Strong Linearizability Without Compare&Swap: The Case of Bags

Faith Ellen
University of Toronto, Canada

Gal Sela
EPFL, Lausanne, Switzerland

Abstract

Because strongly-linearizable objects provide stronger guarantees than linearizability, they serve as valuable building blocks for the design of concurrent data structures. Yet, many objects that have linearizable implementations from base objects weaker than compare&swap objects do not have strongly-linearizable implementations from the same base objects. We focus on one such object: the *bag*, a multiset from which processes can take unspecified elements.

We present the first lock-free, strongly-linearizable implementation of a bag from interfering objects (specifically, registers and test&set objects). This may be surprising, since there are provably no such implementations of stacks or queues.

Since a bag can contain arbitrarily many elements, an unbounded amount of space must be used to implement it. Hence, it makes sense to also consider a bag with a bound on its capacity. However, like stacks and queues, a bag with capacity b shared by more than $2b$ processes has no lock-free, strongly-linearizable implementation from interfering objects. If we further restrict a bounded bag so that only one process can insert into it, we are able to obtain a lock-free, strongly-linearizable implementation from $O(b+n)$ interfering objects, where n is the number of processes.

Our goal is to understand the circumstances under which strongly-linearizable implementations of bags exist and, more generally, to understand the power of interfering objects.

2012 ACM Subject Classification Computing methodologies → Shared memory algorithms; Theory of computation → Data structures design and analysis; Theory of computation → Concurrent algorithms

Keywords and phrases Strong-Linearizability, Bag, Concurrent Data Structures, Wait-Freedom, Lock-Freedom

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.29

Related Version *Full Version*: `https://arxiv.org/abs/2411.19365` [10]

Funding This work was supported in part by the Natural Science and Engineering Research Council of Canada (NSERC) grant RGPIN-2020-04178.

1 Introduction

Concurrent data structures are often built using linearizable implementations of objects as if they were atomic objects. Although linearizability is the usual correctness condition for concurrent algorithms, there are scenarios where composing linearizable implementations does not preserve desired properties. Namely, an algorithm that uses atomic objects may lose some of its properties when these atomic objects are replaced by linearizable implementations. Specifically, linearizability does not preserve hyperproperties (properties of sets of executions), for example, probability distributions of events for randomized programs and security properties such as noninterference [11, 6]. This can be rectified by using strongly-linearizable implementations [11]. They guarantee that the linearization of a prefix of a concurrent execution is a prefix of the linearization of the whole execution. This means that the linearization of operations in an execution prefix cannot depend on later events in the execution.

Attiya, Castaneda, and Enea [3] observed that strongly-linearizable implementations of objects typically use compare&swap objects, which can be used to solve consensus among any number of processes. They provided strongly-linearizable implementations of some objects, such as a wait-free single-writer snapshot object and a wait-free max register from a fetch&add object, and a lock-free, readable, resettable test&set object and a lock-free, readable, fetch&increment object from an infinite array of test&set objects and registers. Both fetch&add and test&set objects are less powerful than a compare&swap object.

In this work, we aim to further explore the power of these well-studied building blocks in the context of strong linearizability. Specifically, we investigate strongly-linearizable implementations of a bag from these primitives. Establishing the existence of such implementations deepens our understanding of the circumstances under which these primitives can be used to achieve strong linearizability.

A *bag* is a multiset to which processes can insert elements and from which they can take unspecified elements. A concurrent bag is an abstraction of the interaction between producers and consumers and is commonly used to distribute tasks among processes for load balancing and improved scalability.

Queues and stacks have strongly-linearizable implementations from compare&swap objects [15, 21, 19]. A bag has a straightforward implementation from a stack or a queue. Therefore, a bag has a strongly-linearizable implementation from compare&swap objects. Although there is a wait-free, linearizable implementation of a stack from registers, test&set objects, and a readable fetch&increment object [1] and there is a lock-free, linearizable implementation of a queue from the same set of objects [16], neither of these implementations is strongly-linearizable. In fact, Attiya, Castañeda, and Hendler [5] proved that no lock-free, strongly-linearizable implementation of a stack or queue from such objects is possible. Attiya, Castañeda, and Enea [3] claimed that the lock-free linearizable implementation of a queue which appears in Figure 5, is a strongly-linearizable implementation of a bag. We give a counterexample to this claim in Appendix A. This was the starting point for our research.

The challenge when implementing a strongly-linearizable bag is to enable an operation that is trying to take an element from the bag to detect when the bag is empty. If elements can reside in multiple locations, it does not suffice for the operation to simply examine these locations one by one, concluding that the bag is empty if no elements are found. The problem is that new elements may have been added to locations after they were examined. Even if there was a configuration during the operation in which the bag was empty, it might not be possible to linearize the operation at this point while maintaining strong linearizability. For example, after this point, in an alternative execution, a value could be added to a location that the operation had not yet examined and the operation could return that value, instead of EMPTY.

We address this challenge in Section 4, modifying the lock-free, linearizable implementation of a queue by adding an additional readable fetch&increment object. This object is used by operations which are inserting elements into the bag to inform operations which are trying to take elements from the bag that the bag is no longer empty. We prove that the resulting algorithm is a strongly-linearizable implementation of a bag.Interestingly, although our algorithm is still a linearizable implementation of a queue, it is not a strongly-linearizable implementation of a queue.

A bag can grow arbitrarily large, so either the number of objects used to implement it or the size of the objects must be unbounded. An alternative is a *b-bounded bag*, which can simultaneously contain up to b elements. Unfortunately, Attiya, Castaneda, and Enea [3] proved that a b-bounded bag shared by more than $2b$ processes has no lock-free, strongly-

linearizable implementation from interfering objects [3]. However, we do provide a lock-free, strongly-linearizable implementation of a restricted version of a b-bounded bag, into which only one process, the *producer*, can insert elements. It uses a bounded number of registers, each of bounded size, and readable, resettable test&set objects. Before presenting this implementation, we give two different implementations of a 1-bounded bag with a single producer, which introduce some of the ideas that appear in our b-bounded bag implementation.

In Section 5, we present a wait-free, linearizable implementation of a 1-bounded bag. Note that it provides a stronger progress guarantee. To keep the space bounded, when the producer inserts an element into the bag, it may reuse objects previously used for other elements. It has to do this carefully, to avoid reusing objects that other processes may be poised to access. This requires delicate coordination between the producer and the consumers.

In Section 6, we present a lock-free, strongly-linearizable implementation of a 1-bounded bag. It combines the mechanism for detecting an empty bag from our lock-free, strongly-linearizable implementation of a bag and the mechanism for reusing objects from our wait-free, linearizable implementation of a 1-bounded bag.

Our lock-free, strongly-linearizable implementation of a b-bounded bag, for any value of b, is presented in Section 7. To enable the producer to detect when the bag is full, we use an approach that is similar to detecting when the bag is empty, but we have to ensure that these two mechanisms do not interfere with one another. In our implementation of a 1-bounded bag in Section 6, the producer determines whether the bag is empty or full by inspecting a single object. This makes this implementation both simpler and more efficient than the special case of our implementation of a b-bounded bag when $b = 1$.

