

An Algorithm for Obstruction-free Double-Ended Queues

Sean Brennan
Advised by Michael Scott

May 1, 2013

Abstract

This paper describes a nonblocking algorithm for implementing a data structure known as a double-ended queue. There have only been a handful of algorithms for non-blocking dequeues, each of which has deficiencies including boundedness of size, reduced concurrency, and a need for special atomic primitives. With the help of Michael Scott, I have devised a new approach built off of prior work from Herlihy et al. that achieves four highly desirable properties: *unboundedness*, *space efficiency*, *high concurrency*, and finally *obstruction-freedom*. Performance analysis shows that this nonblocking double-ended queue can begin to provide higher throughput than a similar algorithm using mutual exclusion, though there is still room for improvement.

1 Introduction

1.1 Nonblocking Algorithms

An algorithm is described as *nonblocking* if it guarantees that separate threads of execution which share some resource cannot be blocked - that is, halted indefinitely. This class of algorithms is highly desirable because they allow the programmer to forgo locks and mutual exclusion. While mutual exclusion is understandable intuitively and has been put to use in a wide range of applications, it has its flaws. Locks can easily lead to very subtle and difficult to track program bugs: deadlock, livelock, the convoy effect, and priority inversion are just a few examples. From a pragmatic standpoint, blocking on locks also wastes cycles that a thread could be using to accomplish more meaningful work.

Despite these benefits, nonblocking algorithms carry the tradeoff of extremely complex code. Nonblocking algorithms must be written such that a thread can be halted anywhere in its execution, wake up some unknown amount of time in the future, and be able to recognize whether it should continue with its operation or try again. This makes these algorithms very difficult to write and to test. Furthermore, different flavors of nonblocking algorithms provide different levels of guarantees about their behavior that can complicate their implementations even more.

- *Obstruction-free*: This class of nonblocking algorithms is the weakest. It specifies that some thread, in the absence of contention, will always complete its operation in a bounded number of steps. All this ensures is that the system is free of *deadlock*.

- *Lock-free*: This class of nonblocking algorithms builds on the guarantees of obstruction freedom. It specifies that some thread will always complete its operation in a bounded number of steps. It follows from this principle that if some thread's operation fails, it is always because some other thread's operation succeeded. This means that the system is free of *deadlock* as well as *livelock*.
- *Wait-free*: This class of nonblocking algorithms is the strongest and builds on the guarantees of lock freedom. To summarize this property more succinctly, all threads in a wait-free algorithm must complete their operations in a bounded number of steps. On top of *deadlock* and *livelock*, wait-free algorithms remove the problem of *starvation*.

The rest of this paper is focused solely on obstruction-free algorithms. These algorithms have the easiest properties to prove and tend to be the simplest to implement.

1.2 Double-Ended Queues

A double-ended queue, or *deque*, is a very powerful and useful data structure. It builds off the notion of the classical FIFO queue, which has a tail for enqueueing items and a head for dequeueing items. The deque extends this data structure by allowing the programmer to push and pop to and from both ends of the queue. For this reason, deques are able to generalize FIFO structures (queues) and LIFO structures (stacks) easily well. Furthermore, if one has a nonblocking algorithm for deques, then one trivially has a nonblocking algorithm for stacks and queues as well.

2 Previous Work

Despite the power of the double-ended queue abstraction, there are so far only a handful of papers concerning nonblocking implementations of these structures. Each has its own strengths and weaknesses, but none so far guarantees the suite of properties that I would like to achieve with my algorithm: *unboundedness*, *space efficiency*, *high concurrency*, and *obstruction-freedom*. Each has nevertheless lent inspiration to my algorithm, even if only a small amount.

The first paper is Michael's "CAS-Based Lock-Free Algorithm for Shared Deques" [2]. In this paper, Michael describes how to implement a lock-free deque. Michael's algorithm achieves this through the use of a double-wide compare-and-swap operation that updates an *anchor* variable, which keeps track of references to the two queue ends. After each successful push operation, a participating thread attempts to *stabilize* the structure by making sure all nodes are connected to one another; if this stabilize operation fails, then the next pop operations attempts it before continuing.

