Strongly Linearizable Linked List and Queue

Steven Munsu Hwang [1]
University of Calgary, Canada

Philipp Woelfel
University of Calgary, Canada

Abstract

Strong linearizability is a correctness condition conceived to address the inadequacies of linearzability when using implemented objects in randomized algorithms. Due to its newfound nature, not many strongly linearizable implementations of data structures are known. In particular, very little is known about what can be achieved in terms of strong linearizability with strong primitives that are available in modern systems, such as the compare-and-swap (CAS) operation.

This paper kick-starts the research into filling this gap. We show that Harris's linked list and Michael and Scott's queue, two well-known lock-free, linearizable data structures, are not strongly linearizable. In addition, we give modifications to these data structures to make them strongly linearizable while maintaining lock-freedom. The algorithms we describe are the first instances of non-trivial, strongly linearizable data structures of their type not derived by a universal construction.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Strong linearizability, compare-and-swap, linked list, queue, lock-freedom

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.28

Funding We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) under Discovery Grant RGPIN-2019-04852, and the Canada Research Chairs program.

1 Introduction and Related Work

Linearizability [9] is the correctness condition of choice for asynchronous shared memory algorithms. Intuitively, it requires that an operation on a concurrent object appears to take effect instantaneously at some point (the linearization point) between the operation's invocation and response. Arranging the operations by these points must result in a sequential history that is valid respective to the object's specification. Due to this notion of "taking effect at a point in time", linearizability was considered to be practically equivalent to atomicity. In fact, atomic and linearizable objects can be interchanged without altering the worst-case behaviour of an algorithm [9]. Importantly, linearizability has been proven to be a local property [9]. Informally, a property is local if the system satisfies the property given that each object used by the system satisfies the property. Linearizability is also composable, meaning that a linearizable object implemented using atomic objects is still linearizable when the atomic objects are replaced with linearizable ones. These two properties make linearizability desirable for modular programming.

Unfortunately, linearizability is not as suitable for use in randomized algorithms: Golab, Higham and Woelfel [5] showed that the probability distributions over the set of outcomes can change when atomic objects are replaced with linearizable ones.

They proposed *strong linearizability*, which when satisfied maintains the same probability distribution over the set of outcomes as with atomic objects, under a strong adaptive adversary. In fact, strong linearizability is sufficient and necessary for that. Strong linearizability demands that future events do not change the linearization points of the past. Strong

[1] Corresponding author

25th International Conference on Principles of Distributed Systems (OPODIS 2021).
Editors: Quentin Bramas, Vincent Gramoli, and Alessia Milani; Article No. 28; pp. 28:1–28:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

linearizability is also local and composable [5], further motivating the search for such implementations.

Until now, most work on strong linearizability assumed that processes communicate only using atomic (read/write) registers. Helmi, Higham and Woelfel [7] have shown that essentially no non-trivial object has a deterministic wait-free strongly linearizable implementation from *single-writer* registers. Using *multi-writer* registers, a number of strongly linearizable *lock-free* algorithms have been devised, such as bounded max-registers [7], counters [2] and single-writer snapshots [2, 12]. On the other hand, for none of these objects, strongly linearizable *wait-free* implementations exist [2].

The impossibility result of wait-free consensus [3, 10] established that atomic read/write registers are too weak to solve fundamental shared memory problems. Even simple data structures, such as queues or stacks, have no lock-free linearizable algorithms [8]. On the other hand, so-called universal constructions, such as Herlihy's [8], show that n-process consensus objects can be used to obtain wait-free linearizable implementations of any type with a deterministic sequential specification. In fact, Herlihy's universal construction is even strongly linearizable [5].

Almost all of today's systems provide strong synchronization primitives, such as atomic compare-and-swap, which allows wait-free solutions to the consensus problem for arbitrary many processes. Therefore, all types with a deterministic sequential specification have wait-free strongly linearizable implementations (using the universal construction). But the universal construction is not practical, as it is neither space nor time efficient.

Employing strong synchronization primitives (most commonly compare-and-swap), many efficient linearizable solutions to fundamental data structure problems have been devised. But it is generally not known whether these data structures are also strongly linearizable, and thus whether they can safely be used in randomized algorithms against a strong adaptive adversary.

Essentially no efficient strongly linearizable implementations are known for fundamental data structures that require strong synchronization primitives (or at least it is not known, whether existing linearizable implementations are also strongly linearizable). This paper aims to kick-start the research needed to fill this gap. We investigate two well known standard data structures: Harris's linked list [6], and Michael and Scott's queue [11]. Both algorithms use compare-and-swap objects, and are linearizable and lock-free. We show that they are not strongly linearizable (see Section 3 for Harris's linked list and Section 5 for Michael and Scott's queue). We then show that relatively simple modifications to these data structures yield strong linearizability (see Sections 4 and 6 respectively).

1.1 Other Related Work

As mentioned earlier, most non-trivial results on strong linearizability assume that processes cannot perform strong atomic operations (only atomic reads and writes are permitted). An exception is a recent randomized implementation of a double-compare-and-swap (DCAS) object from compare-and-swap objects [4], which uses as a build block (and implements) a strongly linearizable restricted DCAS object. Moreover, Attiya, Castañeda, and Hendler [1] proved that wait-free strongly linearizable implementations of stacks and queues from "readable" base objects, require that these base objects have consensus number infinity.

2 Preliminaries

We consider a distributed shared memory system with n processes communicating through shared objects. Each shared object has a *type*, which outlines a set of operations, and is defined by a *sequential specification*, a set of valid sequences of operations. An operation *op* consists of an *invocation event*, denoted $inv(op)$, and possibly a matching *response event* denoted $rsp(op)$.

A *transcript* is a sequence of invocation and response events of operations. For transcripts T and Y, we denote $T \circ Y$ as the concatenation of the two transcripts. A *projection* of a transcript T onto a process p is denoted $T|p$, and is the sequence of invocation and response events by p in T. A projection of T onto an object O, denoted as $T|O$, is the sequence of invocation and response events of operations in T performed on O. An operation *op* is *complete* in some transcript T if T contains its invocation and matching response. A transcript is complete if every operation in the transcript is complete. In a transcript, an operation *op* is *atomic* if $rsp(op)$ immediately follows $inv(op)$. An object O is atomic if every operation on O in any transcript is atomic. On the other hand, an object O is *implemented* if each operation *op* on O is a method using other implemented or base objects (objects provided by the system). Formally, a method is associated with a sequence of operations such that when *op* is called, the sequence of operations are executed.

A transcript T defines a *happens before* order $\xrightarrow{T}$ for operations $op_1, op_2 \in T$, where $op_1 \xrightarrow{T} op_2$ if and only if $rsp(op_1)$ occurs before $inv(op_2)$ in T. Note that this is a partial order.

A *history* is a transcript where for every process p, every operation in $H|p$ is atomic. A history S is *sequential* if every operation in S is atomic. A sequential history S is *valid* if and only if for any object O, $H|O$ is in the sequential specification of the type of O. For every incomplete operation in H, if either the operation is discarded, or a response is appended, we obtain a complete history H' called a *completion* of H.

An *interpreted history* $\Gamma(T)$ can be derived from a transcript detailing an algorithm execution using an implemented object O. It is obtained by iterating through all p, and removing all events by p after $inv(op)$ and not after its matching response $rsp(op)$ for any operation *op* by p. Intuitively, the interpreted history consists of invocation and response events of "high level" operations in T; all intermediate steps for methods are removed from the transcript. For a set of transcripts $\mathcal{T}$, $\Gamma(\mathcal{T}) = \{\Gamma(T) \mid T \in \mathcal{T}\}$.

Consider H', a completion of a history H. A *linearization* [9] of H' is a sequential history S satisfying all of the following:

- All operations in H' are in S.
- For all operations op_1 and op_2 in H, if $op_1 \xrightarrow{H'} op_2$, then $op_1 \xrightarrow{S} op_2$.
- S is valid.