2 Preliminaries

We consider an asynchronous system where processes communicate through operations applied to shared objects. Each type of object supports a fixed set of operations that can be applied to it. For example, a *register* supports `read`(), which returns the value of the register, and `write`(x), which changes the value of the register to x.

Each process can be modelled as a deterministic state machine. A *step* by a process specifies an operation that the process applies to a shared object and the response, if any, that the operation returns. The process can then perform local computation, perhaps based on the response of the operation, to update its state. A process can crash, after which it takes no more steps.

A *configuration* consists of the state of every process and the value of every shared object. In an *initial configuration*, each process is in its initial state and each object has its initial value. An *execution* is an alternating sequence of configurations and steps, starting with an initial configuration. If an execution is finite, it ends with a configuration. The order in which processes take steps is determined by an adversarial scheduler.

The *consensus number* of an object is the largest positive integer n such that there is an algorithm solving consensus among n processes using only instances of this object and registers. A register has consensus number 1. Another example of an object with consensus number 1 is an *ABA-detecting register* [2]. It supports `dWrite`(x) and `dRead`(). When `dWrite`(x) is performed, x is stored in the object. When a process p performs `dRead`(), the value stored in the object is returned, together with an output bit, which is true if and only if process p has previously performed `dRead`() and some `dWrite`() has been performed since its last `dRead`(). We use a restricted version of an ABA-detecting register, where `dWrite`() does not have an input parameter and `dRead`() simply returns the output bit.

A *test&set* object has initial value 0 and supports only one operation, `t&s`(). If the value of the object was 0, this operation changes its value to 1 and returns 0. If the value of the object was 1, it just returns 1. In the first case, we say that the `t&s` was *successful* and, in the second case, we say that it was *unsuccessful*. A *resettable test&set* object is like a test&set object, except that it also supports `reset`(), which changes the value of the object to 0. A *fetch&increment* object supports `f&i`(), which increases the value of the object by 1 and returns the previous value of the object. A *compare&swap* object supports the operation `c&s`(u, v), which checks whether the object has value u and, if so, changes its value to v. It always returns the value of the object immediately before the operation was performed. For any type of object that does not support `read`, its readable version supports `read` in addition to its other operations.

Two operations, op and op', *commute* if, whenever op is applied to the object followed by op', the resulting value of the object is the same as when op' is applied followed by op. Note that the responses to op and op' may be different when these operations are applied in the opposite order. The operation op' *overwrites* the operation op if, whenever op is applied to the object followed by op', the resulting value of the object is the same as when only op' is applied. Note that the response to op' may be different depending on whether op was applied. An object is *interfering* if every two of its operations either commute or one of them overwrites the other. All interfering objects have consensus number at most 2 [13]. Registers are interfering. The test&set, resettable test&set, and fetch&increment objects and their readable versions are all interfering and have consensus number 2.

Both a *stack* and a *queue* are examples of non-interfering objects with consensus number 2 [13]. The value of these objects is an unbounded sequence of elements, which is initially empty. A stack supports `push`(x) and `pop`(), whereas a queue supports `enqueue`(x) and `dequeue`().

A *bag* is a non-interfering object whose value is an initially empty multiset of elements. The number of elements that can be in the multiset is unbounded. The elements are taken from a set of values V that does not include $\perp$. A bag supports two operations, `Insert`(x) where $x \in V$ and `Take`. When `Insert`(x) is performed, the element x is added to the multiset. It does not return anything. If the multiset is not empty, a `Take` operation removes an arbitrary element from the multiset and returns it. In this case, we say that the operation is *successful*. If the multiset is empty, then `Take` returns EMPTY and we say that it is *unsuccessful*. Note that this specification is nondeterministic.

A *b-bounded bag* object is an object whose value is a multiset of at most b elements. If the multiset contains fewer than b elements when `Insert`(x) is performed, the element x is added to the multiset and OK is returned. If the multiset already contains b elements, the value of the bag does not change and FULL is returned. For simplicity, we assume no value repetitions in the bounded bags in our proofs.

An *implementation* of an object O shared by a set of processes P from a collection of objects C provides a representation of O using objects in C and algorithms for each process in P to perform each operation supported by O. In an execution, an operation on an implemented object *begins* when a process performs the first step of its algorithm for performing this operation and is *completed* at the step in which the process returns from the algorithm, with the response of the operation, if any.

A bag has a straightforward implementation from a stack or a queue. The multiset is represented by the sequence, `Insert`(x) is performed by applying `push`(x) or `enqueue`(x) and `Take`() is performed by applying `pop`() or `dequeue`(). Thus a bag has consensus number at most 2.

There is a simple algorithm for solving consensus among 2 processes using two single-writer registers and a bag initialized with one element: To propose the value x, a process writes x to its single-writer register and then performs a `Take`() operation on the bag. If the `Take`() returns an element, the process returns x. Otherwise, it reads and returns the value that the other process wrote to its single-writer register. It follows from Borowsky, Gafni, and Afek's *Initial State Lemma* [7, Lemma 3.2], that consensus among 2 processes can be solved from registers and two initially empty bags. Thus, a bag has consensus number exactly 2.

For any execution, α, consider an ordering of every completed operation and a (possibly empty) subset of the operations that have begun, but have not completed, such that, for every completed operation, *op*, each of these operations that begins after *op* has completed is ordered after *op*. A way to obtain such an ordering is to first assign a linearization point in α to each of these operations within its *operation interval*, i.e., no operation is linearized before it begins or after it has completed. Then specify an ordering of the operations that are linearized at the same point. Suppose there is a sequential execution in which the operations in the ordering are performed in order such that every completed operation in α has the same response as it does in this sequential execution. Then we say that the ordering is a *linearization* of α.

An implementation of an object is *linearizable* [14] if each of its executions has a linearization. An implementation is *strongly-linearizable* [11] if there is a function f that maps each execution α to a linearization of α such that, for every prefix α' of α, $f(\alpha')$ is a prefix of $f(\alpha)$.

An implementation is *wait-free* if, in every execution, every operation by a process that does not crash completes within a finite number of steps by that process. An implementation is *lock-free* if every infinite execution contains an infinite number of completed operations.

3 Related Work

Even though there are wait-free, linearizable implementations of registers, max-registers, and single-writer snapshot objects from single-writer registers, Helmi, Higham, and Woelfel [12] proved that there are no lock-free, strongly-linearizable implementations. They also gave a wait-free, strongly-linearizable implementation of a bounded max-register from registers. Denysyuk and Woelfel [9] proved there are no wait-free, strongly-linearizable implementations of max-registers and single-writer snapshot objects from registers, although they gave lock-free strongly-linearizable implementations. Later, Ovens and Woelfel [20] gave lock-free, strongly-linearizable implementations of an ABA-detecting register and a single-writer snapshot object from a bounded number of bounded size registers.

