  per operation.

Treiber [30] described a lock-free stack based on a singly-linked list with a pointer to the top element. Pushes and pops do a CAS on this pointer and retry if the CAS fails. It has  $\Omega(p)$  amortized step complexity due to the CAS retry problem. Array-based stacks (e.g., [27]) also suffer from the CAS retry problem. Hendler, Shavit and Yerushalmi [14] used *elimination* to improve the performance of stacks: a concurrent push and pop can eliminate each other, with the pop returning the argument of the push. Operations access a traditional list- or array-based stack only if they fail to find a partner operation to eliminate. However, if pushes are only concurrent with other pushes, accesses to that underlying stack still take  $\Omega(p)$  steps. Dodds, Haas and Kirsch’s [7] lock-free stack assigns timestamps to pushed elements. Timestamps provide a partial order on the elements: elements added by concurrent pushes need not be ordered. Each process maintains a list of elements it has pushed, and a pop must check each of these lists to find an element with the youngest timestamp, which requires  $\Omega(p)$  steps in the worst case. Pushes are artificially slowed down to make elimination more likely, thereby improving performance. The authors also sketch how a similar approach could be used for a queue or deque (in which dequeue operations would also require  $\Omega(p)$  steps). Haas’s thesis [12] gives pseudocode for the queue and deque, but says the linearization proof for the deque “is still a work in progress”.

There are also lock-free dequeues where some operations are restricted to certain processes. For example, Arora, Blumofe and Plaxton [3] designed a lock-free deque for their work-stealing algorithm. In their deque, only one process can access one end, and only dequeues can be performed at the other end. It uses a fixed-size array and therefore has bounded capacity. Hendler et al. [13] used a doubly-linked list of arrays to remove this limitation. In both dequeues, the CAS retry problem occurs at the end of the queue that supports concurrent accesses: one successful dequeue at that end can cause all concurrent dequeues to fail.

**Universal Constructions.** A *universal construction* [15] is a general technique for constructing a lock-free implementation of any data structure. However, the resulting implementation is typically much less efficient than hand-crafted ones. Afek, Dauber and Touitou’s universal construction [1] can achieve  $O(\log p)$  steps per operation, as observed by Jayanti [17], but only if memory words can store  $\Omega(p \log p)$  bits. With more reasonably sized  $O(\log p)$ -bit words, the universal construction would take  $\Omega(p \log p)$  steps per operation. Nevertheless, the technique introduced in this construction forms the basis of our deque design.

**Lower Bounds.** Attiya and Fouren [4] gave lower bounds in terms of contention  $c$ , the number of operations that run concurrently. They showed the amortized step complexity of a lock-free stack or queue must be  $\Omega(\min(c, \log \log p))$ . Jayanti, Tarjan and Boix-Adserà [19] showed the amortized step complexity of any stack or queue implementation is  $\Omega(\log p)$ .

**Red-Black Trees.** Our algorithm uses a classical search tree data structure. A *red-black tree* (RBT) [11] is a balanced binary search tree that can store a set of items sorted by key values. Insertions, deletions and searches on a RBT of  $n$  items can be done in  $O(\log n)$  time. In addition, we make use of  $O(\log n)$ -time algorithms for splitting and joining RBTs [29]. As in an order-statistic tree [5], an RBT can be augmented with a count of the number of nodes in the subtree rooted at that node, so that we can also select the  $i$ th element in an in-order traversal of the tree, or split the tree at that  $i$ th element, in  $O(\log n)$  time. We assume a purely functional implementation of RBTs for a single process. In other words, we

## 17:4 A Wait-Free Deque with Polylogarithmic Step Complexity

assume update operations do not destroy the original version of the RBT, but rather create a new, modified version. This persistence can be achieved through *path copying* [8]: each RBT operation makes a copy of any node it visits, and modifies that copy without overwriting the tree's original state. In our deque algorithm, a process that wishes to update a shared RBT creates a new copy of the RBT using path copying and then swings a pointer from the old root to the new root. Processes essentially get a snapshot of the RBT simply by reading the pointer to the root, since all RBT nodes are immutable. Other persistent search trees (such as B-trees or AVL trees) could be used in place of RBTs.

### 3 Implementation

Our deque uses an *ordering tree*, which is a static binary tree of height  $\lceil \log_2 p \rceil$  with one leaf assigned to each process. Each node stores a sequence of *Blocks*; each Block represents a set of concurrent deque operations. The four deque operations `TopEnqueue`, `TopDequeue`, `BotEnqueue` and `BotDequeue` allow items to be enqueued or dequeued from either end of the deque, which we call top and bottom to avoid confusion with the right and left directions used to describe children in the ordering tree. To perform an operation on the deque, a process inserts a Block containing just that operation into its own leaf, and then ensures that the operation is propagated up to the root. Processes help propagate one another's operations along with their own. Operations are linearized when they reach the root. To propagate its operation, a process performs a *double refresh* at each node along the path from its leaf to the root. A refresh at a node  $v$  creates a Block that contains any unpropagated operations in  $v$ 's children and attempts to insert the Block into  $v$  using a CAS instruction. If the process fails this CAS twice, then it is guaranteed that some other process has propagated its operation to  $v$ . So far, this overall approach is similar to previous constructions [1, 18, 26].

In the N&R queue [26], each internal node does not have an *explicit* representation of the operations that have reached the node. More specifically, an item that is enqueued is stored in the leaf of the process that enqueued it, but is not stored in any of the internal nodes. Instead, each Block in an internal node stores some metadata that serves as an implicit representation of its set of operations. This metadata is used to navigate to an operation's Block in each node along the root-to-leaf path. Processes can compute the response for each dequeue by navigating these paths (both up and down the tree). Because each Block can contain up to  $p$  operations, Naderibeni and Ruppert claimed that building an explicit representation of the operations in Blocks of internal nodes would be too costly to achieve polylogarithmic step complexity. However, our new representation shows that, in fact, we can create such explicit representations without sacrificing the polylogarithmic running time. Moreover, this explicit representation is essential for our deque. In a queue, the FIFO ordering makes it easy for a dequeue to find its response using very little information about the sequence of operations: the  $i$ th (non-empty) dequeue returns the argument of the  $i$ th enqueue. Since a deque (or stack) permits LIFO access, matching dequeues to enqueues is less straightforward. The explicit state of our deque allows us to determine this matching.

The key idea is to use *red-black trees* (RBTs) to represent the state of the deque and batches of items to be enqueued into the deque. We use *persistent* RBTs, meaning that update operations are non-destructive because they create a new copies of RBT nodes and modify those copies, leaving the original nodes unchanged. Since RBT nodes are immutable, reading a pointer to an RBT's root gives us a snapshot of the RBT. At the root of the ordering tree, we store a *state* RBT whose in-order traversal gives the items in the implemented deque from bottom to top. In each node  $v$  of the ordering tree, we store an RBT called *topEnqs*

whose in-order traversal is the sequence of values enqueued by all `TopEnqueues` that have been propagated to the node, in the order they reached  $v$ . Another such RBT `botEnqs` is used for `BotEnqueues`. This use of RBTs allows us to perform five key actions efficiently:

1. gather the batch of items enqueued by several consecutive `Blocks` of a node  $v$  by splitting off the required number of most recently added items from  $v$ 's `topEnqs` or `botEnqs` RBT,
2. add such a batch to  $v$ 's parent's `topEnqs` RBT (or `botEnqs` RBT) during a refresh by joining two RBTs,
3. apply a batch of enqueues to the state of the deque when they reach the root by joining the `state` RBT with the RBT representing the batch of enqueued items,
4. apply a batch of dequeues to the state of the deque when they reach the root by splitting the the required number of items off one side of the `state` RBT, and
5. discard old information when it is no longer needed (again by splitting an RBT) to avoid having RBTs that grow too large, which in turn would slow down operations on RBTs.

We linearize operations on our deque based on the order in which they were propagated to the root and applied to `state`. Because other operations may help propagate a dequeue to the root, a dequeue has to retrace the path it took from its leaf to the root to retrieve its response. When dequeues are linearized, RBTs of the items split off of the `state` RBT are saved in the root for later retrieval by those dequeues.

The `Blocks` of a node are also stored in an RBT called `blocks`. This makes it easy to garbage-collect old `Blocks` that are no longer needed by splitting the RBT. Each `Block` stores some metadata to describe its batch of operations. For example, this allows us to find the portion of the `topEnqs` tree that corresponds to `TopEnqueues` within a `Block`. When a `Block` is added to a node's `blocks` RBT, we want to ensure that the items of all enqueue operations represented by that `Block` are simultaneously added to the `topEnqs` and `botEnqs` RBTs of the node. For this reason, we actually store pointers to the roots of all of the node's RBTs in an object that we call the `rbts` tuple of the node. Thus, we can update all of a node's RBTs atomically by writing a pointer to this tuple in the node's `rbts` field. Likewise, we can get a consistent snapshot of all the node's RBTs by simply reading the node's `rbts` field.

### 3.1 Data Structure Description

We now describe the objects of our data structure in more detail. Nodes in our static *ordering tree* contain the following fields.

- `parent` – a pointer to the node's parent node.
- `left` – a pointer to the node's left child.
- `right` – a pointer to the node's right child.
- `rbts` – a pointer to a tuple of (pointers to the roots of) RBTs, described below.

In non-root nodes, the `rbts` tuple has the following five fields.

- `blocks` – an RBT composed of `Block` objects that each represent a set of concurrent operations. This tree is sorted by the order the `Blocks` were added to the node. Initially, `blocks` contains a single dummy `Block` whose fields are all 0.
- `topEnqs` and `botEnqs` – RBTs that store items to be enqueued at the top and bottom ends, respectively, of the deque. These trees are initially empty. When a `Block` of operations is added to `blocks`, the enqueued items of those operations are added to the right end of these enqueue trees. Thus, the items added to the enqueue trees are all items enqueued by operations in `Blocks` that have been added to the `blocks` RBT, in the order the operations were added to `blocks`. Garbage collection will remove from the left end of these enqueue trees those items that are no longer needed because they have been propagated up to the enqueue trees in the parent node.

## 17:6 A Wait-Free Deque with Polylogarithmic Step Complexity

- $discarded_{topEnqs}$  and  $discarded_{botEnqs}$  – integers that store the number of items that have been discarded from the left side of the  $topEnqs$  and  $botEnqs$  RBTs, respectively, by garbage collection.