For an implementation of a shared object to be *linearizable*, every possible history on the object must have a linearizable completion. A function f, mapping each history H from a set $\mathcal{H}$ of histories to a linearization $f(H)$ of H, is called *linearization function* for the set $\mathcal{H}$.

Given a transcript T, if an event e is the t-th element of T, then we say that e occurs at time t in T, or that $time_T(e) = t$. If e is not present in T, then we define $time_T(e) = \infty$. For an atomic operation *am* in a transcript T, we say that *am* occurs at $time_T(rsp(am))$. If an implemented operation *op* performs an atomic operation on line x of the method corresponding to the operation, we refer to this atomic operation as op_x. If *op* is complete in a transcript T, then by $time_T(op_x)$, we refer to the last time at which op_x is executed during *op*.

Linearizability can also be expressed through *linearization points*. Consider a transcript T, and a linearizable object O. A *linearization point function pt* for O maps $op \in \Gamma(T|O)$ to ∞ or a time in T such that

1. $pt(op) \in [time_T(inv(op)), time_T(rsp(op))]$, and
2. there is a valid sequential history S of $\Gamma(T|O)$ where for all $op_1, op_2 \in S$, if $op_1 \xrightarrow{S} op_2$ then $pt(op_1) \leq pt(op_2)$. (This property ensures that S preserve the happens-before-order

of T. It is possible to have $pt(op_1) = pt(op_2)$: this simply means that op_1 and op_2 can appear in S in either relative order without violating the happens-before-order.)

We call $pt(op)$ the linearization point of op. Intuitively, this is the point in time where operation op "appears" to take effect. For a set of transcripts $\mathcal{T}$, the *prefix closure* of $\mathcal{T}$ is the set containing all prefixes of transcripts in $\mathcal{T}$. We denote the prefix closure of $\mathcal{T}$ as $close(\mathcal{T})$. A function $f : \mathcal{T} \to \mathcal{T}'$, where $\mathcal{T}$ and $\mathcal{T}'$ are sets of transcripts, is *prefix preserving* if for any two transcripts $T, Y \in \mathcal{T}$ where T is a prefix of Y, $f(T)$ is a prefix of $f(Y)$.

A function f is a *strong linearization* [5] function for a set of transcripts $\mathcal{T}$ if:

- f is a linearization function for $\Gamma(close(\mathcal{T}))$, and
- f is prefix preserving.

An implemented object O is *strongly linearizable* if and only if the set of transcripts on O has a strong linearization function.

A method of an implemented object is *lock-free* if it guarantees that if a process executing a method takes infinitely many steps, then infinitely many method calls finish within a finite number of steps. An object is lock-free if every operation on the object is lock-free. An implemented object is *wait-free* if every method call terminates after taking finitely many steps.

Our algorithms use an atomic compare-and-swap object, which has an operation denoted as *CAS*. The operation takes two arguments: *old* and *new*. If the value of the object is equal to *old*, then the operation overwrites the value of the object to *new*. Otherwise, the operation has no effect. In addition, the object also allows reads and writes.

3 Harris's linked list is not strongly linearizable

Nodes in Harris's linked list implementation have the following fields: *key* and *succ*. Field *key* stores the value of the node, and is taken as the argument by the node's constructor. Once set, the value of *key* never changes for a particular node. The successor field, *succ*, is a *CAS* object which contains *next*, the next node in the list, and *marked*, a boolean value to indicate whether the node has been "logically deleted". A node is marked before being excised ("physically deleted") from the linked list. As a shorthand, we access different parts of the *succ* field of a node v by $v.next$ or $v.marked$. The successor field is initialized to $(null, false)$.

There are two shared variables $Head$ and $Tail$, which represent the head and tail sentinel nodes of the list. $Head$ has a key value of $-\infty$, and the $Tail$ has a key value of ∞. Initially, $Head$ and $Tail$ are the only nodes in the linked list, where $Head.succ = (Tail, false)$ and $Tail.succ = (null, false)$.

The sequential specification of the linked list we consider is as follows. The linked list consists of three methods: $delete$, $insert$ and $find$. All three operation return a boolean value to designate whether the operation has failed or succeeded. The nodes are sorted by their keys, and all keys in the linked list are unique. An $insert$ operation fails if the key being inserted is already in the linked list, and otherwise it succeeds. Likewise, a $delete$ operation fails if the key being deleted is not in the linked list. The return value of $find$ indicates whether a key is in the linked list.

Harris's linked list uses helper function $search(search\_key)$, which returns nodes $left$ and $right$ with the following guarantees: at some point in time during the execution of $search$,

1. $left.key < search\_key \leq right.key$,
2. $left$ and $right$ are unmarked and

3. $left.next = right$.

The *search* method is used to locate the nodes of interest for each operation of the linked list. For example, a successful *insert* adds a new node between $left$ and $right$ returned by *search*. It is also used to verify existence of keys in the linked list. Since the keys are in sorted order, at points in time when the search conditions are met, $left$ contains the largest key less than *search_key* and *right* contains the smallest key less than or equal to *search_key*. If $right.key = search\_key$ then the linked list contains *search_key* at a point in time when the search conditions held; if $right.key \neq search\_key$, then the linked list does not contain *search_key* at such a point in time. These are precisely the points when $find$, a failed *delete* and a failed *insert* should linearize.

Recall that, intuitively, strong linearizability requires that future events do not affect the linearization order of the past. That is, events that occur after a linearization point should not affect the position of that linearization point in a history. However, at the point in time when the search conditions hold, Harris's linked list does not guarantee which nodes will be returned by *search*. In other words, if time t is when the search conditions are true, it is possible to change the history after t (i.e. change the "future") such that $search(search\_key)$ returns a different pair of nodes, which can change the response of the operation invoking *search*. This suggests that the linked list is not strongly linearizable. By altering the response of an operation through future events, the linearization order of the past will likely need to change to maintain validity. We use this observation in our proof of Lemma 1.

Note that a successful *insert* linearizes when a new node is inserted by a successful *CAS* on line 61. Likewise, a successful *delete* linearizes when a node is marked by a successful *CAS* on line 45. These linearization points are already strongly linearizable; events that occur after a *CAS* cannot influence whether the *CAS* succeeds. Thus our modifications in the next section focus on the *search* function.

▶ **Lemma 1.** *The linked list implementation by Harris (Figure 1) is not strongly linearizable.*

A full proof of the lemma is provided in Appendix A. The high level idea is as follows.

Let f be a linearization function for the linked list. We denote $node_i$ as the node containing key i, $in_p(x)$ as a transcript of an insert of key x by process p and $del_p(x)$ as a transcript of a delete of key x by p.

Consider the following transcripts for processes p and q:

$$S = in_p(3) \circ (del_q(2) \text{ to the first execution of line 15}) \circ in_p(2)$$
$$T_1 = S \circ del_p(3) \circ (del_q(2) \text{ from line 16 to completion})$$
$$T_2 = S \circ (del_q(2) \text{ from line 16 to completion})$$

Note that in S, the $search(2)$ call in $del_q(2)$ will return $Head$ and $node_3$ if $node_3$ is unmarked when line 16 is executed. Otherwise, *search* will restart.