Li [16] gave a lock-free, linearizable implementation of a queue from a fetch&increment object, an infinite array of test&set objects, and an infinite array of registers. He also gave a wait-free, linearizable implementation of a queue in which at most 2 processes may dequeue. It uses a fetch&increment object and infinite arrays of registers and test&set objects. David [8] gave a wait-free, linearizable implementation of a queue in which at most 1 process may enqueue. It uses registers, an infinite array of fetch&increment objects, and a two-dimensional infinite array of swap objects. It is unknown whether there is a wait-free, linearizable implementation of a queue from interfering objects in which any number of processes can enqueue and dequeue.

Afek, Gafni, and Morrison [1] gave a wait-free, linearizable implementation of a stack from interfering objects (specifically, a fetch&add object, registers, and test&set objects). Attiya and Enea [6] showed that this implementation is not strongly-linearizable. Attiya, Castañeda,

and Enea [3] later proved that there is no lock-free, strongly-linearizable implementation of a stack or a queue shared by more than 2 processes from interfering objects.

Many universal constructions (for example, from compare&swap objects and registers or from consensus objects and registers) provide strongly-linearizable, wait-free implementations of any shared object [11]. Treiber [21, 19] gave a simple lock-free, strongly-linearizable implementation of a stack from compare&swap objects and registers. Michael and Scott [18] gave a lock-free, linearizable implementation of a queue from compare&swap objects and registers. Assuming nodes are never reused, it is possible to make a small modification to their implementation to make it strongly-linearizable [15].

4 A Lock-Free, Strongly-Linearizable Implementation of a Bag

The first algorithm we present is a strongly-linearizable implementation of an unbounded bag. The challenge in designing such an algorithm from interfering objects lies in identifying when the bag is empty. To address this, we use a readable fetch&increment object, `Done`, which an `Insert` operation increments as its last step, to inform `Take` operations that the bag is not empty. A detailed description of the algorithm follows.

The implementation uses an infinite array of registers, $Items$, in which elements that have been inserted into the bag are stored, together with an infinite array of test&set objects, TS. The process that performs a successful `t&s`() on $TS[i]$ returns the element stored in $Items[i]$ and removes it from the bag.

Each `Insert`(x) operation begins by performing `f&i`() on a readable fetch&increment object, `Allocated`. This allocates the operation a new location within `Items` in which it writes x. Thus, the value of `Allocated` is the index of the last allocated location within `Items`. Finally, the operation performs `f&i`() on a second readable fetch&increment object, `Done`, to inform `Take`() operations about the insertion.

A `Take`() operation begins by reading `Done` and `Allocated`. Then it reads the allocated locations in `Items`. For each location, `i`, if `Items[i]` contains $x \neq \perp$, the operation performs `t&s`() on the corresponding test&set object, `TS[i]`. If this is successful, x is returned. After reading all the allocated locations in `Items` without performing a successful `t&s`(), the operation rereads `Done`. If its value has not changed since the operation last read `Done`, EMPTY is returned. Otherwise, the operation repeats the entire sequence of steps. Note that it begins again starting from the first location, in case an `Insert` operation that was allocated an early location has recently written to that location. Pseudocode for our implementation of a bag appears in Figure 1.

Strong-linearizability. Consider any execution consisting of operations on this data structure. We linearize the operations as follows:

- A `Take`() operation that performs a successful `t&s`() on Line 17 is linearized at this step. It returns the element it last read on Line 15.
- A `Take`() operation that reads `Done` on Line 18 and obtains the same value it obtained in its previous read of `Done` (on Line 12) is linearized at this last read. It returns EMPTY.
- Consider an `Insert`(x) operation, *ins*, that was allocated location `m` and wrote x to `Items[m]` on Line 8. If some `Take`() operation performs a successful `t&s`() on `TS[m]` after *ins* performed Line 8, but before *ins* performs `Done.f&i`() on Line 9, then *ins* is linearized immediately before this `Take`(). In this case, we say that these two operations are *coupled*. Otherwise, *ins* is linearized when it performs `Done.f&i`().

```
1  Shared variables:
2     Items[1..]: an infinite array of registers, each initialized to ⊥
3     TS[1..]: an infinite array of test&set objects, each initialized to 0
4     Allocated: a readable fetch&increment object, initialized to 0
5     Done: a readable fetch&increment object, initialized to 0
6  Insert(x):
7     m ← Allocated.f&i() + 1
8     Items[m].write(x)
9     Done.f&i()
10 Take():
11    repeat
12       d ← Done.read()
13       m ← Allocated.read()
14       for i ← 1 to m do
15          x ← Items[i].read()
16          if x ≠ ⊥ then
17             if TS[i].t&s() = 0 then return x
18       if d = Done.read() then return EMPTY
```

Figure 1 A lock-free, strongly-linearizable bag.

The linearization point of each operation occurs within its operation interval. At any configuration, C, the bag contains the difference between the multiset of elements inserted by `Insert` operations linearized before C and the multiset of elements returned by successful `Take` operations linearized before C.

Each `Take`() operation is linearized immediately before it returns. Thus, if it is linearized at some point in an execution, it is linearized at the same point in every extension of that execution. Each `Insert`(x) operation that is coupled with a `Take`() operation is linearized immediately before that `Take`() operation. This means that the inserted element is taken from the bag immediately after it is inserted. Thus, if a coupled `Insert`(x) operation is linearized at some point in an execution, it is linearized at the same point in every extension of that execution. Each uncoupled `Insert`(x) operation is linearized when it increments `Done`. This ensures that, if a `Take`() operation sees that the value of `Done` has not changed between Line 12 and Line 18, then no uncoupled `Insert` has been linearized during this part of the execution. An uncoupled `Insert`(x) operation is linearized immediately before it returns, so it is linearized at the same point in every extension of the execution. Hence, if the implementation is linearizable, it is strongly-linearizable.

To show that the ordering is a linearization, it remains to prove that the values returned by the `Take`() operations are consistent with the sequential specifications of a bag. Let tk be a `Take`() operation that performs a successful `t&s`() of `TS`[i] on Line 17 and let $x \neq \perp$ be the element it read from `Items`[i] when it last performed Line 15. Since tk read x from `Items`[i], there was an `Insert`(x) operation that wrote x into `Items`[i] on Line 8. Note that there was exactly one `Insert` operation that was allocated i on Line 7, by the semantics of `f&i`(). This `Insert`(x) operation was linearized before tk, either because it executed Line 9 before tk performed its successful `t&s`() on `TS`[i] or it was coupled with tk and, hence, linearized immediately before tk. The element x remains in the bag until the linearization point of tk because the element in `Items`[i] is only returned by the `Take`() operation that performs the successful `t&s`() on `TS`[i].

Now let tk be a `Take` operation that reads `Done` on Line 18 and obtains the same value it obtained in its previous read of `Done` on Line 12. Let C be the configuration immediately after tk reads `Done` on Line 12 in its last iteration of the repeat loop and let C' be the

configuration immediately before tk reads `Done` for the last time on Line 18. The value of `Done` does not change between these two configurations. Let m be the value tk read from `Allocated` on Line 13, which is the largest location in `Items` that had been previously allocated to an `Insert` operation.