In the root, the  $rbts$  tuple has the following two fields.

- $blocks$  – defined the same way as for non-root nodes.
- $state$  – an RBT storing the items in the deque after all the blocks of operations that have reached the root have been performed. The in-order traversal of the  $state$  RBT gives the order of the items from the bottom of the deque to the top. It is initially empty.

Each Block object in a node's  $blocks$  RBT has the following five fields.

- $index$  – the number of Blocks that were added to the node's  $blocks$  RBT before this one.
- $sum_{topEnqs}$ ,  $sum_{botEnqs}$ ,  $sum_{topDeqs}$  and  $sum_{botDeqs}$  – the total number of operations of each type that have been propagated to the node up to and including the current Block. Blocks in internal nodes have the following two additional fields.

- $end_{left}$  – index of the last Block propagated from the node's left child into this Block.
- $end_{right}$  – index of the last Block propagated from the node's right child into this Block.

Each Block object in the root has the following two additional fields.

- $topDeqs$  and  $botDeqs$  – RBTs containing the responses to be returned to the **TopDequeues** or **BotDequeues**, respectively, contained in the Block. The in-order traversals of  $topDeqs$  and  $botDeqs$  in a root Block  $B$  gives the order in which elements were dequeued from the top and bottom of the deque respectively.

Each Block in a leaf node has the following additional field.

- $response$  – the response to a leaf Block's operation, if it is dequeue.

Most fields of objects are *immutable*, except for the  $rbts$  field of nodes and the  $response$  field in leaf Blocks. That is, after a process creates an object, it sets the values of all its immutable fields before writing a pointer to that object in shared memory, and those fields' values never change thereafter. Even the persistent red-black trees within our data structure have immutable nodes, as described above. This simplifies reasoning about concurrency in our algorithm, because a process can essentially obtain a snapshot of the  $rbts$  field of a node  $v$  and all of the information contained within it, including the information inside  $v$ 's Blocks, simply by reading  $v.rbts$ . Thus, when proving correctness, we can often focus on the few lines of code where nodes'  $rbts$  fields are read or updated.

We assume RBTs have **Split** and **Join** operations [29].  $\text{Join}(\text{RBT } l, \text{RBT } r)$  returns an RBT whose in-order traversal contains the elements of  $l$ 's in-order traversal followed by  $r$ 's in-order traversal. The inverse operation  $\text{Split}(\text{RBT } t, \text{int } count)$  returns a pair of RBTs  $\langle l, r \rangle$  such that  $\text{Join}(l, r)$  would yield  $t$ , and where  $r$  contains  $count$  objects (or  $t.size$  items if  $count > t.size$ ) and  $l$  has the rest. The enqueue and dequeue RBTs have the additional operations **Append** and **SearchByRank**.  $\text{Append}(\text{RBT } t, \text{Object } e)$  returns the RBT that is produced by inserting  $e$  into  $t$  as the rightmost element.  $\text{SearchByRank}(\text{RBT } t, \text{int } r)$  returns the  $r$ th item in the in-order traversal of  $t$ . The  $blocks$  RBTs are sorted by the  $index$  field of the Blocks and also have **Insert**, **MaxBlock** and **SplitByIndex** operations.  $\text{Insert}(\text{RBT } t, \text{Block } B)$  inserts  $B$  into  $t$ .  $\text{MaxBlock}(\text{RBT } t)$  returns the Block with the highest index in  $t$ . Similar to a **Split** operation,  $\text{SplitByIndex}(\text{RBT } blocks, \text{int } i)$  returns a pair of RBTs  $\langle l, r \rangle$ , where  $l$  contains the Blocks with indices less than  $i$  and  $r$  contains the rest.

### 3.2 Pseudocode Description

Pseudocode for our implementation appears in Algorithms 1 to 3. We omit **BotEnqueue**, **BotDequeue**, **GetBotEnqs**, **CompleteBotDeq** and **IndexBotDeq**, which are identical to the corresponding routines for the top end of the deque, except for replacing all occurrences

of *top* by *bot*, and making the changes noted in comments on lines 6, 17 and 87. We use  $blocks[i]$  as a shorthand for the Block with index  $i$  that was inserted into the *blocks* RBT: when the code reads  $blocks[i]$  (e.g., on line 71), it does a BST search for *index*  $i$  in *blocks*.

To enqueue an item  $e$  at the top end of the deque, a process calls **TopEnqueue**( $e$ ). It creates a new Block  $B$  to represent the **TopEnqueue** in the process's leaf.  $B$ 's *index* is one higher than the previous Block, and its  $sum_{topEnq}$  is one higher than the previous Block. The call to **GC** at line 7 makes a copy  $rbts_{new}$  of the  $rbts$  field of the leaf and may perform garbage collection (described in Section 3.4) to discard obsolete information. Line 8 inserts  $B$  into  $rbts_{new}.blocks$  and line 9 appends  $e$  to the right end of  $rbts_{new}.topEnqs$ . Then, line 10 writes  $rbts_{new}$  into the process's leaf. Finally, line 11 calls **Propagate** to propagate the new operation to the root, thereby ensuring that it is linearized.

A **TopDequeue** follows a similar pattern. It creates a new Block  $B$  whose *index* and  $sum_{topDeqs}$  fields are one higher than the previous Block in the process's leaf. It creates a new copy  $rbts_{new}$  of the leaf's  $rbts$  field, possibly discarding unneeded information in **GC**, appends  $B$  to  $rbts_{new}.blocks$  (lines 13–19), and writes  $rbts_{new}$  into the process's leaf (line 20). **TopDequeue** then calls **Propagate** at line 21 to propagate the new operation upward, and finally calls **CompleteTopDeq** (described below) at line 22 to find the **TopDequeue**'s response. This **CompleteTopDeq** may return *null*, but if this happens then another process helping the **TopDequeue** must have written the **TopDequeue**'s response into  $B.response$ .

The **Propagate** method recursively propagates operations from leaves of the *ordering tree* to its root. **Propagate** calls **Refresh** at most twice on each internal node  $v$  along the path from the process's leaf to the root. As in previous work [1, 18, 26], this suffices to ensure that a Block containing the operation is added to the root.

To propagate the non-propagated Blocks in  $v$ 's children along with their enqueued items to  $v$ , an operation calls **Refresh**( $v$ ). **Refresh**( $v$ ) constructs a new  $rbts$  tuple for  $v$  that includes the non-propagated Blocks of  $v$ 's children and their enqueued items, and attempts to CAS the tuple into  $v.rbts$  (line 51). To do this, it first reads snapshots of  $v.rbts$ ,  $v.left.rbts$  and  $v.right.rbts$  into  $rbts_{old}$ ,  $rbts_{left}$  and  $rbts_{right}$  (lines 28–30). Line 31 ensures that **Refresh** has a consistent view of  $v$  and its children; otherwise the **Refresh** aborts and returns false. Line 34 calls **CreateBlock** (described below) to construct a Block  $B_{new}$  with an index one higher than its preceding Block in  $v$ ,  $B_{prev}$ .  $B_{new}$  contains the metadata of the non-propagated operations from  $rbts_{left}$  and  $rbts_{right}$ . If there are no non-propagated operations in  $rbts_{left}$  and  $rbts_{right}$ , then **CreateBlock** returns *null* at line 64 and **Refresh** returns true at line 35 because another **Refresh** already propagated operations from  $v$ 's children into  $v$ .

If  $B_{new}$  is non-empty, line 36 of **Refresh** performs **GC** on  $rbts_{old}$ , which may discard unneeded Blocks and enqueued items in  $rbts_{old}$  (see Section 3.4), and saves the updated tuple in  $rbts_{new}$ . **Refresh** then inserts  $B_{new}$  into  $rbts_{new}.blocks$ , retrieves the enqueued items corresponding to the enqueues in  $B_{new}$  using **GetTopEnqs** and **GetBotEnqs** (described below), and stores them in  $newTopEnqs$  and  $newBotEnqs$ , respectively (lines 39–40). If  $v$  is not the root, **Refresh** joins  $newTopEnqs$  and  $newBotEnqs$  to the right ends of  $rbts_{new}.topEnqs$  and  $rbts_{new}.botEnqs$  (lines 42–43) so that the in-order of these trees represent enqueued items propagated to  $v$  in the order they reached  $v$ . Finally, line 51 attempts to CAS  $rbts_{new}$  into  $v.rbts$ , returning the result of the CAS to indicate whether it succeeded.

If  $v$  is the root, **Refresh** instead performs the concurrent batch of operations represented by  $B_{new}$  on  $rbts_{old}.state$  to obtain the new *state* of the deque,  $rbts_{new}.state$ . To do so, **Refresh** first calculates  $num_{topDeqs}$  and  $num_{botDeqs}$ , the numbers of **TopDequeues** and **BotDequeues** it must perform on the *state*, using the  $sum$  fields in  $B_{new}$  and  $B_{prev}$  (lines 45–46). **Refresh** then does a batch of concurrent **TopDequeues** by **Splitting**  $num_{topDeqs}$  items off of the right

■ **Algorithm 1** Implementation of deque: main routines.