The Michael deque is powerful, but has its share of shortcomings. In terms of strengths, it is *unbounded*, which the next algorithm I will discuss is not. Furthermore, its guarantees of behavior make it *lock-free*, while I propose only to implement an obstruction-free algorithm. Finally, Michael describes several simple extensions to the algorithm to make it highly *space-efficient*. The biggest problem with this deque is its lack of concurrency between ends. It serializes all updates to the deque, meaning that a left pop can cause a right push to abort, for example. In the average use case, this "interference" is completely artificial.

Furthermore, the call for a double-wide compare-and-swap could restrict the architectural possibilities of the algorithm.

These problems are addressed in “Obstruction-Free Synchronization: Double-Ended Queues as an Example” by Herlihy et al [1]. Using the compare-and-swap primitive, they are able to create a much more concurrent deque that is also obstruction-free. Rather than use the more conventional pointer-based approach for constructing queues, the authors elect to create the queue in contiguous memory and use special null values to identify the two queue ends. I will spare a more detailed description of the algorithm, since a description of my algorithm will cover most of the details of theirs as well.

The main strength of the Herlihy et al. deque over the Michael deque is its *concurrency between ends*. Rather than serialize entire operations on the deque, the authors serialize each modification of an individual node. This means that only two concurrent operations to the same end will conflict. A simple extension that makes the deque’s buffer circular also greatly increases *space efficiency*. The primary weakness of this deque is that it is *bounded* - it can only hold as many elements as there is space originally specified.

One additional paper, “Fast Concurrent Queues for x86 Processors” by Afek and Morrison [4], provides hope that a best-of-all-worlds deque implementation can be constructed. This paper describes a lock-free FIFO queue that is unbounded in size and also space efficient. This is achieved through “chunking” the queue into discrete cyclic arrays, which are then chained together into a linked list. Michael Scott and I believe that this chunking approach can be easily ported over to the deque space as an extension of the Herlihy et al. algorithm.

3 The Algorithm

This algorithm builds primarily off of the Herlihy et al. implementation of a bounded double-ended queue [1]. (For the sake of convenience, I will henceforth refer to this as the Herlihy deque) Therefore, I will begin with a more in-depth description of that algorithm.

3.1 Bounded Description

The Herlihy deque begins with a segment of contiguous memory as large as the maximum number of elements in the deque, plus two additional boundary nodes that mark the end of the memory region. Each entry in the region constitutes a node, which has a pointer to some value and a counter which will be described later. The region is initialized so that all nodes have one of two special null values: LN and RN. The “left half” of the region contains the LNs, and the “right half” contains the RNs. An invariant of the deque that must hold over all operations follows: the deque must consist of one or more LNs, followed by zero or more non-null values, followed by one or more RNs.

Each operation on the deque first begins with a call to an oracle function. This function returns what it believes to be the current position of either the left head or the right head. This oracle function can be implemented however one desires, as long as it 1) returns $[0, \text{SIZE} - 2]$ and $[1, \text{SIZE} - 1]$ for left and right operations respectively, and 2) is accurate in the absence of contention with other threads. I will discuss my own implementation of the oracle function in the next section, which uses hints to locate the heads.

```

type element = record val: valtype; ctr: int end
A: array[0..MAX+1] of element initially there is some k in [0,MAX]
  such that A[i] = <LN,0> for all i in [0,k]
  and A[i] = <RN,0> for all i in [k+1,MAX+1].

rightpush(v) // v is not RN or LN
  while (true) {
    k := oracle(right); // find index of leftmost RN
    prev := A[k-1]; // read (supposed) rightmost non-RN value
    cur := A[k]; // read (supposed) leftmost RN value
    if (prev.val != RN and cur.val = RN) { // oracle is right
      if (k = MAX+1) return "full"; // A[MAX] != RN
      if CAS(&A[k-1],prev,<prev.val,prev.ctr+1>) // try to bump up prev.ctr
        if CAS(&A[k],cur,<v,cur.ctr+1>) // try to push new value
          return "ok"; // it worked!
    } // end if (prev.val != RN and cur.val = RN)
  } // end while
rightpop()
  while (true) { // keep trying till return val or empty
    k := oracle(right); // find index of leftmost RN
    cur := A[k-1]; // read (supposed) value to be popped
    next := A[k]; // read (supposed) leftmost RN
    if (cur.val != RN and next.val = RN) { // oracle is right
      if (cur.val = LN and A[k-1] = cur); // adjacent LN and RN
        return "empty"
      if CAS(&A[k],next,<RN,next.ctr+1>) // try to bump up next.ctr
        if CAS(&A[k-1],cur,<RN,cur.ctr+1>) // try to remove value
          return cur.val // it worked; return removed value
    } // end if (cur.val != RN and next.val = RN)
  } // end while