Any element, x, that was in the bag in configuration C was inserted into the bag by an uncoupled `Insert`(x) operation. This operation was linearized when it performed Line 9. Suppose that it wrote x into `Items`[i]. This happened prior to C. Note that $i \leq m$, since m was obtained when tk read `Allocated` on Line 13 after C and the value of `Allocated` only increases. When tk performed Line 15 between configurations C and C', it read x from `Items`[i]. Since tk later reaches Line 18, its `t&s`() on `TS`[i] was unsuccessful. Some other `Take` operation previously performed a successful `t&s`() on `TS`[i] and, when it did, element x was taken from the bag. Therefore, all elements that were in the bag in configuration C were taken by `Take` operations other than tk before C'.

Now consider any element that was inserted into the bag between configurations C and C' by some `Insert` operation. Since the value of `Done` did not change between C and C', this `Insert` operation did not perform `Done.f&i()` on Line 9. Hence, it was coupled with a `Take` operation and it was immediately taken from the bag after it was inserted into the bag. Therefore, all elements that were inserted into the bag between configurations C and C' were taken from the bag prior to configuration C'. Hence, the bag is empty in configuration C', immediately before the step at which tk is linearized.

Lock-freedom. The `Insert` operation performs 3 steps, so it is wait-free. When a `Take` operation finishes an iteration of the loop and is about to begin another iteration, the value of `Done` it last read on Line 18 was different than what it last read on Line 12. This means that some `Insert` operation performed `Done.f&i()` on Line 9 between these two points of the execution. This step completes the `Insert` operation.

It is worth noting that the algorithm in Figure 1 also implements a linearizable queue. The proof of this fact is very similar to the proof used by Li [16]. However, it is not a strongly-linearizable queue, as we prove in Appendix B.

5 A Wait-Free, Linearizable Implementation of a 1-Bounded Bag with a Single Producer

Our first bounded bag algorithm from interfering objects is a wait-free, linearizable implementation of a bag that can contain at most one element. It is shared by n processes, $P_1, \ldots, P_n$, called *consumers*, that can perform `Take`() and a single process, called the *producer*, that can perform `Insert`(x). To keep the space bounded, the producer reuses locations in `Items` to store new elements. The producer announces the most recent location it has allocated by writing it to the shared register `Allocated`. There is a resettable test&set object, instead of a test&set object, associated with each location in `Items`. An array, `Hazards`, of hazard pointers [17] is used to prevent multiple consumers from returning the same element, and to ensure that each element is consumed before being overwritten by a new element. Each consumer announces a location it is about to access and the producer avoids reusing the announced locations. Specifically, the producer avoids resetting the corresponding test&set objects and writing new elements to the corresponding locations in `Items`.

The producer maintains two persistent local variables, `m`, which is the last location in `Items` it allocated, and `used`, which is a set of locations it has used and will need to reset before they are reallocated. To eliminate the need for a special case to handle the first

```
19 Shared variables:
20    Items[1..n+1]: an array of single-writer registers, all initialized to ⊥,
          which can only be written to by the producer
21    TS[1..n+1]: an array of readable, resettable test&set objects, all
          initialized to 0, except for TS[1], which is initialized to 1
22    Allocated: a single-writer register, initialized to 1, which can only be
          written to by the producer
23    Hazards[1..n]: an array of single-writer registers, all initialized to ⊥,
          where Hazards[i] can only be written to by P_i
24 Persistent local variables of the producer:
25    used: a register, initialized to ∅, which contains a subset of
          {1,...,n+1}
26    m: a register, initialized to 1, which is a local copy of Allocated
27 Insert(x) by the producer:
28    if TS[m].read() = 0 then return FULL
29    Items[m].write(⊥)
30    used ← used ∪ {m}
31    hazardous ← COLLECT(Hazards)
32    m ← some index in {1,...,n+1} - hazardous
33    Allocated.write(m)
34    for all i ∈ used - hazardous do
35       TS[i].reset()
36    used ← used ∩ hazardous
37    Items[m].write(x)
38    return OK
39 Take() by consumer P_i for i ∈ {1,...,n}:
40    a ← Allocated.read()
41    Hazards[i].write(a)
42    x ← Items[a].read()
43    if x ≠ ⊥ then
44       if TS[a].t&s() = 0 then
45          Hazards[i].write(⊥)
46          return x
47    Hazards[i].write(⊥)
48    return EMPTY
```

Figure 2 A wait-free, linearizable 1-bounded bag with one producer and n consumers.

`Insert` call (which is the only one not preceded by a previous insertion), the initial state of the data structure is as if location 1 had been allocated to the producer, it had inserted an element into the bag in this location, and then this element had been taken by some consumer. This is simulated by setting the initial values of `m`, `Allocated`, and `TS[1]` to 1.

The producer begins an `Insert`(x) operation by checking whether the test&set object, `TS[m]`, in the last allocated location, `m`, is 0. If so, it returns FULL. Otherwise, it overwrites the element in `Items[m]` with $\perp$ and adds the index `m` to `used`. Afterwards, it collects the set of *hazardous* locations stored in `Hazards`. It then allocates an arbitrary location from $\{1, \ldots, n+1\}$ that is not hazardous and announces this location by writing it to `Allocated`. Next, it resets the test&set for each location in `used` that is not hazardous. Then it removes these locations from `used`. Finally, it writes x to the newly allocated location in `Items` and returns OK.

To perform a `Take`() operation, a consumer reads `Allocated`. It announces the location it read in `Hazards` and then reads the value in this location. If it is an element $x \neq \perp$, the consumer performs `t&s`() on the resettable test&set object for this location. It then clears its announcement. If the `t&s`() was successful, it returns x. If either $x = \perp$ or the `t&s`() was unsuccessful, it returns EMPTY.

If Items[a] contains an element $x \neq \perp$, but this element has already been taken from the bag, it is essential that no consumer can perform a successful t&s() on the associated test&set object, TS[a], and take the element again. To ensure this, a consumer, P_i writes a to Hazards[i] before reading Items[a]. This prevents the producer from resetting TS[a] while P_i is poised to access TS[a].

After it has set Items[m] to $\perp$, but before resetting any test&set objects, the producer will collect the locations that appear in Hazards and refrain from resetting the test&set objects associated with the hazardous locations it collected. Hence, it will not reset a test&set object which any consumer is poised to access.

Assuming that the universe of elements that can be inserted into the bag is bounded, the registers used in the implementations store bounded values and thus, the amount of space used by the implementation is bounded. Pseudocode for our implementation appears in Figure 2.