Here transcripts T_1 and T_2 have the same prefix (the "past") S, with the only difference (in the "future") being that T_1 has $del_p(3)$ before q finishes $del_q(2)$. In T_2, when $del_q(2)$ continues to completion, $node_3$ is unmarked and is returned as *right*. Thus $del_q(2)$ fails in T_2 ($search\_key < right.key$), and must be ordered before $in_p(2)$ in $f(T_2)$ to preserve validity. However in T_1, $del_p(3)$ marks $node_3$ and the $search(2)$ call in $del_q(2)$ restarts its traversal. Eventually $search(2)$ returns $Head$ and $node_2$. In this case, $del_q(2)$ succeeds and $del_q(2)$ must be ordered after $in_p(2)$ in $f(T_1)$ to preserve validity. Then f cannot be a strong linearization function since $del_q(2)$ must be ordered before $in_p(2)$ for $f(S)$ to be a prefix of $f(T_1)$, but then $f(S)$ is not a prefix of $f(T_2)$. Transcript S is a prefix of both T_1 and T_2, but $f(S)$ is not a prefix of $f(T_2)$ in this case.

```
Function search(search_key):
    search_again
    while true do
        curr ← Head
        (curr_next, curr_marked) ← Head.succ
        repeat
            if not curr_marked then
                left ← curr
                left_next ← curr_next
            curr ← curr_next
            if curr = Tail then
                break
            (curr_next, curr_marked) ← curr.succ
        until curr_marked or curr.key < search_key
        right ← curr
        if left_next = right then
            if right ≠ Tail and right.succ.marked then
                goto search_again
            else
                return (left, right)
        if left.succ.CAS(left_next, right) then
            if right ≠ Tail and right.succ.marked then
                goto search_again
            else
                return (left, right)

Function find(search_key):
    left, right ← search (search_key)
    if right = Tail or right.key ≠ search_key then
        return false
    else
        return true

Function delete(search_key):
    while true do
        left, right ← search (search_key)
        if right = Tail or right.key ≠ search_key then
            return false
        end
        (right_next, right_marked) ← right.succ
        if not right_marked then
            if right.succ.CAS((right_next, false), (right_next, true)) then
                break
            end
        end
    end
    if not left.succ.CAS((right, false), (right_next, false)) then
        left, right ← search (right.key)
    end
    return true

Function insert(search_key):
    new_node ← new Node(search_key)
    while true do
        left, right ← search (search_key)
        if right ≠ Tail and right.key = search_key then
            return false
        end
        new_node.succ ← (right, false)
        if left.succ.CAS((right, false), (new_node, false)) then
            return true
        end
    end
```

Figure 1 Harris's Linked List.

4 A strongly linearizable linked list

In this section we describe modifications to Harris's linked list to yield a strongly linearizable variant. The modified algorithm (Figure 2) uses the same node object as in Harris's algorithm. In addition, the list elements are still sorted by their keys, and the list uses $Head$ and $Tail$ sentinels in the same manner as the original.

The largest modification can be seen in the *search* method. The changes guarantee different search conditions: at the last shared memory step when executing *search*,

1. $left.key \leq search\_key < right.key$,
2. $left$ is unmarked, and
3. $left.next = right$.

Recall that in Harris's list the condition guaranteed that $left.key < search\_key \leq right.key$ and that $right$ is also unmarked. Similar to Harris's implementation, if $left$ is returned with $left.key = search\_key$, then the linked list contains $search\_key$ when the search conditions are true; if $left.key < search\_key$ then the linked list does not contain $search\_key$ when the search conditions are true.

Since $search$ can only exit on line 74, the last shared memory step in $search$ is either line 67 or line 77, when $curr.succ$ is read. If $search$ exits, we know that $left.key \leq search\_key < right.key$ (by line 72) and this was true at the last shared memory step (since key does not change). In addition, we know that $left = curr$ was unmarked at the last shared memory step (by line 73), and that $right = curr\_next$ was adjacent to $left$. Thus the search conditions are true. Observe that at every execution of line 67 or line 77, it is known whether it is the last shared memory step; when $curr.succ$ is read, all values used in the exit conditions for $search$ are known. Thus, events after the last shared of memory step of $search$ do not influence which nodes are returned. This is the crux of why the new implementation is strongly linearizable.

The other methods are nearly identical to Harris's counterparts; the methods check whether a key is in the linked list by looking at the $left$ node. One noteworthy change is that $SLdelete$ no longer attempts to excise the marked node. This is because $left$ is now the node to delete, and the predecessor of $left$ is not readily available to "swing" the pointer to $right$.

We define the following terms to use in the proofs below. Let T be a transcript containing operations on O, an implementation of the algorithm in Figure 2. At time t, we say that a node v is reachable if either $v = Head$, or there exists a reachable node u such that $u.next = v$. A node v is *pre-inserted* if it was initialized in line 88 of $SLinsert$, but has not been an argument of a successful CAS operation in line 94.

For a transcript T containing operations on O, we say that at time t, the *interpreted value* of O is the sorted sequence of keys of all unmarked, reachable nodes excluding $Head$ and $Tail$. Intuitively, the interpreted value describes the keys that are currently "in" the linked list. To prove strong linearizability, we will show that any operation that linearizes at time t should behave as if it is acting on a linked list with the keys in the interpreted value at t (Lemmas 7- 9). In addition, we show that the interpreted value at time t is consistent with the operations that have linearized before t (Lemma 10).

For the proof of strong linearizability, we assume without loss of generality that an operation op responds at the time of its last shared memory operation, i.e. if line x of op is the last shared memory operation, $time_T(op_x) = rsp(op)$. Note that the response of an operation is uniquely determined by the time of its last shared memory operation.

We define the function $pt(op)$ for any operation op in a transcript T on a linked list outlined in Figure 2 in the following way:

1. If op is a successful $SLinsert$ operation, then $pt(op)$ is the time at which the CAS operation in $SLinsert$ succeeds. That is, $pt(op) = time_T(op_{61})$.
2. If op is a successful $SLdelete$ operation, then $pt(op)$ is the time at which the CAS operation in $SLdelete$ succeeds. That is, $pt(op) = time_T(op_{45})$.
3. If op is a failed $SLinsert$ or $SLdelete$, or an $SLfind$ operation, then $pt(op) = rsp(op)$ (i.e. at its last shared memory step).
4. Otherwise, op is pending in T and $pt(op) = \infty$.

```
Function search(search_key):
65  while true do
66      curr ← Head
67      (curr_next, curr_marked) ← Head.succ
68      while true do
69          if not curr_marked then
70              start ← curr
71              start_next ← curr_next
72          if curr.key ≤ search_key < curr_next.key then
73              if not curr_marked then
74                  return (curr, curr_next)
75              break
76          curr ← curr_next
77          (curr_next, curr_marked) ← curr.succ
78      while curr_marked do
79          curr ← curr_next
80          (curr_next, curr_marked) ← curr.succ
81      start.succ.CAS((start_next, false), (curr, false))
```

```
Function SLinsert(search_key):
88  new_node ← Node(search_key)
89  while true do
90      left, right ← search(search_key)
91      if left.key = search_key then
92          return false
93      new_node.succ ← (right, false)
94      if left.succ.CAS((right, false), (new_node, false)) then
95          return true
```

```
Function SLdelete(search_key):
96  while true do
97      left, right ← search(search_key)
98      if left.key ≠ search_key then
99          return false
100     if left.succ.CAS((right, false), (right, true)) then
101         return true
```

```
Function SLfind(search_key):
102 left, right ← search(search_key)
103 if left.key ≠ search_key then
104     return false
105 else
106     return true
```

Figure 2 A Strongly linearizable linked list.

For any complete $SLinsert$, $SLdelete$ or $SLfind$ operation op, note that $pt(op)$ corresponds to the execution of an atomic operation in op. Thus if $op_1, op_2 \in T$, $pt(op_1) \neq \infty$, $pt(op_2) \neq \infty$ and $op_1 \neq op_2$, then $pt(op_1) \neq pt(op_2)$. Also note that $pt(op) \in [time_T(inv(op)), time_T(rsp(op))]$.