```

Figure 1: Pseudo-code of the original bounded deque from the Herlihy et al. paper [1]. Left operations are completely symmetrical with right operations.

Next, a copy of the head and an adjacent element are loaded into local memory. We first check that the oracle function is correct by confirming that the following invariants hold:

- The *left head* at i must be an LN node, with the $i + 1$ node being non-LN.
- The *right head* at i must be an RN node, with the $i - 1$ node being non-RN.

If this condition is not the case, the oracle was inaccurate and we immediately retry the operation. If this condition passes, we then quickly check whether the deque is either full or empty. It is full if the head is at a boundary node; it is empty if an adjacent LN and RN are witnessed (this follows from the earlier stated deque invariant).

Finally, each push and pop performs two compare-and-swap operations. It first examines the head's adjacent node and attempts only to modify its counter, trying again immediately if the operation fails. This is necessary to avoid the ABA problem. If this operation succeeds, it tries to either push a new element to the head position or remove some element from the adjacent position.

Figure 1 shows some example code from the original Herlihy et al. paper. Note the possibility of livelock. A thread pushing right is preempted right after its first CAS. Another thread attempts to do a right pop and is then preempted after its first CAS. The first thread wakes up, fails on its second CAS, and retries, succeeding again in its first CAS. This causes the second thread to fail its second CAS, ad infinitum.

3.2 Unbounded Description

Fortunately, this deque algorithm is easily extended to provide unboundedness. The general intuition is that we can use the already “empty” boundary nodes to store pointers to other buffers. This creates a doubly-linked list of buffers that can be easily traversed to find the heads of the deque. Just like pointers to deque values, pointers to other buffers can be atomically stored in a node entry with a compare-and-swap. Please refer to figures 5 and 6 for more detailed pseudo-code of the algorithm.

The deque is structured somewhat differently from the original algorithm. Importantly, left and right hints are stored as structures containing a pointer to a buffer and an index within that buffer. These hints can also be modified atomically using compare-and-swap. This technique is used to keep these hints “fairly accurate”: the hint could still be incorrect, but an update is attempted after every successful operation.

These new features mean the algorithm is now slightly more complex. During a push, we have to consider two new cases. In the first, we are out of space to expand in the current buffer. The extension here is easy: we simply allocate and initialize a buffer of the appropriate size. Additionally, since the buffer is only visible locally to the thread, we can immediately store the new value and backpointer to the old buffer without a full read-modify-write. We then do two CAS operations to insert the pointer to the new buffer, just as we did to insert a new value. In the second case, there is space in an adjacent buffer but not the current one. We must then load the neighbor buffer’s address from the edge node and “straddle” the buffers to perform our operation. The push operation has to ensure in this case that no push happens on the neighbor buffer on top of checking for a conflicting pop.

Popping also changes slightly in this version, but also not by much. There is one extra case to consider: the right head is currently on the right edge of some buffer (or left head, left edge). In this case, we must also “straddle” buffers. Again we load the neighboring buffer via the edge node, and perform our update in the absence of a push to the neighboring buffer and pop from the current buffer.

Importantly, these new cases in the extended algorithm never require more than two compare-and-swaps. This means the extended algorithm does not increase the maximum number of read-modify-write instructions that must be performed to complete a given operation.