Consider any execution consisting of operations on this data structure. We linearize the operations as follows:

- A Take() operation that performs a successful t&s() on Line 44 is linearized at this step. It returns the element it last read on Line 42.
- Consider a Take() operation, tk, that returns EMPTY. If Items[a] $= \perp$ or TS[a] $= 1$ when tk read a from Allocated on Line 40, then tk is linearized at this step. Otherwise, we can show that some other Take() operation performed a successful t&s() on TS[a] after tk performed Line 40, but before tk returned. In this case, tk is linearized immediately after the first such Take() operation.
- An Insert(x) operation that reads 0 from TS[m] (on Line 28) is linearized at this read. It returns FULL.
- An Insert(x) operation that writes x on Line 37 is linearized at this write. It returns OK.

Because the code contains no unbounded loops, the implementation is wait-free. Keeping track of hazardous locations enables objects to be safely reused so that bounded space is used. However, this makes the algorithm and its proof of correctness more intricate. In the full version of the paper [10], we prove that Figure 2 is a linearizable implementation of a 1-bounded bag. Appendix C shows that it is not strongly-linearizable.

6 A Lock-Free, Strongly-Linearizable Implementation of a 1-Bounded Bag with a Single Producer

In this section, we present a lock-free, strongly-linearizable implementation of a bag that can contain at most one element. It is shared by n processes, $P_1, \ldots, P_n$, called *consumers*, that can perform Take() and a single process, called the *producer*, that can perform Insert(x). It combines ideas from our lock-free, strongly-linearizable implementation of an unbounded bag in Section 4 and our wait-free, linearizable implementation of a 1-bounded bag in Section 5. To keep the space bounded, we use an ABA-detecting register for Done, instead of a readable fetch&increment object. An ABA-detecting register has a lock-free, strongly-linearizable implementation from registers using bounded space [20].

An Insert(x) operation by the producer is the same as in the wait-free linearizable implementation presented in the previous section, except the producer writes to the ABA-detecting register Done before it returns OK.

To perform a Take() operation, a consumer, P_i, reads Done and Allocated. In Hazards[i], it announces the location, a, that it read from Allocated. Then P_i reads Items[a] and, if it contains an element $x \neq \perp$, then P_i performs t&s() on TS[a]. Next P_i

```
49 Shared variables:
50    Items[1..n+1]: an array of single-writer registers, all initialized to ⊥,
           which can only be written to by the producer
51    TS[1..n+1]: an array of readable, resettable test&set objects, all
          initialized to 0, except for TS[1], which is initialized to 1
52    Allocated: a single-writer register, initialized to 1, which can only be
          written to by the producer
53    Hazards[1..n]: an array of single-writer registers, all initialized to ⊥,
           where Hazards[i] can only be written to by P_i
54    Done: an ABA-detecting register
55 Persistent local variables of the producer:
56    used: a register, initialized to ∅, which contains a subset of
          {1,...,n+1}
57    m: a register, initialized to 1, which is a local copy of Allocated
58 Insert(x) by the producer:
59    if TS[m].read() = 0 then return FULL
60    Items[m].write(⊥)
61    used ← used ∪ {m}
62    hazardous ← COLLECT(Hazards)
63    m ← some index in {1,...,n+1} - hazardous
64    Allocated.write(m)
65    for all i ∈ used - hazardous do
66       TS[i].reset()
67    used ← used ∩ hazardous
68    Items[m].write(x)
69    Done.dWrite()
70    return OK
71 Take() by P_i, for i ∈ {1,...,n}:
72    Done.dRead()
73    repeat
74       a ← Allocated.read()
75       Hazards[i].write(a)
76       x ← Items[a].read()
77       if x ≠ ⊥ then
78          if TS[a].t&s() = 0 then
79             Hazards[i].write(⊥)
80             return x
81       Hazards[i].write(⊥)
82       if Done.dRead() = false then return EMPTY
```

Figure 3 A lock-free, strongly-linearizable 1-bounded bag with one producer and n consumers.

clears its announcement. If the t&s() was successful, P_i returns x. If either $x = \perp$ or the t&s() was unsuccessful, P_i rereads Done and, if its value has not changed since P_i's previous read, P_i returns EMPTY. Otherwise, P_i repeats the entire sequence of steps. Pseudocode for our implementation appears in Figure 3.

Linearizability. Consider any execution consisting of operations on this data structure. We linearize the operations as follows:

- A Take() operation that performs a successful t&s() on Line 78 is linearized when it performs this t&s(). It returns the value it read in its last execution of Line 76.
- A Take() operation that gets false from Done.dRead() on Line 82 is linearized at this step. It returns EMPTY.
- An Insert(x) operation that reads 0 from TS[m] (on Line 59) is linearized at this read. It returns FULL.

- Consider an `Insert`(x) operation, *ins*, that allocated location `a` to `m` on Line 63 and wrote x to `Items[a]` on Line 68. If some `Take`() operation performs a successful `t&s`() on `TS[a]` after *ins* performed Line 68, but before *ins* performs `Done.dRead()` on Line 69, then *ins* is linearized immediately before the `Take`() operation that performed the first such `t&s`(). In this case, *ins* returns OK and we say that these `Insert`(x) and `Take`() operations are *coupled*.
- An `Insert`(x) operation that performs `Done.dWrite()` on Line 69, but has not already been linearized as a coupled operation, is linearized at this step. In this case, it returns OK and we say that it is *uncoupled*.

The proof that Figure 3 is a lock-free, strongly-linearizable implementation of a b-bounded bag appears in the full version of the paper [10].

7 A Lock-Free Strongly-Linearizable Implementation of a b-Bounded Bag with a Single Producer

Our final algorithm is a lock-free, strongly-linearizable implementation of a bag that can contain at most b elements. It is shared by n processes, $P_1, \ldots, P_n$, called *consumers*, that can perform `Take`() and a single process, called the *producer*, that can perform `Insert`(x). It extends our lock-free, strongly-linearizable implementation of a 1-bounded bag in Section 6. In this algorithm, `Allocated` is a shared register that stores a subset of at most b locations, and `alloc` is a local variable the producer uses to update the contents of `Allocated`. The challenge in designing a strongly-linearizable bag algorithm from interfering objects was for an operation that is trying to take an element from the bag to detect when the bag is empty (and then return EMPTY). This is addressed using an ABA-detecting register, `InsertDone`, to which each `Insert`(x) operation writes after writing x to `Items`. When designing a strongly-linearizable *bounded* bag algorithm from interfering objects, an operation that is trying to insert an element into the bag faces a symmetrical challenge, as it needs to detect when the bag is full (and then return FULL). To address this, we use another ABA-detecting register, `TakeDone`, to which each `Take`() operation writes after performing a successful `t&s`(). Unsuccessful `Take` operations also write to `TakeDone` before returning, to help them be linearized before the unsuccessful `Take`.