Let $\mathcal{T}$ be the set of all transcripts on an implementation of the algorithm in Figure 2. For all $T \in \mathcal{T}$, define a sequential history $f(T)$ such that for all $op_1, op_2 \in \Gamma(T)$ with $pt(op_1) \neq \infty$ and $pt(op_2) \neq \infty$, $op_1 \xrightarrow{f(T)} op_2$ if and only if $pt(op_1) < pt(op_2)$. By the above observation that two different operations map to different times by pt, the history $f(T)$ is unambiguous.

The following four claims show that the invariants (e.g. the linked list is always sorted) maintained by Harris's implementation hold for the modified implementation as well. The proofs of these lemmas are postponed to Appendix B.

▶ **Lemma 2.** *A marked node's succ field never changes.*

▶ **Lemma 3.** *Keys are strictly sorted; For any two nodes v_1 and v_2, if $v_1.next = v_2$ then $v_1.key < v_2.key$.*

▶ **Corollary 4.** *The linked list never contains duplicate keys.*

▶ **Lemma 5.** *All unmarked, not pre-inserted nodes are reachable.*

▶ **Lemma 6.** *Consider a search call that returns and let t be the last time line 67 or line 77 is executed. If $curr.key < search\_key < curr\_next.key$ at t, then the interpreted value does not contain $search\_key$ at t. Otherwise, if $curr.key = search\_key$, the interpreted value contains $search\_key$ at t.*

Proof. Since $search$ returns, $curr\_marked = false$ was read on time t. Suppose $curr.key < search\_key < curr\_next.key$. Since $curr$ is unmarked at t, by Lemma 5, it is reachable and $curr\_next$ is also reachable. To show a contradiction, suppose that $search\_key$ is in the interpreted value at time t. Consider the sequence of nodes

$$v_1, \dots, v_k, v_{k+1}, \dots v_m$$

where $v_1 = Head$, $v_k = curr$, $v_{k+1} = curr\_next$, $v_m = Tail$ and $v_i.next = v_{i+1}$ for all $i < m$. The sequence contains all reachable nodes at time t, thus $i \notin \{k, k+1\}$ exists such that $v_i.key = search\_key$. However, if such an i existed then the linked list is not strictly sorted and Lemma 3 is violated.

Now suppose that $curr.key = search\_key$. Then the interpreted value at t contains $search\_key$ since $curr$ is unmarked. ◀

▶ **Lemma 7.** *A $SLfind(k)$ operations fails if and only if the interpreted value does not contain k at $pt(SLfind(k))$.*

Proof. A $SLfind$ fails if $left.key \neq search\_key$, and we know that either $left.key = search\_key$ or $left.key < search\_key < right.key$. Then at $pt(SLfind(k))$ the interpreted value does not contain k by Lemma 6.

For the converse, $SLfind$ succeeds if $left.key = search\_key$. Similar to above, Lemma 6 implies that the interpreted value contains k at $pt(SLfind(k))$. ◀

▶ **Lemma 8.** *A $SLdelete(k) = op$ operation fails if and only if the interpreted value does not contain k at $pt(op)$.*

Proof. When op fails, by the same reasoning as in the proof of Lemma 7, the interpreted value does not contain k at $pt(op)$.

Suppose op succeeds, meaning $pt(op) = time_T(op_{100})$, and the CAS on line 100 succeeds. This implies that $left$ is unmarked at $time(op_{100})$, therefore $left.key$ is in the interpreted value at this time. Since $left.key = k$, the interpreted value contains k. ◀

▶ **Lemma 9.** *An $SLinsert(k) = op$ operation fails if and only if the interpreted value contains k at $pt(op)$.*

Proof. When op fails, by the same reasoning as in the proof of Lemma 7, the interpreted value contains k at $pt(op)$.

Suppose op succeeds, meaning $pt(op) = time(op_{94})$, and the CAS on line 94 succeeds. This implies that $left.succ = (right, false)$ at $time(op_{94})$, therefore both $left$ and $right$ are reachable at this time. By Lemma 2 the interpreted value does not contain k. ◀

▶ **Lemma 10.** *The interpreted value contains k at time t if and only if there exists a successful insert $SLinsert(k) = op_{in}$ such that $pt(op_{in}) < t$ and no successful delete $SLdelete(k) = op_{del}$ exists such that $pt(op_{in}) < pt(op_{del}) < t$.*

Proof. Suppose op_{in} with $pt(op_{in}) < t$ exists such that no delete op_{del} exists with $pt(op_{in}) < pt(op_{del}) < t$. By Lemma 5, the interpreted value contains k after $pt(op_{in})$. To show a contradiction, suppose that the interpreted value at t does not contain k. A node is not in the interpreted value if it is not reachable, or it is marked. However, only marked nodes are unreachable (when it is not pre-inserted), thus the node containing k must have been marked between $pt(op_{in})$ and t. However, nodes are only ever marked when the CAS on line 100 succeeds, with $left.key = k$. Such a successful CAS corresponds to a op_{del} operation with $pt(op_{in}) < pt(op_{del}) < t$, yielding a contradiction.

To show the converse, first suppose that no successful $SLinsert(k) = op_{in}$ operation exists such that $pt(op_{in}) < t$ in T. It is clear that the interpreted value does not contain k at t by parsing the code; the only method which initializes a new node with $search\_key$ is $SLinsert$, and only a successful CAS on line 94 will make the node reachable. Now suppose that there exists a successful op_{del} such that $pt(op_{in}) < pt(op_{del}) < t$ for any successful $SLinsert(k)$ operation op_{in}. At $pt(op_{del})$, a reachable, unmarked node with key k is marked. There is only one such node at $pt(op_{del})$ by *Corollary 4*. To show a contradiction, suppose that at t, the interpreted value contains k; a reachable, unmarked node with key k exists. The interpreted value does not contain k immediately after $pt(op_{del})$, thus a new node was inserted by a successful CAS in $SLinsert$ in $(pt(op_{del}), t)$. However, such a CAS corresponds to a successful $SLinsert$ operation with $search\_key = k$. ◀

▶ **Theorem 11.** *The linked list implementation in Figure 2 is strongly linearizable; f is a linearization function for O, and f is prefix preserving.*

Proof. For an operation op on O, Lemmas 7, 8 and 9 show that op responds in a way that is consistent with the interpreted value of O at $pt(op)$. By Lemma 10, at any $pt(op)$, the interpreted value contains k if and only if a successful $SLinsert(k)$ linearized before $pt(op)$ with no successful $SLdelete(k)$ that linearized between $pt(op)$ and the insert. Therefore, $f(T)$ is a linearization of the interpreted history $\Gamma(T)$.

Consider step t of T and operation $op \in \Gamma(T)$ where $pt(op) = t$. Then

1. operation op is a successful $SLinsert$ operation and t is when a successful CAS on line 94 is executed
2. operation op is a successful $SLdelete$ operation and t is when a successful CAS on line 100 is executed
3. operation op is either a $SLfind$ operation, a failed $SLinsert$ operation, or a failed $SLdelete$ operation and t is when op last executes line 67 or line 77. It is completely determined by step t whether t is the last execution of line 67 or line 77; all values used in the exit condition of *search* on lines 72 and 73 are known by t. Furthermore, the values used in the exit conditions for a failed $SLinsert$ (line 98) and a failed $SLdelete$ (line 91) are known by t.

At step t it is determined what operation op satisfies $pt(op) = t$. Therefore, if S is a prefix of T, then $f(S)$ is a prefix of $f(T)$. ◀

We prove that the algorithm in Figure 2 is lock-free. For any operation op, op finishes within a finite number of steps after $pt(op)$. Therefore, it suffices to show that if a process p takes infinitely many steps during a method call, then infinitely many operations have linearized.

The *succ* field of a reachable node is only changed by a CAS operation. We will call such successful CAS operations an *update* to the linked list. Note that the CAS in *search* may succeed, but if $start\_next = curr$ then $start.succ$ does not change and this is not an update. A successful CAS in *search* is an update if marked nodes were made unreachable by the operation. Thus the number of updates by *search* is upperbounded by the number of marked nodes, i.e. the number of successful $SLdelete$ that have linearized. A successful $SLinsert$ does a single update, and unsuccessful $SLinsert$ and $SLdelete$ do not update the linked list.