---

1:	TopEnqueue(Object $e$ )	▷ enqueue $e$ to the top of the deque
2:	$B' \leftarrow \text{MaxBlock}(\text{leaf}.rbts.blocks)$	▷ previous Block in process's leaf
3:	let $B$ be a new Block object with fields: $index \leftarrow B'.index + 1$ ,	
4:	$sum_{topDeqs} \leftarrow B'.sum_{topDeqs}$ , $sum_{botDeqs} \leftarrow B'.sum_{botDeqs}$	
5:	$sum_{topEnqs} \leftarrow B'.sum_{topEnqs} + 1$ ,	▷ increment $sum_{topEnqs}$ for top enqueue
6:	$sum_{botEnqs} \leftarrow B'.sum_{botEnqs}$ ,	▷ BotEnqueue increments this instead
7:	$rbts_{new} \leftarrow \text{GC}(\text{leaf}, \text{leaf}.rbts, B, B')$	▷ returns new $rbts$ (after GC if necessary)
8:	$rbts_{new}.blocks \leftarrow \text{Insert}(rbts_{new}.blocks, B)$	
9:	$rbts_{new}.topEnqs \leftarrow \text{Append}(rbts_{new}.topEnqs, e)$	▷ insert $e$ as rightmost element
10:	$\text{leaf}.rbts \leftarrow rbts_{new}$	▷ write new tuple into process's leaf
11:	Propagate( $\text{leaf}.parent$ )	▷ propagate enqueue to root
12: TopDequeue : Object		
13:	$B' \leftarrow \text{MaxBlock}(\text{leaf}.rbts.blocks)$	▷ previous Block in process's leaf
14:	let $B$ be a new Block object with fields: $index \leftarrow B'.index + 1$ ,	
15:	$sum_{topEnqs} \leftarrow B'.sum_{topEnqs}$ , $sum_{botEnqs} \leftarrow B'.sum_{botEnqs}$ ,	
16:	$sum_{topDeqs} \leftarrow B'.sum_{topDeqs} + 1$ ,	▷ increment $sum_{topDeqs}$ for top dequeue
17:	$sum_{botDeqs} \leftarrow B'.sum_{botDeqs}$	▷ BotDequeue increments this instead
18:	$rbts_{new} \leftarrow \text{GC}(\text{leaf}, \text{leaf}.rbts, B, B')$	▷ returns new $rbts$ (after GC if necessary)
19:	$rbts_{new}.blocks \leftarrow \text{Insert}(rbts_{new}.blocks, B)$	
20:	$\text{leaf}.rbts \leftarrow rbts_{new}$	▷ write new tuple into process's leaf
21:	Propagate( $\text{leaf}.parent$ )	▷ propagate the operation to the root
22:	$response \leftarrow \text{CompleteTopDeq}(\text{leaf}, B.index)$	▷ retrieve dequeue response
23:	return ( $response = null ? B.response : response$ )	
24: Propagate(Node $v$ ) ▷ propagate operations from $v$ 's children up to root		
25:	if not Refresh( $v$ ) then Refresh( $v$ )	▷ double Refresh on $v$
26:	if $v \neq \text{root}$ then Propagate( $v.parent$ )	▷ recurse to parent
27: Refresh(Node $v$ ) : boolean		
28:	$rbts_{old} \leftarrow v.rbts$	▷ propagate operations into $v$
29:	$rbts_{left} \leftarrow v.left.rbts$	
30:	$rbts_{right} \leftarrow v.right.rbts$	
31:	if $v.rbts \neq rbts_{old}$ then return false	
32:	$B_{prev} \leftarrow \text{MaxBlock}(rbts_{old}.blocks)$	▷ previous Block in $v$
33:	▷ create a new Block $B_{new}$ from non-propagated $blocks$ in $v.left$ and $v.right$	
34:	$B_{new} \leftarrow \text{CreateBlock}(v, B_{prev}, rbts_{left}, rbts_{right})$	
35:	if $B_{new} = null$ then return true	▷ no new operations to propagate into $v$
36:	$rbts_{new} \leftarrow \text{GC}(v, rbts_{old}, B_{new}, B_{prev})$	
37:	$rbts_{new}.blocks \leftarrow \text{Insert}(rbts_{new}.blocks, B_{new})$	▷ insert new block
38:	▷ get the enqueued items from children that have not previously been propagated to $v$	
39:	$newTopEnqs \leftarrow \text{GetTopEnqs}(B_{prev}, B_{new}, rbts_{left}, rbts_{right})$	
40:	$newBotEnqs \leftarrow \text{GetBotEnqs}(B_{prev}, B_{new}, rbts_{left}, rbts_{right})$	
41:	if $v \neq \text{root}$ then	▷ join new items to enqueue RBTs
42:	$rbts_{new}.topEnqs \leftarrow \text{Join}(rbts_{old}.topEnqs, newTopEnqs)$	
43:	$rbts_{new}.botEnqs \leftarrow \text{Join}(rbts_{old}.botEnqs, newBotEnqs)$	
44:	else	▷ apply operations on root's state
45:	$num_{topDeqs} \leftarrow B_{new}.sum_{topDeqs} - B_{prev}.sum_{topDeqs}$	
46:	$num_{botDeqs} \leftarrow B_{new}.sum_{botDeqs} - B_{prev}.sum_{botDeqs}$	
47:	$\langle newstate, B_{new}.topDeqs \rangle \leftarrow \text{Split}(rbts_{old}.state, num_{topDeqs})$	
48:	$\langle B_{new}.botDeqs, newstate \rangle \leftarrow \text{Split}(newstate, newstate.size - num_{botDeqs})$	
49:	$newstate \leftarrow \text{Join}(newstate, newTopEnqs)$	
50:	$rbts_{new}.state \leftarrow \text{Join}(newBotEnqs, newstate)$	
51:	return CAS( $v.rbts, rbts_{old}, rbts_{new}$ )	▷ CAS new $rbts$ tuple into $v$

---



■ **Algorithm 2** Implementation of deque: helper routines.

---

```

52: CreateBlock(Node  $v$ , Block  $B_{prev}$ , rbts  $rbts_{left}$ , rbts  $rbts_{right}$ ) : Block
53:   ▷ create new Block  $B$  from subblocks in  $v.left$  and  $v.right$ 
54:    $B_L \leftarrow \text{MaxBlock}(rbts_{left}.blocks)$ 
55:    $B_R \leftarrow \text{MaxBlock}(rbts_{right}.blocks)$ 
56:   ▷ create a new Block  $B$  from non-refreshed blocks in  $v.left$  and  $v.right$ 
57:   let  $B$  be a new Block object with fields:  $index \leftarrow B_{prev}.index + 1$ ,
58:      $end_{left} \leftarrow B_L.index$ ,  $end_{right} \leftarrow B_R.index$ , ▷ new ends are max blocks in children
59:      $sum_{topEnqs} \leftarrow B_L.sum_{topEnqs} + B_R.sum_{topEnqs}$ ,
60:      $sum_{topDeqs} \leftarrow B_L.sum_{topDeqs} + B_R.sum_{topDeqs}$ ,
61:      $sum_{botEnqs} \leftarrow B_L.sum_{botEnqs} + B_R.sum_{botEnqs}$ ,
62:      $sum_{botDeqs} \leftarrow B_L.sum_{botDeqs} + B_R.sum_{botDeqs}$ 
63:   if  $B.end_{left} = B_{prev}.end_{left}$  and  $B.end_{right} = B_{prev}.end_{right}$  then
64:     return null ▷  $B$  is empty (it has no subblocks)
65:   else return  $B$ 

66: GetTopEnqs(Block  $B_{prev}$ , Block  $B_{new}$ , rbts  $rbts_{left}$ , rbts  $rbts_{right}$ ) : RBT
67:   ▷ retrieve top enqueue items belonging to a new Block  $B_{new}$  from children
68:    $blocks_L \leftarrow rbts_{left}.blocks$ 
69:    $blocks_R \leftarrow rbts_{right}.blocks$ 
70:   ▷ calculate number of newly promoted enqueues
71:    $leftEnqs \leftarrow blocks_L[B_{new}.end_{left}].sum_{topEnqs} - blocks_L[B_{prev}.end_{left}].sum_{topEnqs}$ 
72:    $rightEnqs \leftarrow blocks_R[B_{new}.end_{right}].sum_{topEnqs} - blocks_R[B_{prev}.end_{right}].sum_{topEnqs}$ 
73:   ▷ split the required number of items from right ends of the children's  $topEnqs$  RBTs
74:    $\langle *, leftChunk \rangle \leftarrow \text{Split}(rbts_{left}.topEnqs, leftEnqs)$ 
75:    $\langle *, rightChunk \rangle \leftarrow \text{Split}(rbts_{right}.topEnqs, rightEnqs)$ 
76:   return Join( $leftChunk$ ,  $rightChunk$ ) ▷ join the child chunks left to right

77: CompleteTopDeq(Node  $leaf$ , int  $h$ ) : Object ▷ return response of TopDequeue in Block  $h$  of  $leaf$ 
78:   ▷ after it has propagated to root, or null if helper has already recorded response.
79:   ▷ first, find Block index of the dequeue in the root and its position within the block
80:    $\langle b, i \rangle \leftarrow \text{IndexTopDeq}(leaf, h, 1)$ 
81:   ▷ find response to the top dequeue according to its position in the root
82:   if  $\langle b, i \rangle \neq \langle 0, 0 \rangle$  then
83:      $B \leftarrow root.rbts.blocks[b]$  ▷ root Block containing dequeue
84:     if  $B \neq null$  then
85:       if  $B.topDeqs.size < i$  then return empty
86:       else return SearchByRank( $B.topDeqs$ ,  $B.topDeqs.size - i + 1$ )
87:       ▷ in CompleteBotDeq do SearchByRank( $B.botDeqs$ ,  $i$ ) instead
88:     return null ▷ dequeue needs a Block discarded by GC

89: IndexTopDeq(Node  $v$ , int  $b$ , int  $i$ ) :  $\langle \text{int}, \text{int} \rangle$  ▷ returns  $\langle b', i' \rangle$  s.t.  $i^{th}$  TopDequeue in Block  $b$  of  $v$ 
90:   ▷ is  $i^{th}$  TopDequeue of root Block  $b'$  or  $\langle 0, 0 \rangle$  if GC discarded Block needed to find answer
91:   if  $v = root$  then return  $\langle b, i \rangle$ 
92:    $dir \leftarrow (v.parent.left = v ? left : right)$ 
93:    $blocks \leftarrow v.rbts.blocks$ 
94:    $blocks_p \leftarrow v.parent.rbts.blocks$ 
95:   ▷ N.B. if any Block required on lines 96–99 is not found, stop and return  $\langle 0, 0 \rangle$  instead
96:    $B_p \leftarrow \text{min Block in } blocks_p \text{ with } end_{dir} \geq b$  ▷  $B_p$  contains TopDequeue
97:    $B'_p \leftarrow \text{max Block in } blocks_p \text{ with } end_{dir} < b$ 
98:    $i' \leftarrow i + blocks[b - 1].sum_{topDeqs} - blocks[B_p'.end_{dir}].sum_{topDeqs}$ 
99:   if  $dir = right$  then  $i' \leftarrow i' + v.parent.left.rbts.blocks[B_p'.end_{left}].sum_{topDeqs}$ 
   -  $v.parent.left.rbts.blocks[B_p'.end_{left}].sum_{topDeqs}$ 
100:  return IndexTopDeq( $v.parent$ ,  $B_p.index$ ,  $i'$ )

```

---

end of  $rbts_{old}.state$  to obtain  $newstate$  and storing an RBT of the TopDequeue responses in  $B_{new}.topDeqs$  (line 47). If the split point ( $num_{topDeqs}$  in this case) is 0 or greater than the size of the RBT being split, **Split** returns the original tree and an empty tree. A batch of concurrent BotDequeues is then performed in a similar way at line 48. The batches of items enqueued by operations in  $B_{new}$  are contained in  $newTopEnqs$  and  $newBotEnqs$ , and they are added to  $newstate$  by joining them at the left and right of  $newstate$ , respectively (lines 49–50). Finally, **Refresh** attempts to CAS  $rbts_{new}$  into  $v$  as in the non-root case.