The oracle function I implemented for this deque uses either hint as a starting point for a full search of the structure. While this could theoretically run in time linear to the size of the deque, with “fairly accurate” hints we can avoid the search penalty in many cases. However, this only occasional penalty can be quite substantial when the deque is large enough. This issue will be discussed in later sections.

4 Properties

4.1 Unboundedness

Proving the unboundedness of this deque is trivial. Any time a null-valued boundary node is discovered, a new buffer is allocated of appropriate size and then pointed to by the old boundary node. The amount of times you can do this is bounded only by how much the memory management system is willing to cede to you.

4.2 Obstruction Freedom

I defer most of the proof of obstruction freedom to the original Herlihy et al. proof. *rightpush* is linearized by its second compare-and-swap operation, which either changes a node from RN to v or a boundary node from RN to some buffer pointer. *rightpop* is similarly linearized by its second compare-and-swap that changes some value from v to RN.

4.3 End Concurrency

As an extension built upon the Herlihy et al. algorithm, my proposed algorithm also allows for total concurrency between the two ends. The only exception to this rule is when the two heads are separated by two nodes or less. In this case, at least one of the compare-and-swap operations from each end will be conflicting. This is a very rare case, though: the deque would have at most one element in it.

4.4 Space Efficiency

Unfortunately, this algorithm as currently described is not space efficient across all access patterns. For example, a typical queue-like access pattern would keep allocating new buffers and fail to re-use old empty ones. Though I have not done so yet, I am planning to explore some variation on Maged Michael's hazard pointers to solve this problem [3]. To keep it brief, hazard pointers track which threads have references to what memory locations using a single-writer multireader protocol. Periodically, a thread will check its retired list to see if any other threads still refer to its retired pointers. If not, it can safely reuse or reclaim that memory.

5 Results

These results were obtained on Algonquin in the undergraduate network and Hylan non-major's lab. This machine has 4 cores, each a 3.30 GHz i5-2500 from Intel with 6144 KB of cache. It has 4 GB of RAM in total. The operating system is Red Hat 4.7.2-2.

My goals in these experiments were twofold. First, I wanted to demonstrate that my unbounded obstruction-free deque can perform better than a counterpart that uses mutual exclusion locks to serialize all updates to the data structure. Second, I wanted to demonstrate how different and varied access patterns can affect performance of my deque.

To prove these points, I set up an experimental framework for both locking and obstruction-free deques that allows the experimenter to control variables such as thread count, problem

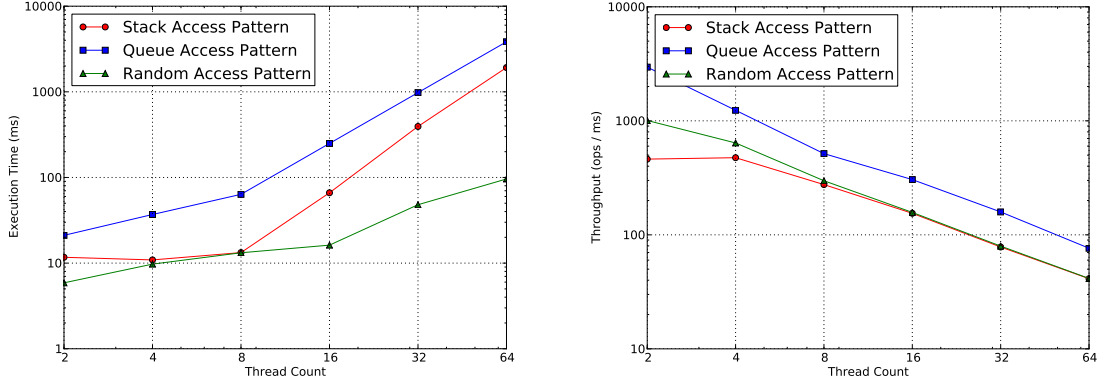


Figure 2: Execution times (left) and average throughputs (right) of common access patterns in my deque implementation. Note the logarithmic scale in the y-axis.

size, and the access pattern to test over. This framework allows for two metrics: raw execution time of the algorithm and average throughput of all threads. The former metric interprets the “problem size” variable as the number of operations to be performed by all threads and times how long it takes for all threads to complete their operations. The latter metric interprets the “problem size” as a fixed time interval and checks how many operations were performed over this period.