▶ **Lemma 12.** *The search method is lock-free.*

We prove this lemma in Appendix B.

▶ **Theorem 13.** *The linked list implementation in Figure 2 is lock-free.*

Proof. It is clear that since *search* is lock-free by Lemma 12, $SLfind$ is lock-free.

Without loss of generality, consider a $SLdelete$ execution that lasts at least k iterations (of the loop in $SLdelete$). For every iteration, $left.key \neq search\_key$ and the CAS in $SLdelete$ must have failed. However $left.succ = (right, false)$ when last read in $search$. Thus an update occurred between when $left.succ$ was read and CAS failed. Then at least $k/2$ successful $SLinsert$ or $SLdelete$ have linearized. This implies that if infinitely many steps are taken by a process executing $SLdelete$, then infinitely many successful $SLinsert$ or $SLdelete$ have linearized. ◀

5 Michael and Scott's queue is not strongly linearizable

Michael and Scott's queue [11] is a linked list based algorithm. The node object consists of two fields; $value$ and $next$, where $next$ is a pair containing a node (the next node in the linked list) and a sequence number. The $value$ contains the element that was enqueued, and the $next$ field is a CAS object. The queue maintains $Head$ and $Tail$ CAS objects which are both initialized to $(v_{dummy}, 0)$, where v_{dummy} is a dummy node. The sequence numbers are present to prevent the ABA problem, but for brevity we will commonly refer to $Head$, $Tail$ and the $next$ field as if they refer to nodes, instead of a node-sequence number pair.

At a high level, enqueued elements are appended to the $Tail$, and the $Head$ is set to $Head.next$ to dequeue elements. The $Head$ refers to the last element that was dequeued (hence the dummy node) to simplify cases when the queue is empty. The queue also prevents $Tail$ from lagging behind $Head$. This ensures that freeing a dequeued node (by the call to $free$ on line 130) does not corrupt the data structure.

The linearization point of *enqueue* is when a new node is successfully appended to the list (at CAS success on line 113). For a successful *dequeue*, it is when $Head$ changes to $Head.next$ (line 129); for a failed *dequeue* it is when *null* was found when reading *start.next* (line 121).

The linearization points for *enqueue* and successful *dequeue* are already strong linearization points; similar to successful *insert* and *delete* for Harris's implementation, they correspond to to a successful CAS, after which the methods return. However, the linearization point of a failed *dequeue* operation is not a strong linearization point. If $Head$ was changed between the execution of line 121 (the linearization point) and line 122, then the *dequeue* restarts and may no longer fail. Similar to a failed *delete* in Harris's linked list, events after the linearization point can change the response of the operation. We have only examined one particular linearization point for a failed *dequeue*, but this observation can be extended to prove that the implementation is not strongly linearizable similar to the proof of Lemma 1. We postpone the proof the next lemma to Appendix C.

▶ **Lemma 14.** *Michael and Scott's queue (Figure 3) is not strongly linearizable.*

6 A strongly linearizable queue

As previously stated, Michael and Scott's queue is not strongly linearizable only because the linearization point for a failed *dequeue* is not strongly linearizable. The problem was that because of the condition on line 122, events after the linearization point could change the response of a failed *dequeue* operation.

```
    Function enqueue(x):
107     node ← new Node(x)
108     while true do
109         (end, endc) ← Tail
110         (next, nextc) ← end.next
111         if (end, endc) = Tail then
112             if next = null then
113                 if end.next.CAS((next, nextc),
                     (node, nextc + 1)) then
114                     break
115             else
116                 Tail.CAS((end, endc), (next,
                     endc + 1))
```

```
117 Function dequeue():
118     while true do
119         (start, startc) ← Head
120         (end, endc) ← Tail
121         (next, nextc) ← start.next
122         if (start, startc) = Head then
123             if start = end then
124                 if next = null then
125                     return false
126                 Tail.CAS((end, endc), (next,
                     endc+1))
127             else
128                 value ← next.value
129                 if Head.CAS((start, startc),
                     (next, startc+1)) then
130                     free(start)
131                     return true
```

Figure 3 Michael and Scott's lock-free queue.

A simple modification that will yield a strong linearizable queue is to remove the condition on line 122. The linearization point remains the same; it is when *null* is read on line 121. Intuitively, if *null* is read then *start* refers to the last node, and so should *Head* and *Tail* (thus *start* = *end*). This means that the method commits to failing exactly when *null* is read, and at this point the queue is empty (recall that *Head* points to the last element dequeued). Not checking whether *Head* changed since its last read (line 122) will not corrupt the queue since if *Head* changed, the *CAS* on line 129 will fail. In our proof that the algorithm in Figure 4 is strongly linearizable, we disregard line 157 (the *free* function call). Thus, if the method exits on line 158, the *CAS* on line 156 is the last shared memory operation. Calling *free* does not affect strong linearizability. For the proofs below, we assume that no ABAs occur due to our use of sequence numbers. Let T be a transcript containing operations on O, an implementation of the queue. We define whether a node is reachable identically as with the linked list; a node is reachable at time t if it can be obtained by traversing the

```
    Function dequeue():
146     while true do
147         (start, startc) ← Head
148         (end, endc) ← Tail
149         (next, nextc) ← start.next
150         if start = end then
151             if next = null then
152                 return false
153             Tail.CAS((end, endc), (next, endc+1))
154         else
155             value ← next.value
156             if Head.CAS((start, startc), (next, startc+1)) then
157                 free(start)              // ignored in the proof of strong linearizability
158                 return true
```

Figure 4 Dequeue operation of a strongly linearizable lock-free queue.

sequence of nodes from $Head$ (let no node be reachable if $Head = null$). We say that $Head$ (or $Tail$) is *incremented* if $Head = (v, \_)$ is changed to $(v.next, \_)$, where v is a node, and $v.next \neq null$. Suppose at time t, we have the following sequence of nodes

$$v_1, \ldots, v_k$$

where $Head = v_1$, $v_{i-1}.next = v_i$ for $i \in \{2, \ldots, k\}$ and $v_k.next = null$. The sequence is well defined if $Head \neq null$, (guaranteed by Lemma 16). We define the interpreted value of O at time t as the following sequence of numbers:

$$v_2.value, \ldots, v_k.value.$$

Once again, we assume without loss of generality an operation *op* responds at its last shared memory operation.

The linearization function $pt(op)$ for an operation *op* in T on an implementation of the queue in Figure 4 (with *enqueue* from Figure 3 is defined as follows:

1. If *op* is an *enqueue* operation, then $pt(op)$ is the time at which the CAS on line 113 succeeds.
2. If *op* is a *dequeue* operation, then $pt(op)$ is the first time at which *null* is read on line 149 or the CAS on line 156 succeeds.
3. Otherwise, *op* did not perform its last shared memory step and $pt(op) = \infty$.

Let $\mathcal{T}$ be the set of all transcripts on an implementation of the queue in Figure 4. For all $T \in \mathcal{T}$, we define a sequential history $f(T)$ that orders operations according to *pt*, and excludes all operations *op* with $pt(op) = \infty$. That is, for $pt(op_1) \neq \infty$ and $pt(op_2) \neq \infty$, $op_1, op_2 \in T$, $op_1 \xrightarrow{f(T)} op_2$ if and only if $pt(op_1) < pt(op_2)$. Again, $f(T)$ is unambiguous since for every operation $op \in \Gamma(T)$ such that $pt(op) \neq \infty$, the step of T at $pt(op)$ is performed by *op*.

The following two lemmas describe invariants of the queue which are used to argue strong linearizability. Their proofs can be found in Appendix D.