The producer begins an `Insert`(x) operation by reading `TakeDone`. Then it looks at each of the locations in `TS` that have been allocated. For each location m that contains 1, it clears the element in that location (setting `Items[m]` to $\perp$), removes `m` from `alloc`, and adds `m` to `used`. If `alloc` contains less than b locations, it adds x to the bag as follows. It first collects the non-$\perp$ values from the `Hazards` array into the set `hazardous`. Afterwards, it allocates an arbitrary location, `m`, from $\{1, \ldots, n+b\}$, excluding those in `alloc` and in `hazardous`, adds this location to `alloc`, and announces it by copying `alloc` into `Allocated`. Next, for each location in `used` that is not in `hazardous`, it resets the associated test&set object and removes the location from `used`. Then it writes x to the allocated object, `Items[m]`. Finally, it writes to the ABA-detecting register `InsertDone` and returns OK. If `alloc` contained b locations, it rereads `TakeDone` and, if its value has not changed since its previous read, it returns FULL; otherwise, it repeats the entire sequence of steps.

To perform a `Take`() operation, a process reads `InsertDone` and `Allocated`. For each location a in the set of locations obtained from `Allocated`, it announces a in `Hazards` and then reads the element, `x` from `Items[a]`. If $x \neq \perp$, it performs `t&s`() on the associated test&set object, `TS[a]`, and, if successful, it clears its announcement, writes to `TakeDone`, and returns x. If the process completes the for loop, it clears its announcement, and

rereads InsertDone. If the value of InsertDone has not changed since its previous read, it writes to TakeDone and returns EMPTY. Otherwise, it repeats the entire sequence of steps. Pseudocode for our implementation appears in Figure 4.

```
83      Shared variables:
84      Items[1..n+b]: an array of single-writer registers, each initialized
            to ⊥, which can only be written to by the producer
85      TS[1..n+b]: an array of readable, resettable test&set objects, each
            initialized to 0
86      Allocated: a single-writer register, initialized to ∅, which contains
            a subset of {1,...,n+b} of size between 0 and b and can only be
            written to by the producer
87      Hazards[1..n]: an array of single-writer registers, each initialized
            to ⊥, where Hazards[i] can only be written to by P_i
88      InsertDone, TakeDone: ABA-detecting registers
89      Persistent local variables of the producer:
90      used: a register, initialized to ∅, which contains a subset of
            {1,...,n+b}
91      alloc: a register, initialized to ∅, which is used to update Allocated
92      Insert(x) by the producer:
93      TakeDone.dRead()
94      repeat
95      for all m ∈ alloc do
96      if TS[m].read() = 1 then
97      Items[m].write(⊥)
98      alloc ← alloc - {m}
99      used ← used ∪ {m}
100     if |alloc| < b then
101     hazardous ← COLLECT(Hazards)
102     m ← some index in {1,...,n+b} - alloc - hazardous
103     alloc ← alloc ∪ {m}
104     Allocated.write(alloc)
105     for all i ∈ used - hazardous do
106     TS[i].reset()
107     used ← used ∩ hazardous
108     Items[m].write(x)
109     InsertDone.dWrite()
110     return OK
111     else if TakeDone.dRead() = false then
112     return FULL
113     Take() by P_i, for i ∈ {1,...,n}:
114     InsertDone.dRead()
115     repeat
116     allocated ← Allocated.read()
117     for all a ∈ allocated do
118     Hazards[i].write(a)
119     x ← Items[a].read()
120     if x ≠ ⊥ then
121     if TS[a].t&s() = 0 then
122     Hazards[i].write(⊥)
123     TakeDone.dWrite()
124     return x
125     Hazards[i].write(⊥)
126     if InsertDone.dRead() = false then
127     TakeDone.dWrite()
128     return EMPTY
```

Figure 4 A lock-free, strongly-linearizable b-bounded bag with one producer and n consumers.

Linearizability. Consider any execution consisting of operations on this data structure. We linearize the operations as follows:

- An `Insert` operation that obtains `false` when it reads `TakeDone` on Line 111 is linearized at this read. It returns FULL.
- Consider an `Insert(x)` operation, *ins*, that allocated location a to `m` on Line 102 and wrote x to `Items[a]` on Line 108. Suppose that, while it is poised to write to `InsertDone` on Line 109, some `Take` operation, *tk*, performs a successful `t&s()` on `TS[a]` and then some (possibly different) `Take` operation writes to `TakeDone`. In this case, *ins* is linearized immediately before *tk*, *ins* returns OK, and we say that *ins* and *tk* are *coupled*.
  In the full version of the paper [10], we prove that at most one successful `t&s()` can be performed on `TS[a]` after *ins* writes to `Items[a]` and before it writes to `InsertDone`. Hence, an `Insert` operation is coupled with at most one `Take` operation.
- Suppose that an `Insert` operation, *ins*, writes to `InsertDone` on Line 109, but has not already been linearized as a coupled operation. Then, *ins* is linearized when it performs this write and returns OK, and we say that *ins* is *uncoupled*.
- Consider a `Take` operation, *tk*, that performs a successful `t&s()` on `TS[a]` on Line 121. It is linearized at the first among the following events to occur after the successful `t&s()` by *tk*: a write to `TakeDone` by any `Take` operation on Line 123 or Line 127, a read of 1 from `TS[a]` on Line 96, and a write to `InsertDone` on Line 109 by an uncoupled `Insert` that wrote to `Items[a′]` for some $a' \neq a$. Multiple successful `Take` operations linearized at the same step are ordered arbitrarily and before any other operations linearized at this step, with one exception: a coupled `Insert` is linearized right before its coupled `Take` operation. In all cases, *tk* returns the value it last read on Line 119.
- Consider a `Take` operation, *tk*, that obtains `false` when it reads `InsertDone` on Line 126. If no uncoupled `Insert` operation writes to `InsertDone` while *tk* is poised to write to `TakeDone`, then *tk* is linearized at its write to `TakeDone` (after any other operations linearized at this step). Otherwise, *tk* is linearized at the first such write to `InsertDone`, before the `Insert` and after any successful `Take` operations linearized at this step. The ordering among unsuccessful `Take` operations linearized at this write is arbitrary. In both cases, *tk* returns EMPTY.

To make the linearization points clearer, we also list the types of steps at which operations are linearized. For each, we specify the operations that can be linearized there. If multiple operations can be linearized at the same step, we specify the ordering of these operations.

- An `Insert` reads 1 from `TS[m]` on Line 96:
  - A **successful `Take`** that previously performed a successful `t&s` on `TS[m]`.
- A **successful uncoupled `Insert`**, *ins*, which wrote to `Items[m]`, writes to `InsertDone` on Line 109:
  - **Successful `Take`s** that previously performed a successful `t&s` on `TS[m′]` for $m' \neq m$, arbitrarily ordered.
  - **Unsuccessful `Take`s**, arbitrarily ordered.
  - *ins*.
- An **unsuccessful `Insert`**, *ins*, obtains false from `TakeDone.dRead()` on Line 111:
  - *ins*.
- A **successful `Take`** writes to `TakeDone` on Line 123:
  - **Successful `Take`s**, arbitrarily ordered.
    - **Successful coupled `Insert`s**, each immediately before its coupled `Take`.

- An **unsuccessful `Take`** *tk* writes to `TakeDone` on Line 127:
  - **Successful `Takes`**, arbitrarily ordered.
    - **Successful coupled `Inserts`**, each immediately before its coupled `Take`.
  - *tk*.