**CreateBlock** creates a new Block  $B_{new}$  to be inserted into node  $v$  after Block  $B_{prev}$ .  $B_{new}$  will represent all operations in  $v$ 's children that are not included in  $v$ 's Blocks up to  $B_{prev}$ . **CreateBlock** calculates the  $end_{left}$  and  $end_{right}$  fields of  $B_{new}$  by finding the indices of the latest Blocks in  $v$ 's children (line 58). The sum fields of  $B_{new}$  are calculated by adding the sum fields of these end Blocks in  $v$ 's children (lines 59–62). If  $B_{new}$  has no new operations, that is, if the  $end$  fields of  $B_{new}$  and  $B_{prev}$  are the same, **CreateBlock** returns *null* at line 63.

**GetTopEnqs** retrieves the items of TopEnqueues to be propagated with a new Block into a node  $v$ . It first calculates the number of items to be split from the  $topEnqs$  trees of  $v$ 's children (lines 71–72), splits those trees to get  $rightChunk$  and  $leftChunk$  (lines 74–75), and returns the RBT that results from joining the two chunks together (line 76).

To retrieve its response, a TopDequeue calls **CompleteTopDeq** on the dequeue's leaf Block after the dequeue has propagated to the root (line 22). Because operations help propagate one another, **CompleteTopDeq** first calls **IndexTopDeq** (line 80), a modified version of N&R's IndexDequeue, which uses recursion to “retrace” the TopDequeue's path from its location in a leaf Block to its location in the root's  $blocks$ . At each node  $v$  along the path up the tree, **IndexTopDeq** finds the Block  $B_p$  in  $v.parent$  that contains the TopDequeue at location  $\langle b, i \rangle$  in  $v$  (where  $b$  is a Block index and  $i$  is the rank of the TopDequeue in that Block) by searching for the minimum block in  $v.parent$ 's Blocks whose  $end$  field is at least  $b$  (line 96). The rank  $i'$  of the TopDequeue in  $B_p$  is found by adding to  $i$  the number of TopDequeues in  $B_p$  that came from Blocks before  $B$  in  $v$  (line 98). In the case that  $v$  is a right child, line 99 also adds to  $i'$  the number of all TopDequeues in  $B_p$  that came from  $v$ 's left sibling, because they will be linearized before  $B_p$ 's TopDequeues that came from  $v$  (see Section 3.3). If any Block that is needed to find the location of the TopDequeue has been discarded by garbage collection, **IndexTopDeq** returns  $\langle 0, 0 \rangle$  (line 90). Before discarding the Block, the garbage collection will ensure that some helper writes the TopDequeue's response in the  $response$  field of the TopDequeue's leaf Block. In this case, **CompleteTopDeq** returns *null*, indicating the TopDequeue should look there for its response (see line 23).

After **IndexTopDeq** finds that the TopDequeue is the  $i$ th TopDequeue in  $root.rbts.blocks[b]$ , **CompleteTopDeq** checks at line 85 if the deque was empty when that operation was performed, and if so, returns *null*. Otherwise, **CompleteTopDeq** returns the response to the  $i$ th TopDequeue in  $B$  by searching  $B.topDeqs$  for the item with the correct rank (line 86). (We use opposite orderings of the  $topDeqs$  and  $botDeqs$  trees, as described in line 87, because both trees hold a chunk of items split off the deque ordered from the bottom to top of the deque, so BotDequeues should take them in this order and TopDequeues should take them in the reverse order; this allows us to use a more uniform ordering of operations when we define the linearization in the next section.)

### 3.3 Linearization

We shall show that operations within a Block are all concurrent, which allows us to choose how to linearize them. Here, we define the order that is consistent with our pseudocode. We define  $D_{top}(B)$ ,  $D_{bot}(B)$ ,  $E_{top}(B)$ , and  $E_{bot}(B)$  to be the sequences of TopDequeues, BotDequeues, TopEnqueues and BotEnqueues, respectively, in a Block  $B$ . We define them recursively after first defining the *direct subblocks* of a Block.

► **Definition 1.** *The direct subblocks of a Block  $B_i = v.rbts.blocks[i]$  (where  $i > 0$ ) in an internal node  $v$  are the blocks that were inserted in  $v.left$  with indices from  $v.blocks[i-1].end_{left} + 1$  to  $v.blocks[i].end_{left}$  and the blocks that were inserted in  $v.right$  with indices from  $v.rbts.blocks[i-1].end_{right} + 1$  to  $v.rbts.blocks[i].end_{right}$ .*

If  $B$  is in a leaf node,  $D_{top}(B)$  is either a single `TopDequeue` if a `TopDequeue` created  $B$ , or empty otherwise. If  $B$  is in an internal node  $v$ , where  $B_1^L \cdots B_l^L$  are the direct subblocks of  $B$  in  $v.left$  and  $B_1^R \cdots B_r^R$  are the direct subblocks of  $B$  in  $v.right$ ,

$$D_{top}(B) = D_{top}(B_1^L) \cdots D_{top}(B_l^L) \cdot D_{top}(B_1^R) \cdots D_{top}(B_r^R). \quad (3.1)$$

$E_{top}(B)$  is defined similarly except that if  $B$  is a leaf Block,  $E_{top}(B)$  is either a single `TopEnqueue(e)` if a call to `TopEnqueue(e)` created  $B$  or the empty sequence otherwise.  $D_{bot}(B)$  and  $E_{bot}(B)$  are defined similarly to  $D_{top}(B)$  and  $E_{top}(B)$ .

Operations are linearized in the order their Blocks reach the root. Within each Block, `Refresh` does them in the following order: `TopDequeues`, `BotDequeues`, `TopEnqueues`, and `BotEnqueues` (lines 47–50). Thus, if  $B_0, \dots, B_n$  are the blocks that get inserted into the root, ordered by their indices, we define the linearization order as:

$$L_{total} = L(B_0) \cdot L(B_1) \cdots L(B_n), \text{ where } L(B) = D_{top}(B) \cdot D_{bot}(B) \cdot E_{top}(B) \cdot E_{bot}^{rev}(B). \quad (3.2)$$

It is convenient to linearize `BotEnqueues` in the reverse order of  $E_{bot}$ , denoted  $E_{bot}^{rev}$ , for uniformity in the pseudocode.

### 3.4 Garbage Collection

To guarantee the efficiency of our deque, we bound the size of the RBTs at each node so that operations on them can be done efficiently. To do so, we use a more refined version of the garbage collection method of the N&R queue. Specifically, **GC** discards obsolete elements of the RBTs of a node  $v$  every  $G_v = p_v^2 \lceil \log p \rceil$  operations added to  $v$ , where  $p_v$  is the number of leaves in the sub-tree rooted at  $v$ .

Before adding a new Block to  $v$  at line 8, 19 or 37, **GC** is called on  $v$  at line 7, 18 or 36. **GC** first creates an `rbts` object  $rbts_{new}$  with fields copied from  $v.rbts$ . The test at line 106 checks if the new Block to be added to  $v$  includes an operation whose rank among operations propagated to  $v$  is a multiple of  $G_v$ . (Since the number of operations in the new Block is at most  $p_v$  and  $p_v < G_v$ , the test detects when the total number of operations propagated surpasses another multiple of  $G_v$ .) If so, lines 107–120 perform garbage on collection  $rbts_{new}$ .

If  $v$  is not the root, lines 112–117 of **GC** discard unneeded items from the enqueue trees of  $rbts_{new}$  by splitting off from their left sides all enqueued items that have already been propagated to  $v$ 's parent. Since enqueued items are propagated along with the Blocks that represent the enqueues, we can discard the items enqueued by operations in all blocks up to  $B_{last}$ , computed on lines 108–110 to be the last Block of  $v$  propagated to  $v$ 's parent. Thus, the number of items to discard is calculated on lines 112–113 by subtracting the number of enqueued items that have already been discarded by previous garbage collection phases from the total number of enqueues in  $v$ 's Blocks up to  $B_{last}$ .

Unlike enqueued items in  $v$ , which can be discarded once they are propagated into  $v.parent$ , a Block of  $v$  must be retained if a pending dequeue will need to examine the Block while retracing its path to the root in `IndexTopDeq`. Discarding a Block in  $v$  could prevent a dequeue in the sub-tree rooted at  $v.parent$  (or  $v$  if  $v = root$ ) from retrieving its response. To avoid this problem, **GC** first calls **SplitBlock** (line 118), which recursively finds the latest Block  $B$  in  $v$  that has been propagated to the root by following `end` fields from the last Block

## 17:12 A Wait-Free Deque with Polylogarithmic Step Complexity

■ **Algorithm 3** Implementation of deque: garbage collection routines.