The three common access patterns experimented over are these: the “stack” access pattern, in which threads push and pop only to one side; the “queue” access pattern, in which threads push to one side and pop from the other; and the “random” access pattern, in which threads randomly select their next operation. These access patterns are differentiated primarily by the amount of inter-end concurrency they enable: stack enables practically none, queue enables as much as possible, and random is somewhere between the two on average.

5.1 Exclusively Nonblocking Performance

Figure 2 shows timing and throughput results for all three access patterns over an array of thread counts, with stack in red, queue in blue, and random in green. The results for stack and random are intuitive, since stack will tend to have more conflicting updates and therefore take longer to execute on average than random. I was rather perplexed by the results for queue, however. It seemed to me that this should have the best performance, if anything, and certainly not the worst.

I used the gprof tool to track the amount of time each access pattern spent in different deque operations. I found that, while stack and random would spend approximately equal amounts of time on their operations, queue was spending a disproportionate amount of time in *right_pop*, specifically *right_oracle*. It seemed like the time taken to find the head was dominating the entire algorithm.

This goes back to my previous discussion about the oracle function: mine tries to be right most of the time through “fairly accurate” hints, but can suffer huge penalties when

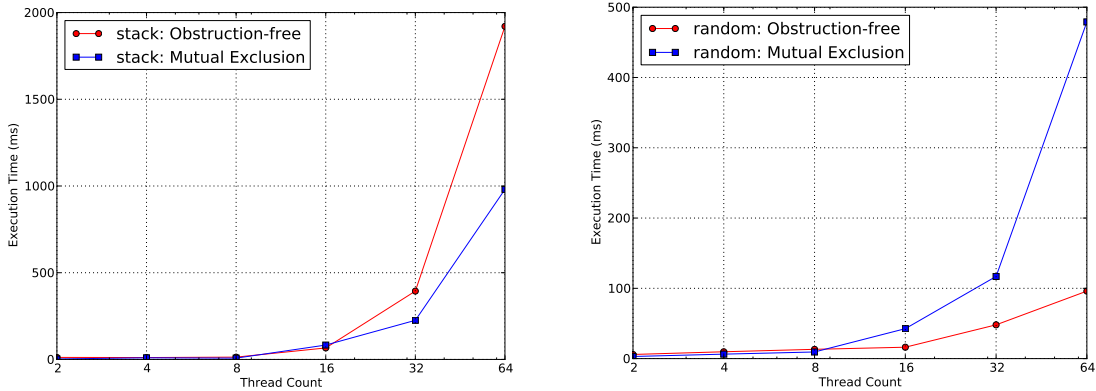


Figure 3: A comparison of execution times between locking and nonblocking dequeues in the stack access pattern (left) and random access pattern (right).

these hits are wrong. Since the queue pattern is always moving to the left, in this case, the algorithm will continue leaving more and more empty space to the right. This increases the amount of space the oracle might have to cover if its first guess is wrong.

There are two solutions to this problem. The first and most obvious is to write a more sophisticated oracle function that can make better worst-case performance guarantees. I attempted to write an oracle that inferred the direction it needed to look for the head based on the contents it currently saw (e.g. if current and previous are LN, search right). However, this venture didn't pan out in time for project completion. Another solution to this problem is to remove very old empty buffers from the deque, thus reducing the amount the oracle has to search. This relates back to my brief discussion of Michael's hazard pointers [3].

5.2 Locking vs. Nonblocking Performance

With access patterns thoroughly dissected in the previous section, it is time to move on to comparing a locking deque to my new obstruction-free deque. Since performance proved to be very bad for the queue access pattern in the previous section, I decided to focus purely on stack and random access patterns for this analysis.