Next, we provide some intuition explaining our choice of linearization points, as well as why unsuccessful `Take` operations write to `TakeDone` on Line 127 before returning EMPTY.

We pick the linearization point for an `Insert` operation as we do in Section 6: a successful uncoupled Insert is linearized at its write to `InsertDone`, a successful coupled `Insert` operation is linearized right before its coupled `Take` operation, and an unsuccessful `Insert` operation is linearized when it obtains false from `TakeDone.dRead()` on Line 111.

We want to linearize a successful `Take` operation at its write to `TakeDone`, which is intended to notify the producer that the bag is not full, similarly to how a successful uncoupled `Insert` operation notifies the consumers that the bag is not empty. However, in the following four situations, we linearize a successful `Take` operation earlier than its write to `TakeDone`.

Suppose a successful `Take` operation, *tk*, has performed a successful `t&s` operation on Line 121, but has not yet been linearized when another successful `Take` operation writes to `TakeDone`. Then *tk* is linearized at this write. Linearizing *tk* at the earliest possible write to `TakeDone` in this case is done to simplify the linearization.

An unsuccessful `Take` operation, *tk*, cannot be linearized when it obtains false from `InsertDone.dRead()`, because there might be successful `Take` operations that remove elements from the bag, but are not yet linearized. These operations must be linearized before *tk*. To ensure this, *tk* writes to `TakeDone` before returning EMPTY. This linearizes every successful `Take` operation that has performed a successful `t&s` operation on Line 121, but has not been linearized before the write to `TakeDone` by *tk*. In this case, *tk* is linearized at this write, after all such successful `Take` operations.

A coupled `Take` operation witnesses that an `Insert` operation has happened and causes it to be linearized. Similarly, an `Insert` operation, *ins*, that witnesses a successful `Take` operation causes it to be linearized: If *ins* reads 1 from `TS[a]` on Line 96, but the `Take` that last performed a successful `TS[a].t&s()` is not yet linearized, then the `Take` is linearized at this `read`, which is before *ins* removes *a* from `alloc` on Line 98. Then, when *ins* sees that `|alloc|` $< b$ on Line 100, the bag is not full and it may insert a new element.

Suppose an unsuccessful `Take` operation, *tk*, got false from `InsertDone.dRead()`, but has not yet written to `TakeDone` when an uncoupled `Insert` operation, *ins*, writes to `InsertDone` on Line 109. Then *tk* must be linearized while the bag is still empty. Hence *tk* is linearized before *ins*, at this `dWrite`. Each successful `Take` operation that performed a successful `t&s` on Line 121, but has not yet been linearized, is also linearized at this step, before *tk*, to ensure the bag is empty when *tk* is linearized. There is one exception: If *ins* wrote to `Items[a]` on Line 108, a `Take` operation that performed a successful `TS[a].t&s()` at the same location after this write should be linearized after *ins*. A `Take` operation that performed a successful `TS[a].t&s()` at the same location before this write is guaranteed to be linearized before *ins*. This is because it is linearized at or before the producer last read 1 from `TS[a]` on Line 96, which occurs before the producer removes location *a* from `alloc` on Line 98, which, in turn, must occur before *ins* is allocated location *a* on Line 102.

The proof that Figure 4 is a lock-free, strongly-linearizable implementation of a *b*-bounded bag appears in the full version of the paper [10].

8 Discussion

This paper explores strongly-linearizable implementations of bags from interfering objects. We presented the first lock-free, strongly-linearizable implementation of a bag from interfering objects, which is, interestingly, also a linearizable implementation of a queue, but not a strongly-linearizable implementation. We also presented several implementations of bounded bags with a single producer from interfering objects: a wait-free, linearizable implementation of a 1-bounded bag, a lock-free, strongly-linearizable implementation of a 1-bounded bag, and a lock-free, strongly-linearizable implementation of a b-bounded bag for any value of b.

A direct extension of this work is investigating whether it is possible to extend our bounded bag implementations to support two producers or multiple producers but only one or two consumers. Other open questions are whether there are wait-free, strongly-linearizable implementations of ABA-detecting registers and bags from interfering objects. It would also be interesting to construct a lock-free, strongly-linearizable implementation of a bag from interfering objects, such that, at every point in the execution, the number of interfering objects used is bounded above by a function of the number of processes and the number of elements the bag contains. More broadly, exploring strongly-linearizable implementations of objects beyond bags using interfering objects is a compelling direction for future work.

References

1 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 4(20):239–252, 2007. doi:10.1007/s00446-007-0023-3.

2 Zahra Aghazadeh and Philipp Woelfel. On the time and space complexity of aba prevention and detection. In *PODC*, pages 193–202, 2015. doi:10.1145/2767386.2767403.

3 Hagit Attiya, Armando Castañeda, and Constantin Enea. Strong linearizability using primitives with consensus number 2. In *PODC*, pages 432–442, 2024. doi:10.1145/3662158.3662790.

4 Hagit Attiya, Armando Castañeda, and Constantin Enea. Strong linearizability using primitives with consensus number 2. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.13618.

5 Hagit Attiya, Armando Castañeda, and Danny Hendler. Nontrivial and universal helping for wait-free queues and stacks. *J. Parallel Distributed Comput.*, 121:1–14, 2018. doi:10.1016/j.jpdc.2018.06.004.

6 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. In *DISC*, pages 2;1–2:17, 2019. doi:10.4230/LIPIcs.DISC.2019.2.

7 Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! In *PODC*, pages 363–372, 1994. doi:10.1145/197917.198126.

8 Matei David. A single-enqueuer wait-free queue implementation. In *DISC*, volume 3274 of *Lecture Notes in Computer Science*, pages 132–143, 2004. doi:10.1007/978-3-540-30186-8_10.

9 Oksana Denysyuk and Philipp Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 60–74, 2015. doi:10.1007/978-3-662-48653-5_5.

10 Faith Ellen and Gal Sela. Strong linearizability without compare&swap: The case of bags. *arXiv preprint*, 2025. arXiv:2411.19365.

11 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *STOC*, pages 373–382, 2011. doi:10.1145/1993636.1993687.

12 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *PODC*, pages 385–394, 2012. doi:10.1145/2332432.2332508.

13 Maurice Herlihy. Wait-free synchronization. *TOPLAS*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

14 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

15 Steven Munsu Hwang and Philipp Woelfel. Strongly linearizable linked list and queue. In *OPODIS*, pages 28–1, 2021. doi:10.4230/LIPIcs.OPODIS.2021.28.

16 Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. Master's thesis, University of Toronto, 2001. Department of Computer Science.

17 Maged M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *TPDS*, 15(6):491–504, 2004. doi:10.1109/TPDS.2004.8.

18 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996. doi:10.1145/248052.248106.

19 Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distributed Comput.*, 51(1):1–26, 1998. doi:10.1006/jpdc.1998.1446.