---

```

101: GC(Node  $v$ , rbts  $rbts_{old}$ , Block  $B_{new}$ , Block  $B_{prev}$ ) : rbts
102:   ▷ garbage collect the RBTs of node  $v$  before  $B_{new}$  gets appended to  $v.blocks$ 
103:    $rbts_{new} \leftarrow$  new rbts object with fields copied from  $rbts_{old}$ 
104:    $sum_{prev} \leftarrow B_{prev}.sum_{topEnqs} + B_{prev}.sum_{botEnqs} + B_{prev}.sum_{topDeqs} + B_{prev}.sum_{botDeqs}$ 
105:    $sum_{new} \leftarrow B_{new}.sum_{topEnqs} + B_{new}.sum_{botEnqs} + B_{new}.sum_{topDeqs} + B_{new}.sum_{botDeqs}$ 
106:   if  $sum_{prev} \bmod G_v \geq sum_{new} \bmod G_v$  then ▷ trigger garbage collection
107:     if  $v \neq root$  then ▷ discard propagated enqueues
108:        $B_p \leftarrow$  MaxBlock( $v.parent.rbts.blocks$ )
109:        $dir \leftarrow (v.parent.left = v ? left : right)$ 
110:        $B_{last} \leftarrow rbts_{new}.blocks[B_p.end_{dir}]$ 
111:       if  $B_{last} \neq null$  then ▷ skip if GC already done
112:          $num_{oldTopEnq} \leftarrow B_{last}.sum_{topEnqs} - rbts_{new}.discarded_{topEnqs}$ 
113:          $num_{oldBotEnq} \leftarrow B_{last}.sum_{botEnqs} - rbts_{new}.discarded_{botEnqs}$ 
114:          $rbts_{new}.discarded_{topEnqs} \leftarrow B_{last}.sum_{topEnqs}$ 
115:          $rbts_{new}.discarded_{botEnqs} \leftarrow B_{last}.sum_{botEnqs}$ 
116:          $\langle *, rbts_{new}.topEnqs \rangle \leftarrow$  Split( $rbts_{new}.topEnqs, num_{oldTopEnq}$ )
117:          $\langle *, rbts_{new}.botEnqs \rangle \leftarrow$  Split( $rbts_{new}.botEnqs, num_{oldBotEnqs}$ )
118:          $i \leftarrow$  SplitBlock( $v$ ).index ▷ find index of oldest Block to keep
119:         Help( $v$ ) ▷ help pending dequeues in  $v$ 's subtree
120:          $\langle *, rbts_{new}.blocks \rangle \leftarrow$  SplitByIndex( $blocks, i$ ) ▷ discard blocks with indices  $< i$ 
121:       return  $rbts_{new}$ 

122: SplitBlock(Node  $v$ ) : Block ▷ returns  $v$ 's latest Block propagated to root
123:   if  $v = root$  then  $B \leftarrow$  MaxBlock( $root.rbts.blocks$ )
124:   else
125:      $B_p \leftarrow$  SplitBlock( $v.parent$ )
126:      $B \leftarrow (v.parent.left = v ? v.rbts.blocks[B_p.end_{left}] : v.rbts.blocks[B_p.end_{right}])$ 
127:   return ( $B = null ?$  MinBlock( $v.rbts.blocks$ ) :  $B$ )

128: Help(Node  $v$ ) ▷ complete pending dequeues in  $v.parent$ 's sub-tree
129:   for each leaf  $l$  in subtree rooted at  $v.parent$  (or at  $v$  if  $v = root$ ) do
130:      $B_{cur} \leftarrow$  MaxBlock( $l.rbts.blocks$ )
131:     if  $B_{cur}.index \neq 0$  then ▷ do not help dummy Block
132:        $B_{prev} \leftarrow l.rbts.blocks[B_{cur}.index - 1]$ 
133:        $response \leftarrow null$ 
134:       if  $B_{cur}.sum_{topDeqs} \neq B_{prev}.sum_{topDeqs}$  and Propagated( $l, B_{cur}.index$ ) then
135:          $response \leftarrow$  CompleteTopDeq( $l, B_{cur}.index$ )
136:       else if  $B_{cur}.sum_{botDeqs} \neq B_{prev}.sum_{botDeqs}$  and Propagated( $l, B_{cur}.index$ ) then
137:          $response \leftarrow$  CompleteBotDeq( $l, B_{cur}.index$ )
138:       if  $response \neq null$  then
139:          $B_{cur}.response \leftarrow response$ 

140: Propagated(Node  $v$ , int  $b$ ) : boolean ▷ has  $v$ 's Block with index  $b$  been propagated to root?
141:   if  $v = root$  then return true
142:   else
143:      $blocks_p \leftarrow v.parent.rbts.blocks$ 
144:      $dir \leftarrow (v.parent.left = v ? left : right)$ 
145:     if MaxBlock( $blocks_p$ ).end $_{dir} < b$  then return false
146:     else ▷ Block  $b$  has been propagated to parent
147:      $B_p \leftarrow$  min Block in  $blocks_p$  with end $_{dir} \geq b$  ▷  $B_p$  exists, by line 145's test
148:     return Propagated( $v.parent, B_p.index$ )

```

---

in the root. Then, GC calls **Help**, which helps pending dequeues in all leaves in the subtree rooted at  $v$ 's parent to compute their responses. Finally, GC splits  $rbts_{new}.blocks$  to discard all Blocks strictly older than  $B$  (line 120).

**Help** ensures that the response to any pending dequeue that may need Blocks that are being discarded is written in the dequeue's leaf Block. **Help** only has to help dequeues that have been propagated to the root, since only those might need the discarded Blocks. For each leaf  $l$  in the subtree of  $v.parent$  (or  $v$  if  $v = root$ ) (see line 129), **Help** checks at line 134 or 136 if  $l$  contains a dequeue (using the first condition) that has been propagated to the root (using the call to **Propagated**). If so, **Help** completes the dequeue by calling **CompleteTopDeq** or **CompleteBotDeq** at line 135 or 137, and records the response in the *response* field of the dequeue's leaf Block (line 139). If **CompleteTopDeq** or **CompleteBotDeq** returns *null*, due to a missing Block on the path used by **IndexTopDeq** (or **IndexBotDeq**) at line 80, **Help** does nothing further since another process must have already helped the dequeue.

**Help** uses **Propagated** to check if a leaf's dequeue operation has been propagated to the root. **Propagated** recurses from the leaf to the root, finding in each node along that path the Block that contains the operation, returning false if there is no such Block.

## 4 Correctness

The goal of the correctness proof is to show that each operation is propagated to the root and applied to the state of the deque before the operation terminates. This allows us to argue that the linearization ordering of Equation (3.2) is valid. Garbage collection requires us to prove that no operation needs to access any of the discarded information. Due to space constraints, some details of the correctness proof are deferred to the full version.

### 4.1 Basic Properties

We first prove some basic properties of nodes' *rbts* fields. The first lemma describes how the *blocks* RBT is updated, ensuring the RBT's Blocks always have consecutive indices.

► **Lemma 2.** *If a node's blocks field is updated from  $T$  to  $T'$  and  $I$  is the set of indices in the RBT  $T$ , then the set of indices in  $T'$  is  $\{I \cap [i, \infty)\} \cup \{\max(I) + 1\}$  for some  $i \geq 0$  and the only new Block is the one with index  $\max(I) + 1$ .*

**Proof Sketch.** Each node's *blocks* field initially contains one dummy Block with *index* = 0. The *blocks* field of a node can change only when the *rbts* field of the node is updated at line 10, 20 or 51. It is straightforward to check that each such change modifies the RBT by optionally splitting off some blocks with indices below some threshold  $i$  (if GC is called) and adding one new Block with index  $\max(I) + 1$ . ◀

► **Definition 3.** *Define  $v.blocks[i]$  to mean the Block with index  $i$  that was in  $v.rbts.blocks$  at some point during the execution. By Lemma 2, this Block is unique.*

Next, we show that  $end_{left}$  and  $end_{right}$  fields of Blocks in a node are in sorted order. This ensures that Definition 1, which defines direct subblocks of a Block, makes sense.

► **Lemma 4.** *If a Block with index  $h > 0$  has been added to an internal node  $v$ 's blocks then  $v.blocks[h].end_{left} \geq v.blocks[h-1].end_{left}$  and  $v.blocks[h].end_{right} \geq v.blocks[h-1].end_{right}$ .*

**Proof.** Let  $B_{h-1}$  and  $B_h$  be the Blocks with indices  $h-1$  and  $h$  installed in  $v$  by two calls to **Refresh**,  $R_{h-1}$  and  $R_h$ , respectively. Since  $R_h$  performs a successful CAS at line 51, it must have read  $v.rbts$  at line 28 after  $R_{h-1}$  performed its successful CAS at line 51. Thus,  $R_h$  read

## 17:14 A Wait-Free Deque with Polylogarithmic Step Complexity

$v.left.rbts$  at line 29 after  $R_{h-1}$  did. By Lemma 2, the maximum index in  $v.left.rbts.blocks$  can only increase over time. Thus, the value stored in  $B.end_{left}$  at line 58 of  $R_{h-1}$ 's call to `CreateBlock` is less than or equal to the value stored in  $B'.end_{left}$  by  $R_h$ 's call to `CreateBlock`, as required. The argument for  $end_{right}$  is identical. ◀

► **Definition 5.** The subblocks of a Block  $B$  are defined recursively to be either direct subblocks of  $B$  (as defined by Definition 1), or subblocks of the direct subblocks of  $B$ .

► **Definition 6.** A Block  $B$  is propagated to node  $v$  if it is a subblock of some Block that has been inserted into  $v.rbts.blocks$ .

► **Definition 7.** A Block  $B$  contains an operation if the Block inserted by the operation into a leaf is a subblock of  $B$ .

Next, we show the  $sum$  fields of a Block  $B$  in node  $v$  correctly capture the number of operations contained in  $v$ 's Blocks up to and including Block  $B$ . We use the notation  $E_{top}(blocks[i..j])$  for  $E_{top}(blocks[i]) \cdots E_{top}(blocks[j])$ , and likewise for  $E_{bot}$ ,  $D_{top}$ , and  $D_{bot}$ .

► **Invariant 8.** If  $v$  is a node in the ordering tree and  $B = v.rbts.blocks[i]$ , then

$$\begin{aligned} B.sum_{topEnqs} &= |E_{top}(v.blocks[0..i])|, & B.sum_{botEnqs} &= |E_{bot}(v.blocks[0..i])|, \\ B.sum_{topDeqs} &= |D_{top}(v.blocks[0..i])|, & B.sum_{botDeqs} &= |D_{bot}(v.blocks[0..i])|. \end{aligned}$$

**Proof Sketch.** The invariant holds initially, since each  $blocks$  RBT contains a single empty Block with no operations and  $sum$  fields set to 0. Lemma 2 ensures each update of the RBT adds only a single Block  $B$ . We check that each such step preserves the invariant. When  $B$  is added to a leaf's  $blocks$  RBT at line 10 or 20, it contains a single operation, which is recorded in the appropriate  $sum$  field at lines 4–6 or 15–17. If an internal node's  $blocks$  RBT is updated at line 51, the new Block  $B$  was created by the call to `CreateBlock` on line 34 and inserted into the RBT at line 37. By Definition 1 and Lemma 2, the direct subblocks of  $v.blocks[0 \dots B.index]$  are  $v.left.blocks[0 \dots B.end_{left}]$  and  $v.right.blocks[0 \dots B.end_{right}]$ . It follows that  $B$ 's  $sum$  fields are computed correctly at lines 59–62 of the `CreateBlock`. ◀

The following result is easy to prove by induction on the height of the node(s) containing  $B$  and  $B'$ , using Definition 1 and Lemma 4.