Figure 3 contrasts the execution times of stack and random access patterns in both locking and obstruction-free contexts. The stack access pattern in the obstruction-free algorithm shows poorer execution time scaling than its corresponding lock-based algorithm. This is unsurprising, given that constant attempts to push or pop from the same side will force threads to step on each others' toes.

However, the plot comparing the random access pattern between lock-based and obstruction-free algorithms gives reason for hope. The random access pattern here shaves off extra milliseconds here by exploiting inter-end concurrency, which the lock-based version is not able to do, and by reducing the severity of oracle penalties by not expanding too far out.

Figure 4 also demonstrates how the throughput of the obstruction-free algorithm scales better in both the stack and random access patterns. This result may seem counter-intuitive

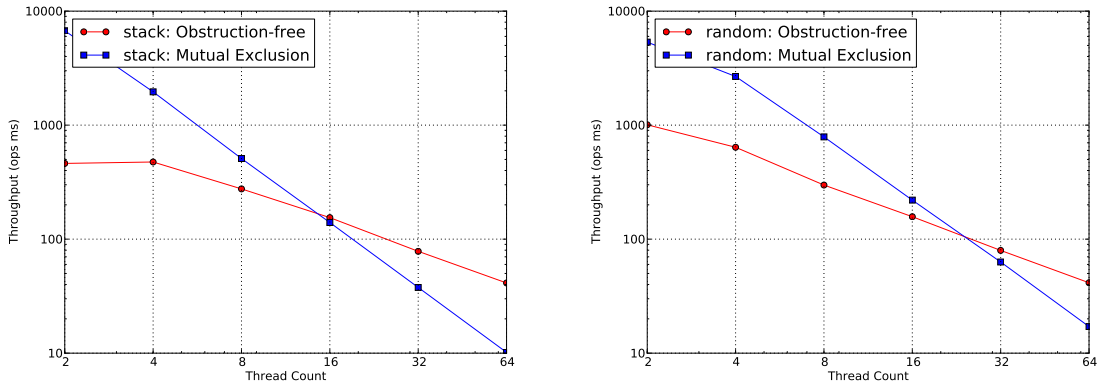


Figure 4: A comparison of throughputs between locking and nonblocking dequeues in the stack access pattern (left) and random access pattern (right).

given that obstruction-free stack access was shown to scale poorly execution-time wise. One thing to note is that the execution time’s experimental methodology ensures that there will always be an equal number of pushes and pops in the stack access pattern. In contrast, the throughput methodology makes no such guarantee, leaving threads to attempt as many operations on the deque in the time frame allotted.

This improved scaling in both cases is probably due to the well-known problem of contention for naive test-and-set locks. Since test-and-set requires all participants to write to the same location in memory, the interconnect is constantly bogged down with write traffic. When the lock is finally freed, all processors rush to grab it, and only one succeeds. This problem obviously scales quite badly with the number of participating processors.

While individual deque operations will contend with each other in the obstruction-free deque, it is rarely the case that most or all of them contend on the exact same memory location for an extended period of time. This means the chance of creating a memory “hot spot” that can seriously degrade performance is extremely low, and thus the nonblocking approach will scale much better with the number of processes. This seems to be true especially when the number of threads begins to exceed the number of processors available.

6 Future Work

As previously discussed, the most serious performance setbacks in this current algorithm are a high-penalty oracle function and poor space efficiency in certain access patterns. The former has a rather straightforward solution that I am unfortunately unable to implement myself due to time constraints. From inspecting a current incorrect hint, it is possible to figure out in which direction to search for the head, assuming that the deque is in a consistent state. Using this heuristic could greatly improve worst-case performance of the oracle. It could also be that using an entirely different scheme for hints could make writing a more performant oracle function easier.

Of course, the current oracle’s penalty would also be reduced through better manage-

ment of older empty buffers. To make the algorithm more space-efficient as well, it is possible to write an obstruction-free (or even lock-free) garbage collection and memory management module for the deque, based on Michael’s hazard pointers [3]. Threads in this system would speculatively try to release buffers that they have observed to be empty, only committing them for reuse once they are sure that no other thread has a reference to them. These reclaimed buffers could then be reused when a new buffer is called for, or have their memory freed after a certain amount of time.