▶ **Lemma 15.** *$Tail$ is always reachable.*

▶ **Lemma 16.** *$Head$ is never null, and is only ever incremented.*

▶ **Lemma 17.** *Suppose the interpreted value of the queue is $(x_1, \ldots, x_k)$ at a CAS call on line 113. Let t be the time at which this CAS call occurs. If the CAS call succeeds, then the interpreted value immediately after t is $(x_1, \ldots, x_k, value)$, where value is the argument of enqueue.*

Proof. Suppose the CAS operation on line 113 succeeds, meaning $end.next = null$ at t. By Corollary 23, $Tail.next$ is also *null* when it was assigned to *end*. Since $Tail$ is only ever incremented, and $Tail.next = null$ up until the CAS operation, $Tail$ and *end* refer to the same node at t. By Lemma 15 *end* is a reachable node. Since $end.next = null$, *end* is the last reachable node by Observation 24. Thus the interpreted value immediately after t is $(x_1, \ldots, x_k, value)$. ◀

▶ **Lemma 18.** *Suppose that at time t, $start.next = null$ is read on line 121. Then the interpreted value of the queue at t is empty.*

Proof. This follows immediately from Lemma 16 and Corollary 23; since $start.next = null$ and $Head$ is only ever incremented, $Head$ cannot have changed between when it was assigned to *start* and when $start.next$ was read. ◀

▶ **Lemma 19.** *If $start.next = null$ was read on line 149 then $start = end$.*

Proof. As seen in the proof of Lemma 18, $Head$ and $start$ reference the same node when $start.next = null$ was read. Since $Tail$ is always reachable (Lemma 15) and $Head$ references the last reachable node, $Tail$ references the same node as $Head$ during the execution of lines 147 and 148. Thus when $Tail$ is read on line 148, it references the same node as $start$. ◀

▶ **Lemma 20.** *Suppose at time t the interpreted value of the queue is $(x_1, \ldots, x_k)$, and a successful CAS on line 156 is executed. Then immediately after time t, the interpreted value of the queue is $x_2, \ldots, x_k$.*

Proof. By Lemma 16, upon CAS success the $Head$ changes to $head.next$. ◀

▶ **Theorem 21.** *The queue in Figure 3 but with the dequeue function from Figure 4 is strongly linearizable and lock-free.*

Proof. Michael and Scott showed that their queue (in particular *enqueue*) is lock-free [11]. The only method that was changed is *dequeue*, and the only change was the removal of a condition which could have caused another iteration of the loop. Thus *dequeue* is still lock-free.

We now show that the queue is strongly linearizable. For an *enqueue* and a successful *dequeue* on O, Lemmas 17 and 20 ensure that both operations modify the interpreted value appropriately at their linearization points. Lemma 18 guarantees that for a failed *dequeue*, the interpreted value is empty at its linearization point. Thus, $f(T)$ is a linearization of the interpreted history $\Gamma(T)$.

Consider a step t of T and an operation $op \in \Gamma(T)$ where $pt(op) = t$. Then

1. operation op is an *enqueue* operation and t is when a successful CAS on line 113 is executed
2. operation op is a *dequeue* operation and t is when a successful CAS on line 156 is executed
3. operation op is a *dequeue* operation and t is when $null$ is read on line 149. Notice that by Lemma 19, reading $null$ guarantees that op will fail.

At step t it is determined what operation op satisfies $pt(op) = t$. Therefore, if S is a prefix of T, then $f(S)$ is a prefix of $f(T)$. ◀

7 Discussion

We proved that Harris's linked list and Michael and Scott's queue, two well-known lock-free data structures, are not strongly linearizable. We have carefully analyzed where the strong linearizability breaks, and gave modifications to derive strongly linearizable variants.

An observation we made on the original data structures is that an operation exists such that the response of the operation was not determined by the time of its linearization point. Using this observation, we constructed transcripts where events after an operation's linearization point changed the linearization order of the past. It is currently unknown whether such observations directly imply that a data structure is not strongly linearizable.

Simple modifications addressing these operations were given but the proofs of strong linearizability were non-trivial. The minor changes required gives hope for future work on deriving strongly linearizable data structures.

We hope that our insights can be used to develop techniques either for determining whether other linearizable implementations are strongly linearizable, or to derive strongly linearizable implementations from linearizable ones. For example, interpreted values have been used to great effect in this paper and by others [4, 12] in proving whether implementations are strongly linearizable, albeit in an ad-hoc manner. A future direction could be to formalize the concept of interpreted values, then develop techniques around it.

References

1 H. Attiya, A. Castañeda, and D. Hendler. Nontrivial and universal helping for wait-free queues and stacks. *Journal of Parallel and Distributed Computing*, 121:1–14, 2018.
2 O. Denysyuk and P. Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *International Symposium on Distributed Computing*, pages 60–74, 2015.
3 M.J. Fischer, N.A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, 1985. `doi:db/journals/jacm/FischerLP85.html,10.1145/3149.214121`.
4 G. Giakkoupis, M.J. Giv, and P. Woelfel. Efficient randomized DCAS. In *Proceedings of the 53rd Symposium on Theory of Computing*, pages 1221–1234. ACM, 2021.
5 W. Golab, L. Higham, and P. Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd Symposium on Theory of Computing*, pages 373–382, New York, NY, USA, 2011. Association for Computing Machinery.
6 T.L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.
7 M. Helmi, L. Higham, and P. Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *Proceedings of the 2012 ACM symposium on Principles of Distributed Computing*, pages 385–394, 2012.
8 M.P. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.
9 M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
10 M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing research*, 4:163–183, 1987.
11 M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, New York, NY, USA, 1996. Association for Computing Machinery.
12 S. Ovens and P. Woelfel. Strongly linearizable implementations of snapshots and other types. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019*, pages 197–206. ACM, 2019.

A Proofs of claims from Section 3

▶ **Lemma 1.** *The linked list implementation by Harris (Figure 1) is not strongly linearizable.*

Proof. We denote $node_i$ as the node containing key i, $in_p(x)$ as a transcript of an insert of key x by process p and $del_p(x)$ as a transcript of a delete of key x by p.

Consider the following transcripts for processes p and q:

$$S = in_p(3) \circ (del_q(2) \text{ to the first execution of line 15}) \circ in_p(2)$$
$$T_1 = S \circ del_p(3) \circ (del_q(2) \text{ from line 16 to completion})$$
$$T_2 = S \circ (del_q(2) \text{ from line 16 to completion})$$

To show a contradiction, assume that the algorithm is strongly linearizable. Then there exists a strong linearization function f for $\{S, T_1, T_2\}$. In S, the insert of 3 happens before every other operation, thus $in_p(3)$ is the first operation in $f(S)$. Either $del_q(2)$ is ordered before $in_p(2)$ in $f(S)$, or it is not. We consider both cases below. For $del_q(2)$, we can see from tracing the code that during its execution up to line 15, $node_3$ is assigned to $right$.

Suppose $del_q(2)$ linearizes prior to $in_p(2)$ in S. Then we have

$$f(S) = in_p(3) \circ del_q(2) \circ in_p(2).$$

In T_1, p completes $del_p(3)$ before q finishes its *delete*. Note that while p executes $del_p(3)$, q takes no steps. By the post-conditions of *search*, $node_3$ is assigned to $right$ on line 39 of $del_p(3)$. This $right$ will fail the if condition on the next line. During the execution of S, no node is marked, and $right.next$ does not change after its insertion. Thus, the if condition on line 44 is satisfied during the execution of $del_p(3)$. For the same reason, $del_p(3)$ will succeed its CAS call on line 45, and $node_3$ is marked. Therefore, when q resumes its execution of $del_q(2)$, $right$ ($node_3$) will be marked. The condition on line 15 succeeds, and q restarts search.