► **Lemma 9.** The sets of subblocks of two Blocks  $B$  and  $B'$  at the same level of the ordering tree are disjoint.

Since the set of operations contained in a Block  $B$  is the set of all operations that appear in subblocks of  $B$  in the leaves of the ordering tree, we have the following corollary.

► **Corollary 10.** For any node  $v$  in the ordering tree, if  $i \neq j$ ,  $v.blocks[i]$  and  $v.blocks[j]$  cannot contain the same operation.

## 4.2 Operations are Propagated to the Root

In this section, we prove that operations are correctly propagated to the root of the *ordering tree*, where they are applied to the *state* field that represents the sequence of items in the deque. This requires showing (in Invariant 12) that enqueued items are added to a node's  $topEnqs$  or  $botEnqs$  RBT at the same time the corresponding `Enqueues` are added to the node's  $blocks$  RBT. But first, we must show that garbage collection does not discard any required information. Since Blocks in a node are propagated upwards in order by their index and also discarded by GC in order by their index, the following invariant implies that the GC routine only discards Blocks (at line 120) that are already propagated to the root.

► **Invariant 11.** For any non-root node  $v$ , the minimum Block in  $v.rbts.blocks$  and any Block returned by `SplitBlock( $v$ )` are subblocks of a Block that has been inserted into the root.

**Proof Sketch.** Initially, each node has a single dummy Block whose  $index$ ,  $end_{left}$  and  $end_{right}$  fields are 0. Thus, every Block is a subblock of the dummy Block in the root.

We show that each step maintains the invariant. The minimum Block in  $v$  is only modified because GC discards Blocks (at line 120) whose indices are smaller than the result of the call to `SplitBlock` at line 118. Assuming the invariant held for the response of the `SplitBlock`, the new minimum Block in  $v$  also satisfies the invariant. Proving the claim for the response of a `SplitBlock` is an easy induction on the depth of  $v$ , since `SplitBlock` simply returns a subblock of the Block returned by `SplitBlock( $v.parent$ )`. ◀

The next invariant says that the items stored in the  $topEnqs$  tree of a non-root node  $v$  are the arguments of all `TopEnqueues` that have propagated to  $v$ , except that some of the oldest ones may have been discarded by garbage collection. The  $discarded_{topEnqs}$  field keeps track of how many have been discarded. The second claim of the invariant ensures that we never discard items from the  $topEnqs$  tree before they have been propagated to  $v$ 's parent. The invariant also holds if all occurrences of  $top$  are replaced by  $bot$  (and the proof is identical).

► **Invariant 12.** Let  $v$  be a non-root node and let  $k$  be the maximum index of any Block in  $v.rbts.blocks$ . The in-order traversal of  $v.rbts.topEnqs$  yields the arguments of some suffix of the sequence  $E_{top}(v.blocks[1 \dots k])$ , where the suffix is obtained by removing the first  $v.rbts.discarded_{topEnqs}$  elements of the sequence. Moreover,  $v.rbts.discarded_{topEnqs}$  is less than or equal to the  $sum_{topEnqs}$  field of the last propagated Block in  $v$ .

**Proof Sketch.** Initially,  $v.rbts.blocks$  has one Block with  $index$  0,  $v.rbts.topEnqs$  is empty, and  $v.rbts.discarded_{topEnqs}$  is 0. We show each update to  $v.rbts$  preserves the invariant.

If  $v$  is a leaf, the argument is straightforward because the  $rbts$  field is updated by an `Enqueue` or `Dequeue` that adds one Block containing a single operation at line 8 or 19, and  $topEnqs$  gets one new item at line 9 in the case of `TopEnqueue` or none in the case of other operations.

If  $v$  is an internal node, we must show that a `Refresh` operation's successful CAS on line 51 preserves the invariant. The CAS adds a new Block  $B_{new}$  created by the call to `CreateBlock` at line 34 to  $v$ 's  $blocks$  RBT and simultaneously joins  $newTopEnqs$  to the right side of  $v$ 's  $topEnqs$  RBT (see line 42). We must therefore prove the following claim.

▷ **Claim 12.1.** The call to `GetTopEnqs` on line 39 returns a RBT  $newTopEnqs$  whose in-order traversal yields the arguments of the sequence  $E_{top}(B_{new})$ .

This claim is proved by tracing the code of `GetTopEnqs` and using the fact that the  $sum$  fields are accurate (by Invariant 8). The proof also uses the fact that the required enqueued items have not been discarded from  $v$ 's children's  $topEnqs$  trees, which follows from the induction hypothesis that the second claim of the invariant holds prior to updating  $v$ 's  $rbts$  field. ◀

When a Block reaches the root, instead of appending its enqueued items to  $topEnqs$  or  $botEnqs$ , the `Refresh` attaches them to the left and right ends of the  $state$  RBT. Similarly, it detaches the appropriate number of items for dequeues and saves them in the  $topDeqs$  and  $botDeqs$  fields of the Block (lines 45–50). This allows us to prove the following key invariant that the  $state$  field represents the state an abstract deque would have after sequentially performing, in their linearization order, all the operations that have been propagated to the

## 17:16 A Wait-Free Deque with Polylogarithmic Step Complexity

root so far. Moreover, the *topDeqs* and *botDeqs* fields of each Block contain the non-empty results of all the dequeue operations contained in that Block, a fact that is crucial for showing that Dequeues return results consistent with the linearization. Recall that  $L(B)$  is defined in Equation (3.2) to be the linearization order of operations in a Block  $B$  of the root.

► **Invariant 13.** *Let  $B_0, \dots, B_n$  be all the Blocks that have so far been added to the root's blocks RBT. The in-order traversal of  $\text{root.rbts.state}$  is the state of an initially empty deque after the sequential execution  $L(B_0) \cdot L(B_1) \cdot \dots \cdot L(B_n)$ . Moreover, for  $0 \leq i \leq n$ , the reverse in-order traversal of  $B_i.\text{topDeqs}$  gives the sequence of non-empty responses to the TopDequeues of  $L(B_i)$  in this sequential execution, and the in-order traversal of  $B_i.\text{botDeqs}$  gives the sequence of non-empty responses to the BotDequeues of  $L(B_i)$  in this sequential execution.*

**Proof Sketch.** The claim holds initially: the root has a single Block  $B_0$  with no operations and the *state* is empty. We show each addition of a Block to the root preserves the invariant.

Consider the Refresh whose CAS on line 51 adds  $B_n$  to the root's *blocks* tree. This CAS simultaneously updates the *state* field to  $\text{rbts}_{\text{new}}.\text{state}$ , which was constructed by performing two Split and two Join operations on the previous state (see lines 45–50).

By Invariant 8, line 45 sets  $\text{num}_{\text{topDeqs}}$  to  $|D_{\text{top}}(B_n)|$ . Thus, line 47 splits  $|D_{\text{top}}(B_n)|$  elements off the right end of the old *state* and stores these values, which are the non-empty responses to operations the TopDequeues of  $B_n$ , into  $B_n.\text{topDeqs}$ . Line 48 handles the BotDequeues of  $D_{\text{bot}}(B_n)$  similarly. By Claim 12.1, line 49 adds the items enqueued by the TopEnqueues of  $E_{\text{top}}(B_n)$  to the right end of the state. Line 50 handles the BotEnqueues similarly. The order in which the operations are applied matches  $L(B_n)$  defined in Equation (3.2). ◀

A double Refresh on a node  $v$  guarantees that all operations that had previously propagated to  $v$ 's children are propagated to  $v$ : if both Refreshes fail their CAS, then a concurrent Refresh must have successfully propagated the operations. The following two lemmas formalize this argument, similarly to previous papers that use a double Refresh [1, 18, 26].

► **Lemma 14.** *All operations contained in Blocks of  $v$ 's children when a Refresh( $v$ ) performed line 28 are contained in Blocks of  $v$  when the Refresh returns true at line 35 or performs a successful CAS at line 51.*

**Proof.** Suppose an operation  $op$  is contained in a Block  $B$  of  $v$ 's left child  $v'$  when Refresh performs line 28. (The proof for the right child of  $v$  is identical.) Refresh gets a snapshot of  $v'.\text{rbts}$  at line 29. Then, the call to CreateBlock on line 34 finds the Block  $B_L$  in this snapshot of  $v'.\text{rbts.blocks}$  with the maximum index at line 54. By Lemma 2,  $B'.\text{index} \geq B.\text{index}$ . At line 58, CreateBlock writes  $B'.\text{index}$  into the  $\text{end}_{\text{left}}$  field of the new Block  $B_{\text{new}}$ .

If the Refresh returns true at line 35, then  $B_{\text{prev}}.\text{end}_{\text{left}} = B_{\text{new}}.\text{end}_{\text{left}} \geq B.\text{index}$ . If the Refresh performs a successful CAS at line 51 then that CAS successfully installs Block  $B_{\text{new}}$  in  $v$ 's Blocks tree. Either way,  $v$  contains a Block whose  $\text{end}_{\text{left}}$  field is greater than or equal to  $B.\text{index}$ . It follows from Definition 1 and Lemma 4, that  $B$  is a direct subblock of some Block that has been added to  $v$ . Thus,  $op$  is contained in a Block of  $v$ , as required. ◀

► **Lemma 15.** *All operations contained in Blocks of  $v$ 's children when a Propagate( $v$ ) was invoked are contained in Blocks of  $v$  when the Propagate completes line 25.*

**Proof.** If either call to Refresh on line 25 returns true, then the claim follows from Lemma 14. So, suppose the Propagate's calls  $R_1$  and  $R_2$  to Refresh both return false, which can happen at line 31 or 51. In either case, some other Refresh must have changed  $v.\text{rbts}$  between line 28



of the Refresh and the time it returns false. Let  $R'_1$  and  $R'_2$  be instances of Refresh that update  $v.rbts$  during  $R_1$  and  $R_2$ , respectively. Because  $R'_2$  performs a successful CAS, it must have read  $v.rbts$  at line 28 after  $R'_1$  modifies it (and hence after  $R_1$  executes line 28). Moreover,  $R'_2$  performs its successful CAS before  $R_2$ 's failed CAS. Thus, the claim follows from Lemma 14 applied to  $R'_2$ . ◀

The following lemma implies that each operation's linearization point is within the interval of time when the operation is executing. It follows easily from Lemma 15.