References

- [1] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *In preparation*, 2003.
- [2] Maged M. Michael. Cas-based lock-free algorithm for shared deques. In *In the 9th Euro-Par Conference on Parallel Processing*, pages 651–660. Springer Verlag, 2003.
- [3] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.
- [4] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP ’13, pages 103–112, New York, NY, USA, 2013. ACM.

```

node = { void *val, int ctr };
hint = { node *nodes, int index };
deque = { hint left, hint right };

rightpush(v) // v is not RN or LN
while (true) {
    k := oracle(right); // find index of leftmost RN
    prev := k.nodes[k.index - 1]; // read (supposed) rightmost non-RN value
    cur := k.nodes[k.index]; // read (supposed) leftmost RN value
    if (prev.val != RN and (cur.val = RN or k.index = SIZE - 1)) { // oracle is right
        if (k.index == SIZE - 1 and cur.val == RN) { // need more space
            buffer = new node[SIZE]
            buffer[0].val = k.nodes
            buffer[1].val = v
            if CAS(&k.nodes[k.index - 1], prev, <prev.val, prev.ctr + 1>)
                if CAS(&k.nodes[k.index], cur, <buffer, cur.ctr + 1>)
                    update_hints(right, buffer, k.index + 3)
                    return "ok"
        } else if (k.index == SIZE - 1 and cur.val != RN) {
            buffer = cur.val
            peek = buffer[k.index + 2]
            if (peek.val != RN)
                continue
            // ensure no one pushes on neighboring right buffer + no one pops on current buffer
            if CAS(&k.nodes[k.index - 1], prev, <prev.val, prev.ctr + 1>)
                if CAS(&buffer[k.index + 2], peek, <v, peek.ctr + 1>)
                    update_hints(right, buffer, k.index + 3)
                    return "ok"
        } else {
            if CAS(&A[k-1], prev, <prev.val, prev.ctr+1>) // try to bump up prev.ctr
                if CAS(&A[k], cur, <v, cur.ctr+1>) // try to push new value
                    update_hints(right, k.nodes, k.index + 1)
                    return "ok"; // it worked!
        } // end if (prev.val != RN and cur.val = RN)
    }
} // end while

```

Figure 5: Pseudo-code of a novel unbounded deque push. Left operations are completely symmetrical with right operations.

```

node = { void *val, int ctr };
hint = { node *nodes, int index };
deque = { hint left, hint right };

rightpop()
while (true) { // keep trying till return val or empty
    k := oracle(right); // find index of leftmost RN
    cur := k.nodes[k.index - 1]; // read (supposed) value to be popped
    next := k.nodes[k.index]; // read (supposed) leftmost RN
    if (cur.val != RN and (next.val = RN or k.index = SIZE - 1)) { // oracle is right
        if (cur.val = LN and k.nodes[k.index - 1] = cur) // adjacent LN and RN
            return "empty"

        if(k.index == SIZE - 1 and next.val != RN) {
            buffer = next.value
            peek = buffer[k.index + 2]
            if(peek.val != RN)
                continue
            // ensure no one pops on current buffer + pushes to neighboring buffer
            if CAS(&buffer[k.index + 2], peek, <peek.val, peek.ctr + 1>)
                if CAS(&k.nodes[k.index - 1], cur, <RN, cur.ctr + 1>)
                    update_hints(right, k.nodes, k.index - 1)
                    return cur.val
        } else {
            if (cur.val = LN and k.nodes[k,index - 1] = cur)"; // adjacent LN and RN
                return "empty"
            if CAS(&k.nodes[k.index], next, <RN, next.ctr + 1>) // try to bump up next.ctr
                if CAS(&k.nodes[k.index - 1], cur, <RN, cur.ctr + 1>) // try to remove value
                    update_hints(right, k.nodes, k.index - 1)
                    return cur.val // it worked; return removed value
        }
    } // end if (cur.val != RN and next.val = RN)
} // end while

```

Figure 6: Pseudo-code of a novel unbounded deque pop. Left operations are completely symmetrical with right operations.