From the post-conditions of *search*, line 56 of $in_p(2)$ assigns $Tail$ to $right$, which will fail the if condition on line 57. No shared memory operations have completed between p's execution of lines 56 and 61, thus the CAS call on line 61 will succeed, and $in_p(2)$ will succeed (return *true*). Therefore, when q executes another *search*, $right = node_2$, and the CAS will succeed on line 45, thus $del_q(2)$ will succeed.

From our assumption that f is strongly linearizable, if $del_q(2)$ linearizes before $in_p(2)$ in S, then $del_q(2)$ also linearizes before $in_p(2)$ in T_1. We have,

$$f(T_1) = in_p(3) \circ del_q(2) \circ in_p(2) \circ del_p(3).$$

However, this sequential history is not valid since $del_q(2)$ succeeds when no node in the linked list contains 2. This contradicts the assumption that $del_q(2)$ linearizes before $in_p(2)$.

Now suppose that $del_q(2)$ does not linearize before $in_p(2)$ in S; either $del_q(2)$ linearizes after $in_p(2)$ in S, or it does not linearize in S. That is,

$$f(S) = in_p(3) \circ in_p(2) \circ del_q(2) \text{ or } f(S) = in_p(3) \circ in_p(2).$$

No *delete* operation occurs in T_2 other than $del_q(2)$, thus when q continues $del_q(2)$, it fails the if condition on line 15 and returns $node_3$ as $right$. The search key (2) and the key of $right\_node$ (3) are different, and $del_q(2)$ returns $false$.

From our assumption that f is strongly linearizable, if $f(S) = in_p(3) \circ in_p(2) \circ del_q(2)$, then $f(T_2) = in_p(3) \circ in_p(2) \circ del_q(2)$. Otherwise, since $del_q(2)$ is complete in T_2, $del_q(2)$ linearizes in T_2 but not in S. Since $del_q(2)$ still linearizes after $in_p(2)$ in T_2, $f(T_2)$ is the same as above. However, this sequential history is not in the sequential specification; a *delete* method fails when the key being deleted is in the linked list. This contradicts the assumption that $del_q(2)$ does not linearize before $in_p(2)$ in S.

Both cases contradict the assumption that f is a strong linearization function. Therefore, no strong linearization function can be defined over $\{S, T_1, T_2\}$, and Harris's linked list implementation is not strongly linearizable. ◀

B Proofs of claims from Section 4

▶ **Lemma 2.** *A marked node's succ field never changes.*

Proof. A *succ* field is only changed by the three CAS operations and on line 94. It is clear that none of the three CAS operations will succeed if a node is marked. A node when constructed is by default unmarked, and when changed on line 94 it is left unmarked. Thus new_node on line 94 is always unmarked when it changes. ◀

▶ **Lemma 3.** *Keys are strictly sorted; For any two nodes v_1 and v_2, if $v_1.next = v_2$ then $v_1.key < v_2.key$.*

Proof. Initially, there are only $Head$ and $Tail$ where $Head.key < Tail.key$, thus the lemma is true. The $next$ field of a node is only ever altered on lines 94 and 95 in $SLinsert$, and on line 86 in $search$. We show that if the lemma is true before each of the listed operations, then it is true after the operation. For the lines corresponding to CAS operations, we assume that the call succeeds since otherwise no change takes place.

Consider an $SLinsert$ operation. The $search$ call return only if $left.key \leq search\_key < right.key$. Line 94 is executed if $left.key \neq search\_key$, thus $left.key < search\_key < right.key$. Then we have that after line 94, $new\_node.next = right$ and $new\_node.key = search\_key < right.key$. Furthermore, if the CAS on line 95 succeeds, $left.next = new\_node$ and we have already established that $left.key < search\_key = new\_node.key$.

For the CAS in $search$, consider the sequence of nodes

$$v_1, v_2, \ldots, v_k$$

where $v_1 = start$, $v_k = curr$ at the execution of the CAS, and v_{i+1} is $v_i.next$ when it was read on line 67, 80 or 84. The sequence is well defined since $curr$ is set to $curr\_next$ after $curr.succ$ is read. After the CAS succeeds, $start.next = curr$. Since we assume that the lemma is true before the CAS succeeds, $v_1.key < v_k.key$, thus it is still maintained. ◀

▶ **Corollary 4.** *The linked list never contains duplicate keys.*

Proof. If it contained duplicate keys Lemma 3 is violated. ◀

▶ **Lemma 5.** *All unmarked, not pre-inserted nodes are reachable.*

Proof. Reachability is only affected by the CAS on line 95, when a node is inserted, and on line 86, when $start.next$ is changed to $curr$.

At the CAS success on line 95, $left$ is unmarked, and is thus reachable. The $next$ field of $left$ is new_node, hence new_node is reachable. The node $right$ is still reachable since $new\_node.next = right$. For the CAS in $search$, we want to show that no unmarked node exists "between" $start$ and $curr$ on line 86. Again consider the sequence of nodes

$$v_1, v_2, \ldots, v_k$$

where $start = v_1$, $v_k = curr$ and v_{i+1} is the node seen when $v_i.next$ is read. Note that for $1 < i < k$, v_i was seen to be marked when $v_{i-1}.next$ was read. Thus by Lemma 2, such v_i are marked at the execution of the CAS. We show that at the CAS execution, $v_i.next = v_{i+1}$ for all $1 \leq i < k$, proving that all nodes "between" $start$ and $curr$ are all marked.

Note that $start\_next = v_2$; when v_1 was assigned to $start$, $curr\_next = v_2$ was assigned to $start\_next$ (line 71). At CAS execution, $start\_next = start.next$ since it succeeds, so $v_1.next = v_2$ at this time. For all other v_i, $v_i.next = v_{i+1}$ at the CAS success since after $v_i.succ$ has been read (and was seen to be marked), it cannot change by Lemma 2. ◀

▶ **Lemma 12.** *The search method is lock-free.*

Proof. Consider the first inner loop in $search$. After every iteration of the loop, $curr$ advances down the linked list by one node. For every $search\_key$, $Head.key < search\_key < Tail.key$. The linked list is strictly sorted by their keys, and $Tail$ is always reachable. Thus the exit condition on line 72 is always met before $Tail$ is assigned to $curr$. Consider an execution of the loop that lasts more than k iterations. At iteration k, the variable $curr$ is not $Tail$, and $curr$ has advanced down the linked list k times. Since initially the linked list consists only of $Head$ and $Tail$, at least k $SLinsert$ operations have linearized.

Now consider the second inner loop in $search$. After every iteration of the loop, $curr$ advances down the linked list by one node. The $Tail$ node is unmarked, thus by the time $curr = Tail$, the exit condition of the loop is met. By similar reasoning as above, if the loop execution lasts more than k iterations, then at least k successful $SLinsert$ have linearized.

Finally, consider the outer loop execution that lasts at least $k > 2$ iterations. For clarity, we denote the node assigned to variable x on iteration j as x_j. For an iteration $i < k$, suppose that the CAS on line 81 86 fails; $start.succ \neq (start\_next, false)$. Then $start.succ$ was modified by an update between the CAS execution and when $start.succ$ was read on line 67 or 77. Now suppose that the CAS succeeds. Observe that on line 81, $start.key \leq search\_key < curr_i.key$, and at this point $start$ is unmarked and therefore reachable; no reachable node with key between $start.key$ and $curr_i.key$ exists. However on iteration $i+1$, when line 73 is executed to exit from the first inner loop, $curr_{i+1}$ is marked. One of the following updates must have occurred between the CAS on iteration i and when $curr_{i+1}.marked$ was read:

1. $start_i$ was marked, or
2. A node v with $start.key \geq v.key < search\_key$ was inserted.