► **Lemma 16.** *Propagate ensures that the operation that has called it is contained in a Block in the root before the Propagate terminates.*

It follows from Invariant 13 and Lemma 16 that each operation is performed on the root's *state* when a Block containing the operation reaches the root and the root's *state* is updated.

### 4.3 Retrieval of a Dequeue's Response and Linearizability

In this section, we show each dequeue returns the response it would in the sequential execution  $L_{total}$ . We first show `IndexTopDeq` correctly locates the `TopDequeue` in the root.

► **Lemma 17.** *If  $v.rbts.blocks[b]$  contains at least  $i$  `TopDequeues` and it is a subblock of a root Block, then `IndexTopDeq`( $v, b, i$ ) returns  $\langle b', i' \rangle$  such that the  $i$ th `TopDequeue` in  $D_{top}(v.rbts.blocks[b])$  is the  $i'$ th `TopDequeue` in  $D_{top}(root.rbts.blocks[b'])$ , or  $\langle 0, 0 \rangle$  in the case that any of the Blocks looked up at lines 96–99 have been discarded.*

**Proof Sketch.** We use induction on the depth of  $v$ . If  $v$  is the root, line 91 satisfies the claim. Suppose the claim holds for  $v.parent$  and Blocks looked up at lines 96–99 are found. By Definition 1,  $v.rbts.blocks[b]$  is a direct subblock of the Block  $B_p$  found on line 96. It follows from Invariant 8 that line 98 computes  $i'$  to be the rank of the  $i$ th `TopDequeue` within  $D_{top}(v.rbts.blocks[b])$  plus the number of `TopDequeues` in the subblocks of  $B_p$  in  $v$  that precede  $D_{top}(v.rbts.blocks[b])$ . By Definition 1, the  $i'$ th `TopDequeue` in  $D_{top}(B_p)$  is the  $i$ th `TopDequeue` in  $D_{top}(v.rbts.blocks[b])$  if  $v$  is the left child of  $v.parent$ . If  $v$  is the right child of  $v.parent$ , the  $i'$ th dequeue in  $D_{top}(B_p)$  is shifted at line 99 by the number of `TopDequeues` in the left subblocks of  $B_p$ . Thus, the  $i$ th `TopDequeue` in  $D_{top}(v.rbts.blocks[b])$  is the  $i'$ th `TopDequeue` in  $D_{top}(B_p)$ . By the induction hypothesis, the call to `IndexTopDeq`( $v.parent, B_p.index, i'$ ), at line 100 returns the required pair. ◀

The next two lemmas show that each dequeue returns a result consistent with the linearization, either because the dequeue reads the result directly from the `botDeqs` or `topDeqs` field of the root Block that contains the dequeue, or because some other operation has retrieved the response from there and stored it in the dequeue's leaf Block as part of the helping performed in GC. The argument about GC's helping must also carefully show that information is never discarded before all operations that need the information have been helped.

► **Lemma 18.** *A call to `CompleteTopDeq`( $leaf, h$ ) returns either the response the `TopDequeue` in  $leaf.rbts.blocks[h]$  would receive in the sequential execution  $L_{total}$  or null. Moreover, it returns null only if one of the Blocks looked up in lines 96–99 or line 83 have been discarded. The same claim holds for `CompleteBotDeq`.*

**Proof Sketch.** Before calling `CompleteTopDeq` at line 22 or 135, the call to `Propagate` at line 21 or the test at line 134 ensures the `TopDequeue` to complete is already propagated to the root. By Lemma 17, the call to `IndexTopDeq` at line 80 returns the `TopDequeue`'s location in

## 17:18 A Wait-Free Deque with Polylogarithmic Step Complexity

the root, or  $\langle 0, 0 \rangle$  if some Block needed has been discarded. In the latter case, or if the search at line 83 does not find the required Block, `CompleteTopDeq` returns *null* at line 88, and the lemma is satisfied. Otherwise, line 83 finds the root Block  $B$  containing the `TopDequeue`. Then, if  $B.topDeqs$  contains fewer than  $i$  elements (line 85), `CompleteTopDeq` returns *empty*, since there are fewer than  $i$  non-empty `TopDequeues` in  $B$ , by Invariant 13. Otherwise, Invariant 13 ensures the response of the  $i$ th `TopDequeue` in  $B$  is the  $(n - i + 1)$ th element in  $B.topDeqs$ , which is returned by line 86. The argument for `BotDequeues` is similar. ◀

► **Lemma 19.** *If a call to `CompleteTopDeq` on a leaf Block returns *null*, then either the `TopDequeue` in that leaf Block has terminated or the value returned by that `TopDequeue` in the sequential execution  $L_{total}$  has been written into the `TopDequeue`'s leaf Block. The same claim holds for `CompleteBotDeq`.*

**Proof Sketch.** We use induction on the number of completed calls to `CompleteTopDeq`. (The proof for `CompleteBotDeq` is identical.) Consider a call  $C$  to `CompleteTopDeq` and assume the claim holds for all calls that complete before  $C$  does. By Lemma 18,  $C$  can return *null* only if one of the Blocks it searches for on line 83 or lines 96–99 has been discarded.

First, suppose  $C$  fails to find a required Block  $B$  in some node  $v$  on line 83, 96 or 97.  $B$  is either the Block that contains the `TopDequeue` or the preceding Block.  $B$  must have been discarded by some call to `GC(v)` whose call to `SplitBlock(v)` on line 118 returned a Block  $B'$  with a larger index than  $B$ . By Invariant 11,  $B'$  was already propagated to the root, so  $B$  must also have been propagated to the root. If the `TopDequeue` has not terminated, the call to `Help(v)` on line 119 performs `CompleteTopDeq` on that `TopDequeue`. If that call of `CompleteTopDeq` returned *null*, the claim holds by the induction hypothesis. Otherwise, it returned the correct response of the `TopDequeue` by Lemma 18 and this response was recorded in the *response* field of the dequeue's leaf Block, so the claim holds.

The case where  $C$  returns *null* because a required Block is not found in the left sibling of  $v$  on line 99 of `IndexTopDeq` can be argued similarly. (This is where we use the fact that `Help` helps all processes in the subtree of  $v.parent$ , not just in  $v$ 's subtree.) ◀

Finally, we prove the main result of our correctness proof.

► **Theorem 20.** *The deque is linearizable.*

**Proof.** By Corollary 10 applied to the root,  $L_{total}$  is a permutation of operations in the concurrent execution. Lemma 16 ensures that the permutation includes all completed operations, since they are propagated to the root before they terminate. Lemma 16 also ensures that an operation appears in  $L_{total}$  before any operations that begin after it terminates. Finally, we show each `TopDequeue` returns the same response as it would in the sequential execution  $L_{total}$ . (The proof for `BotDequeues` is identical.) This follows from Lemma 19 if the call to `CompleteTopDeq` at line 22 returns *null*, or from Lemma 18 otherwise. ◀

## 5 Space Complexity

We bound the amount of memory accessible through the ordering tree data structure. We assume that the number of operations performed on the deque can be represented in binary using  $O(1)$  memory words. We first bound the number of operations and Blocks in a node  $v$ 's *blocks* tree. We say that `GC` on  $v$  *succeeds* if the test at line 106 is satisfied and the operation that called `GC` successfully installs the resulting RBTs tuple in  $v$ . A successful `GC` discards at line 120 any Blocks that have propagated to the root. Thus, each Block that remains after a successful `GC` has an operation that was in the process of propagating that Block to the root

at the time the *RBTs* field of  $v$  was read prior to GC. Since each process has at most one pending operation, at most  $p_v$  Blocks remain immediately after GC, where  $p_v$  is the number of leaves in the subtree rooted at  $v$ . Since each Block contains at most  $p_v$  operations, there are at most  $p_v^2$  operations remaining immediately after GC. The next GC on  $v$  succeeds after another  $G_v = p_v^2 \lceil \log p \rceil$  operations propagate to  $v$ . Thus, the number of operations in  $v$ 's *blocks* tree can never exceed  $p_v^2 + G_v = O(p_v^2 \log p)$ . The test on line 63 ensures that each Block added to a node has at least one subblock, and it follows that each Block contains at least one operation. Thus, the number of Blocks in  $v$  is also  $O(p_v^2 \log p)$  at all times.

When GC on a non-root node  $v$  succeeds, all items in  $v$ 's enqueue RBTs that have been propagated to  $v$ 's parent are discarded at lines 116–117. So, following the GC, at most  $p_v$  enqueued items remain (at most one per process). Since GC succeeds every  $p_v^2 \lceil \log p \rceil$  operations added to  $v$  (line 106), there are at most  $p_v + p_v^2 \lceil \log p \rceil$  items in  $v$ 's enqueue RBTs at any time. Thus, the total size of  $v$ .*RBTs* is  $O(p_v^2 \log p)$  for each non-root node  $v$ .

Each Block in the root contains a RBT of responses to dequeues in that Block. Since it was shown above that the *blocks* RBT of the root contains  $O(p^2 \log p)$  operations, the total size of the dequeue trees in all of the root's Blocks is  $O(p^2 \log p)$ . The space used by the *state* tree in the root is  $O(q_{max})$  where  $q_{max}$  is the maximum size of the deque. The root's total space, including its *state*, *blocks*, and its Blocks' dequeue trees, is  $O(q_{max} + p^2 \log p)$ .

The value of  $p_v$  of a non-root node  $v$  at each level of our *ordering tree*, from top to bottom, is  $\frac{p}{2}, \frac{p}{4}, \dots, 1$ , and there are  $2, 4, \dots, p$  non-root nodes at each level of the tree. Since each non-root node  $v$  uses  $O(p_v^2 \log p)$  space, the total space used by the non-root nodes is  $O(\sum_{i=1}^{\lceil \log p \rceil} 2^i (\frac{p}{2^i})^2 \log p) = O(p^2 \log p \sum_{i=1}^{\lceil \log p \rceil} \frac{1}{2^i}) = O(p^2 \log p)$ . Together with the space used by the root, the total space usage of the ordering tree is  $O(q_{max} + p^2 \log p)$ .<sup>1</sup>

## 6 Step Complexity

RBT operations on the root's *state* tree take  $O(\log q_{max})$  steps. Since the sizes of all other RBTs are polynomial in  $p$ , operations on them each take  $O(\log p)$  steps.