20 Sean Ovens and Philipp Woelfel. Strongly linearizable implementations of snapshots and other types. In *PODC*, pages 197–206, 2019. doi:10.1145/3293611.3331632.

21 R Kent Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, 1986.

A An Implementation of a Bag that is not Strongly-Linearizable

Consider the following lock-free, linearizable implementation of a queue [16]. We show that it is not a strongly-linearizable implementation of a bag.

```
129 Shared variables:
130    Items[1..]: an infinite array of registers, each initialized to ⊥
131    TS[1..]: an infinite array of test&set objects, each initialized to 0
132    Max: a readable fetch&increment object, initialized to 1
133 Insert(x):
134    max ← Max.f&i()
135    Items[max].write(x)
136 Take():
137    taken_old ← 0
138    max_old ← 0
139    repeat
140       taken_new ← 0
141       max_new ← Max.read() - 1
142       for i ← 1 to max_new do
143          x ← Items[i].read()
144          if x ≠ ⊥ then
145             if TS[i].t&s() = 0 then return x
146             taken_new ← taken_new + 1
147       if (taken_new = taken_old) and (max_new = max_old) then return EMPTY
148       taken_old ← taken_new
149       max_old ← max_new
```

Figure 5 An implementation of a queue that is not a strongly-linearizable bag.

Let ins_1 be a call of `Insert(1)`, let ins_2 be a call of `Insert(2)`, and let tk be a call of `Take()`. Consider the following execution α of their algorithm:

- ins_1 performs its f&i of `Max` on Line 134.
- ins_2 performs its f&i of `Max` on Line 134.

- tk performs Line 137, Line 138, and does an entire iteration of the repeat loop starting on Line 139, in which it sets *max_new* to 2 on Line 141 and sets *max_old* to 2 on line Line 149.
- tk starts a second iteration of the repeat loop, in which it sets *max_new* to 2 on Line 141 and reads $\perp$ from `Items[1]` on Line 143. However, it does not read `Items[2]` on Line 143.
- ins_1 writes 1 to `Items[1]` on Line 135 and returns.

Since ins_1 completes in α, it must be linearized in α.

Let α_1 be the continuation of α in which tk runs by itself to completion. It returns EMPTY as it encounters $\perp$ in every entry of `Items` it reads. Since tk returns EMPTY in $\alpha \cdot \alpha_1$, it must be linearized before ins_1. Hence, tk must be linearized in α.

Let α_2 be another continuation of α, in which ins_2 writes 2 to `Items[2]` on Line 135 and returns, and then tk runs by itself to completion. From the code, tk reads 2 from `Items[2]` and returns 2. This implies that ins_2 must be linearized before tk in $\alpha \cdot \alpha_2$ and, hence, in α. Therefore tk returns 2 in $\alpha \cdot \alpha_1$, which is a contradiction.

This shows that the algorithm in Figure 5 is not a strongly-linearizable implementation of a bag for any choice of linearization points. In particular, there is a problem with the linearization points mentioned in [4]:

- Each `Insert` operation is linearized when it performs Line 135.
- Each successful `Take` operation is linearized when it performs a successful `test&set` on Line 145.
- Each unsuccessful `Take` operation is linearized when it last reads `Max` on Line 141.

The issue is that, when an unsuccessful `Take` operation is linearized, the bag might not be empty. Let ins be a call of `Insert(1)` and let tk_1 and tk_2 be calls of `Take()`. Consider the following execution:

- ins performs its f&i of `Max` on Line 134.
- tk_1 performs Line 137, Line 138, and does an entire iteration of the repeat loop starting on Line 139 in which it sets *max_new* to 1 on Line 141 and sets *max_old* to 1 on line Line 149.
- ins writes 1 to `Items[1]` on Line 135 and returns.
- tk_1 starts a second iteration of the repeat loop, in which it sets *max_new* to 1 on Line 141.
- tk_2 performs a `Take` operation, returning the value 1 when it first performs Line 145.
- tk_1 completes its `Take` operation, returning EMPTY when it returns on Line 147.

Note that, when tk_1 last performs Line 141, where it is linearized, the bag contains the element 1.

B A Proof that Figure 1 is not a Strongly-Linearizable Implementation of a Queue

We show that our strongly-linearizable implementation of a bag is not a strongly-linearizable implementation of a queue.

Let ins_1 be a call of `Insert(1)`, let ins_2 be a call of `Insert(2)`, and let tk, tk_1, and tk_2 be calls of `Take()`. Consider the following execution α of the algorithm in Figure 1:

- ins_1 performs its f&i of `Allocated` on Line 7 and is allocated location 1.
- ins_2 performs its f&i of `Allocated` on Line 7 and is allocated location 2.
- tk reads 2 from `Allocated` on Line 13 and then reads $\perp$ from `Items[1]` on Line 15.
- ins_1 and ins_2 run to completion.

Since ins_1 and ins_2 complete in α, they must be linearized in α.

Let α_1 be the continuation of α in which tk runs by itself to completion. It reads 2 from `Items[2]` on Line 15 during its second iteration of the for loop and returns 2. To satisfy the semantics of a queue, ins_2 must be linearized before ins_1 in $\alpha \cdot \alpha_1$ and, hence, in α.

Let α_2 be another continuation of α, in which tk_1 runs by itself to completion and then tk_2 runs by itself to completion. From the code, tk_1 reads 1 from `Items[1]` and returns 1, and tk_2 reads 2 from `Items[2]` and returns 2. To satisfy the semantics of a queue, tk_1 must be linearized before tk_2 in $\alpha \cdot \alpha_2$ and, hence, in α. This is a contradiction.

C A Proof that Figure 2 is not a Strongly-Linearizable Implementation of a 1-Bounded Bag

We show that our wait-free implementation of a 1-bounded bag with a single producer is not strongly-linearizable.

Let ins_i be a call of `Insert(`i`)` and tk_i a call of `Take()`, for $1 \leq i \leq 3$. Consider the following execution α of the algorithm in Figure 2:

- ins_1 starts running, and it is allocated location 1 on Line 32. It then runs to completion.
- tk_1 runs and returns 1.
- tk_2 reads 1 from `Allocated` on Line 40.
- ins_2 starts running. It writes $\perp$ to `Items[1]` on Line 29. It is allocated location 2 on Line 32. It then runs to completion.

Let α_1 be the continuation of α in which tk_2 runs by itself to completion. It returns EMPTY as it encounters $\perp$ in every entry of `Items` it reads including `Items[1]`. Since tk_2 returns EMPTY in $\alpha \cdot \alpha_1$, it must be linearized before ins_2. Hence, tk_2 must be linearized in α, with return value EMPTY.

Let α_2 be another continuation of α, in which the following occur:

- tk_3 runs and returns 2.
- ins_3 starts running. It is allocated location 1 on Line 32. It continues to run, writes 3 to `Items[1]` and returns.
- tk_2 runs to completion.

From the code, tk_2 reads 3 from `Items[1]` and returns 3. Hence, tk_2 must be linearized after ins_3 in $\alpha \cdot \alpha_2$. This contradicts strong-linearizability.