Otherwise, $curr_{i+j} = start_i$, $start_i$ is unmarked and $search\_key < curr\_next_{i+1}.key$, meaning that $search$ should return on iteration i. An update occurred for all cases, thus for k iterations of the outer loop, k updates occurred. For k updates, at least $k/2$ successful $SLinsert$ or $SLdelete$ operations have linearized. Then we have that if a process takes infinitely many steps (infinitely many iterations of any loop) while executing the $search$ function, then infinitely many successful $SLinsert$ or $SLdelete$ operations have linearized. ◀

C Proofs of claims from Section 5

▶ **Lemma 14.** *Michael and Scott's queue (Figure 3) is not strongly linearizable.*

Proof. Again, we denote $node_i$ as a node containing value i. Similarly, $deq_p()$ as a transcript of a $dequeue$ operation by process p, and $enq_p(x)$ as a transcript of a $enqueue$ operation of value x by process p. For clarity, when tracing the execution of different processes, we denote variable var from p's execution var_p.

Consider the following transcripts for processes p and q:

$$S = (deq_p() \text{ to the first execution of line 121}) \circ enq_q(1) \circ enq_q(2)$$
$$T_1 = S \circ deq_q() \circ (deq_p() \text{ from line 122 to completion})$$
$$T_2 = S \circ (deq_p() \text{ from line 122 to completion })$$

To show a contradiction, suppose that the algorithm is strongly linearizable with a strong linearization function f over $\{S, T_1, T_2\}$. When we trace the execution outlined by S, v_{dummy} is assigned to $start_p$ and end_p, and $null$ is assigned to $next$ (lines 119- 121). In addition, $enq_q(1)$ and $enq_q(2)$ append their respective nodes to the linked list.

Now consider the rest of the execution in T_1. The $deq_q()$ operation terminates successfully. $Head$ is assigned to $start_q$ and $node_2$ is assigned to end_q (thus $start_q \neq end_q$). $Head$ never changed ($Head = v_{dummy}$) thus the CAS on line 129 succeeds. When $deq_p()$ resumes its execution, it fails the condition on line 122 (since $Head$ was changed by $deq_q()$) and restarts.

During the next iteration of the loop, *Head* does not change, as q does not execute any operations. In addition, $start_q = node_1$ and $end_q = node_2$ are read on lines 119-120. The *CAS* on line 129 is therefore reached, and succeeds to change *Head* to $node_2$.

For $f(T_1)$ to be a linearization of T_1, $deq_p()$ cannot be ordered first. Otherwise, a *dequeue* operation succeeded (as we saw when tracing the execution) when no *enqueue* operation preceded before it. As S is a prefix of T_1, for $f(S)$ to be prefix-preserving, $f(S)$ also cannot start with $deq_p()$. Now, we consider the transcript T_2. Continuing from our tracing of S, $start_p = v_{dummy}$ and $next_p = null$. *Head* has yet to change (is still v_{dummy}), thus the condition on line 122 passes. The next two if statements (line 123-124) is also satisfied, and the $deq_p()$ fails.

In order to preserve validity,

$$f(T_2) = deq_p() \circ enq_q(1) \circ enq_q(2).$$

Since $f(S)$ is a prefix of $f(T_2)$ (S is a prefix of T_2, and f is prefix-preserving, and S contains complete operations $enq_q(1)$ and $enq_q(2)$,

$$f(S) = deq_p() \circ enq_q(1) \circ enq_q(2).$$

However $f(S)$ cannot start with $deq_p()$, yielding a contradiction. ◀

D Proofs of claims from Section 6

▶ **Observation 22.** *For a node v, if $v.next \neq null$, then $v.next$ does not change.*

Proof. Initially, $v.next = null$. The *next* field of a node is only ever altered in *enqueue* by a *CAS* in line 113. Such a *CAS* only succeeds if $v.next = null$, and after the *CAS*, $v.next \neq null$. ◀

▶ **Corollary 23.** *For a node v, if $v.next = null$, then $v.next$ never changed since v was constructed.*

Proof. Otherwise $v.next$ was changed to a node u between v's initialization and when $v.next = null$. However by Lemma 22 $v.next$ can never change back to *null*. ◀

▶ **Observation 24.** *If node v is reachable and $v.next = null$, then v is the last reachable node.*

▶ **Lemma 25.** *$Tail$ is only ever incremented; if $Tail = node$, then $Tail$ only ever changes to $node.next$ where $node.next \neq null$.*

Proof. *Tail* is only ever changed through *CAS* operations on lines 153 and 116. We show that if such a *CAS* succeeds on either line, *Tail* is incremented.

For line 116, a successful *CAS* changes *Tail* from $(end, endc)$ to $(next, endc+1)$, where $next \neq null$ by the prior if condition. By Observation 22, at the time of *CAS* success, $end.next = (next, nextc)$. Next, we show that $end.next = (next, nextc)$ on line 153. We know that $next \neq null$ since the if condition on line 151 was not satisfied (otherwise the method call would not reach line 153). By the if condition on line 150, $start = end$, meaning $start.next = end.next = next$ at CAS success (line 153) by Observation 22. ◀

▶ **Observation 26.** *If $Head$ changes from node v to node u, then $v.next = u$ when $Head$ was changed.*

Proof. $Head$ is only altered by a successful CAS on line 156, and it is changed to $next$. Suppose the CAS operation succeeds and changes $Head$ from v to u. Then, v was assigned to $start$ on line 147 and $u \neq null$ was assigned to $next$ on line 149. By Observation 22, $v.next = u$ at the time of CAS success. ◀

▶ **Lemma 15.** *$Tail$ is always reachable.*

Proof. Initially, $Tail$ and $Head$ refer to the same node. By induction, we show that the lemma continues to hold even after $Tail$ or $Head$ is change by a successful CAS operation. For every CAS operation that changes $Tail$ or $Head$, suppose that $Tail$ is reachable up until the CAS operation. By Lemma 25, any time $Tail$ changes it is changed to $Tail.next$. Since $Tail$ is reachable, $Tail.next$ is also reachable. Thus, successful CAS operations on line 153 and 116 maintain the lemma.

We first prove that if the CAS on line 156 is reached, then $next \neq null$. To show a contradiction, suppose otherwise. By the induction hypothesis, $Tail$ is reachable during the execution of lines 147-148. By the assumption that $next = null$ and Corollary 23, $start.next = null$ on lines 147-148 and $start$ is the last node in the list in this duration. If $Head = start$ on line 148, then $start$ is the last reachable node and $start$ is assigned to end on line 148 (since $Tail$ is reachable at this line). Then $dequeue$ exits on line 152 and line 156 is never reached, yielding a contradiction. Otherwise, $Head \neq start$ on line 148. $Head$ must have changed since its assigned to $start$, but $Head \neq null$ for $Tail$ to be reachable. Then we have that $Head$ changed from $start$ to a node u. However, this contradicts Lemma 26; $Head$ changed from $start \neq null$ to $u \neq null$, but $start.next = null \neq u$.

We now have that $next \neq null$ at the CAS operation on line 156. By Lemma 26, if the CAS operation succeeds, then $Head$ changes to $Head.next$. The only way such a change can make $Tail$ unreachable from $Head$ is if $Tail = Head$ at CAS success. To show a contradiction, suppose that this is the case. For the CAS on line 156 to succeed, $Head$ does not change after it was assigned to $start$ on line 147. $Tail$ was assigned to end on line 148, and $start$ was evaluated to not equal end on line 150. The node pointed to by $Tail$ must have changed for $Tail$ and $Head$ to reference the same node at the CAS, but such a change can only be an increment by Lemma 25. Thus $Tail$ was unreachable from $Head$ prior to the CAS execution, which contradicts the inductive hypothesis. ◀

▶ **Lemma 16.** *$Head$ is never null, and is only ever incremented.*

Proof. If $Head$ was $null$, then $Tail$ would be unreachable. Thus if $Head$ changes, then it changes from a node v to a node u. By Lemma 26, $Head$ is then only ever incremented. ◀