We first bound the step complexity of operations *excluding garbage collection*. Each operation performs  $O(\log p)$  steps to insert the operation at a leaf of the ordering tree and then calls `Propagate`. `Propagate` calls `Refresh` at most twice at each of  $O(\log p)$  nodes along a path from the leaf to the root. At non-root nodes, each `Refresh`, including the calls to the subroutines `CreateBlock`, `GetTopEnqs` and `GetBotEnqs`, does  $O(1)$  RBT operations and  $O(1)$  other steps, for a total of  $O(\log p)$  steps. At the root, `Refresh` also does RBT operations on the *state* tree (lines 47–50), so it takes  $O(\log p + \log q_{max})$  steps. Thus, `Propagate` takes  $O(\log^2 p + \log q_{max})$  steps in total. A `TopDequeue` also calls `CompleteTopDeq` at line 22, which calls `IndexTopDeq`. `IndexTopDeq` searches a *blocks* tree at each level of the ordering tree, for a total of  $O(\log^2 p)$  steps and the rest of `CompleteTopDeq` searches two RBTs in  $O(\log p)$  steps. Summing up, each enqueue or dequeue takes  $O(\log^2 p + \log q_{max})$  steps, excluding calls to GC.

Now we consider the contribution of GC to the amortized step complexity of operations. First, we bound the steps taken by routines called by GC. `Propagated` takes at most  $O(\log^2 p)$  steps to search *blocks* RBTs (lines 145–147) at each level of the ordering tree. In the worst

<sup>1</sup> In addition, processes could have pointers to additional objects in local memory. For example, in a pathological execution where processes fall asleep during a `Refresh` holding pointers in their local memory to old, totally disjoint states of the deque, this could add an additional  $\Theta(pq_{max})$  memory usage.

## 17:20 A Wait-Free Deque with Polylogarithmic Step Complexity

case,  $\text{Help}(v)$  performs a *blocks* search (line 130), calls  $\text{Propagated}$  (line 134 or 136) and calls  $\text{CompleteTopDeq}$  (line 135) or  $\text{CompleteBotDeq}$  (line 137) for the  $2p_v$  leaf descendants of  $v.\text{parent}$ . So  $\text{Help}(v)$  takes  $O(p_v \log^2 p)$  steps.  $\text{SplitBlock}(v)$  takes  $O(\log^2 p)$  steps since it recurses to the root from  $v$ , searching a *blocks* RBT at each level.

If garbage collection is not triggered at line 106, GC takes  $O(1)$  steps. If it is triggered, GC performs  $O(\log p)$  steps on RBTs (lines 108–117) and calls  $\text{SplitBlock}$  (line 118),  $\text{Help}$  (line 119), and  $\text{SplitByIndex}$  on a *blocks* tree (line 120) for a total of  $O(p_v \log^2 p)$  steps. Each of the  $p_v$  processes whose leaves are descendants of  $v$  satisfy the trigger at line 106 only once every  $G_v$  operations added to  $v$ , and each of them performs  $O(p_v \log^2 p)$  steps in that case. So, all processes perform a total of  $O(p_v^2 \log^2 p)$  steps doing GC at  $v$  once every  $G_v$  operations propagated to  $v$ . So the amortized number of steps for GC at node  $v$  is  $O(p_v^2 \log^2 p / G_v) = O(\log p)$  per operation. Each operation does GC at each level of the tree, the total amortized number of steps spent on GC is  $O(\log^2 p)$  per operation. The total amortized step complexity of a deque operation, including GC, is thus  $O(\log^2 p + \log q_{max})$ .

Our deque is wait-free because each recursion recurses from a leaf of the ordering tree to the *root* or vice versa, and its only loop (line 129) iterates through a finite set. Also, all RBT operations are applied to a local snapshot of the RBT, and are therefore wait-free.

### 7 Elimination

As an optimization, it would be straightforward to incorporate elimination [14] into our deque. If any block (in any non-root node) contains both enqueues and dequeues on the same end of the deque, they can be eliminated and there is no need to propagate them further up the tree. For example, suppose a  $\text{Refresh}$  builds a block  $B_{new}$  that represents 5  $\text{TopDequeues}$  and 8  $\text{TopEnqueues}$ . We could eliminate 5 pairs of operations by reducing the  $sum_{topDeqs}$  field of the block by 5, removing 5 elements from the  $newTopEnqs$  tree constructed on line 39 using  $\text{Split}$ , and storing those 5 elements in a new  $topDeqs$  RBT field of the block, as is done in the Blocks of the root. Thus, only the 3 remaining  $\text{TopEnqueues}$  would be propagated further. Elimination also requires some changes to the  $\text{CompleteTopDeq}$  function: when tracing a  $\text{TopDequeue}$  up the tree, it would detect if the operation got eliminated at some node, and then use the operation's rank among  $\text{TopDequeues}$  eliminated in that Block to select the response from the Block's  $topDeqs$  RBT. The eliminated operations would not appear in the  $E_{top}$  and  $D_{top}$  sequences of the Block that are used to define the linearization ordering  $L_{total}$ . The eliminated operations would be linearized when the Block that eliminates them is installed in the node, instead of when they are propagated to the root.

### 8 Open Questions

It would be interesting to experimentally compare the performance of the new deque to existing lock-free dequeues. Such experiments could also measure the effects of elimination on throughput. Our new deque is designed to ensure time bounds even in worst-case executions. However, this comes at the cost of operations performing more steps in the best case (for example, when an operation runs with no contention). Could the deque be made adaptive so that its step complexity depends on the number of concurrent operations, rather than on the number of processes in the system? This might be achieved by having operations choose a leaf at which to inject an operation (as in [1]) rather than having a statically assigned leaf for each process. (This might also handle the case where the set of processes is not known in advance.) Or perhaps the fast-path slow-path methodology of [21] could be used to make the

best case faster while maintaining good bounds in the worst case. An interesting theoretical direction would be narrowing the gap between the  $\Omega(\log p)$  lower bound on the amortized step complexity of operations [19] and our  $O(\log^2 p + \log q)$  upper bound. Can we design (implicit or explicit) representations of batches of operations for other data structures so that ordering trees yield sublinear-time operations for still more data structures?

---

## References

- 1 Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 538–547, New York, NY, USA, 1995. doi:10.1145/225058.225271.
- 2 Ole Agesen, David Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS-based concurrent dequeues. *Theory of Computing Systems*, 35(3):349–386, 2002. doi:10.1007/S00224-002-1058-2.
- 3 Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001. doi:10.1007/S00224-001-0004-Z.
- 4 Hagit Attiya and Arie Fouren. Lower bounds on the amortized time complexity of shared objects. In *Proc. 21st International Conference on Principles of Distributed Systems*, volume 95 of *LIPICs*, pages 16:1–16:18, 2017. doi:10.4230/LIPICs.OPODIS.2017.16.
- 5 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 17.1. MIT Press, fourth edition, 2022.
- 6 David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent dequeues. In *Proc. 14th International Conference on Distributed Computing*, volume 1914 of *LNCS*, pages 59–73, 2000. doi:10.1007/3-540-40026-5\_4.
- 7 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A scalable, correct time-stamped stack. In *Proc. 42nd ACM Symposium on Principles of Programming Languages*, pages 233–246, 2015. doi:10.1145/2676726.2676963.
- 8 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.
- 9 Matthew Graichen, Joseph Izraelevitz, and Michael L. Scott. An unbounded nonblocking double-ended queue. In *Proc. 45th International Conference on Parallel Processing*, pages 217–226, 2016. doi:10.1109/ICPP.2016.32.
- 10 Michael Barry Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, 1999. Available from <http://i.stanford.edu/TR/CS-TR-99-1624.html>.
- 11 Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978. doi:10.1109/SFCS.1978.3.
- 12 Andreas Haas. *Fast Concurrent Data Structures Through Timestamping*. PhD thesis, University of Salzburg, 2015.
- 13 Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18(3):189–207, 2006. doi:10.1007/S00446-005-0144-5.
- 14 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *Journal of Parallel and Distributed Computing*, 70(1):1–12, 2010. doi:10.1016/J.JPDC.2009.08.011.
- 15 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.
- 16 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003. doi:10.1109/ICDCS.2003.1203503.

- 17 Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 201–210, 1998. doi:10.1145/277697.277735.
- 18 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proc. 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *LNCS*, pages 408–419, 2005. doi:10.1007/11590156\_33.
- 19 Siddhartha V. Jayanti, Robert E. Tarjan, and Enric Boix-Adserà. Randomized concurrent set union and generalized wake-up. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 187–196, 2019. doi:10.1145/3293611.3331593.
- 20 Donald E. Knuth. *The Art of Computer Programming*, chapter 2.2.1. Addison-Wesley, third edition, 1997.
- 21 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proc. 17th ACM Symposium on Principles and Practice of Parallel Programming*, pages 141–150, 2012. doi:10.1145/2145816.2145835.
- 22 Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on JVM. In *Proc. 35th International Conference on Computer Aided Verification, Part I*, volume 13964 of *LNCS*, pages 156–169, 2023. doi:10.1007/978-3-031-37706-8\_8.
- 23 Paul Martin, Mark Moir, and Guy Steele. DCAS-based concurrent deques supporting bulk allocation. Technical Report SMLI TR-2002-11, Sun Microsystems, oct 2002. Available from <https://dl.acm.org/doi/10.5555/1698157>.
- 24 Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *Proc. 9th International Euro-Par Conference on Parallel Processing*, volume 2790 of *LNCS*, pages 651–660, 2003. doi:10.1007/978-3-540-45209-6\_92.
- 25 Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 103–112, 2013. doi:10.1145/2442516.2442527.
- 26 Hossein Naderibeni and Eric Ruppert. A wait-free queue with polylogarithmic step complexity. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 124–134, 2023. doi:10.1145/3583668.3594565.
- 27 Niloufar Shafiei. Non-blocking array-based algorithms for stacks and queues. In *Proc. 10th International Conference on Distributed Computing and Networking*, volume 5408 of *LNCS*, pages 55–66, 2009. doi:10.1007/978-3-540-92295-7\_10.
- 28 Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008. doi:10.1016/J.JPDC.2008.03.001.
- 29 Robert Endre Tarjan. *Data Structures and Network Algorithms*, chapter 4.2. SIAM, Philadelphia, USA, 1983. doi:10.1137/1.9781611970265.
- 30 R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.