Space and Time Bounded Multiversion Garbage Collection

FORTH ICS TR 482

Naama Ben-David
VMware Research, USA

Guy E. Blelloch
Carnegie Mellon University, USA

Panagiota Fatourou
Université Paris Cité, LIPADE and FORTH ICS and University of Crete, Greece

Eric Ruppert
York University, Canada

Yihan Sun
University of California, Riverside, USA

Yuanhao Wei
Carnegie Mellon University, USA

Abstract

We present a general technique for garbage collecting old versions for multiversion concurrency control that simultaneously achieves good time and space complexity. Our technique takes only $O(1)$ time on average to reclaim each version and maintains only a constant factor more versions than needed (plus an additive term). It is designed for multiversion schemes using version lists, which are the most common.

Our approach uses two components that are of independent interest. First, we define a novel *range-tracking data structure* which stores a set of old versions and efficiently finds those that are no longer needed. We provide a wait-free implementation in which all operations take amortized constant time. Second, we represent version lists using a new lock-free doubly-linked list algorithm that supports efficient (amortized constant time) removals given a pointer to any node in the list. These two components naturally fit together to solve the multiversion garbage collection problem–the range-tracker identifies which versions to remove and our list algorithm can then be used to remove them from their version lists. We apply our garbage collection technique to generate end-to-end time and space bounds for the multiversioning system of Wei et al. (PPoPP 2021).

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

Keywords and phrases Lock-free, data structures, memory management, snapshot, version lists

Funding *Guy E. Blelloch and Yuanhao Wei: NSF CCF-1901381, CCF-1910030, and CCF-1919223. Eric Ruppert: NSERC Discovery Grant. Yihan Sun: NSF grant CCF-2103483. Panagiota Fatourou: EU Horizon 2020, Marie Sklodowska-Curie GA No 101031688.*

Acknowledgements *We thank the anonymous DISC 2021 referees for their helpful comments and suggestions.*

1 Introduction

Supporting multiple "historical" versions of data, often called multiversioning or multiversion concurrency control, is a powerful technique widely used in database systems [44, 11, 40, 34, 38, 53], transactional memory [42, 24, 41, 33, 31], and shared data structures [8, 23, 37, 51]. This approach allows complex queries (read-only transactions) to proceed concurrently with

updates while still appearing atomic because they get data views that are consistent with a single point in time. If implemented carefully, queries do not interfere with one another or with updates. The most common approach for multiversioning uses version lists [44] (also called version chains): the system maintains a global timestamp that increases over time, and each object maintains a history of its updates as a list of value-timestamp pairs, each corresponding to a value written and an update time. Each node in the list has an associated interval of time from that node's timestamp until the next (later) node's timestamp. A query can first read a timestamp value t and then, for each object it wishes to read, traverse the object's version list to find the version whose interval contains t.

Memory usage is a key concern for multiversioning, since multiple versions can consume huge amounts of memory. Thus, most previous work on multiversioning discusses how to reclaim the memory of old versions. We refer to this as the multiversion garbage collection (MVGC) problem. A widely-used approach is to keep track of the earliest active query and reclaim the memory of any versions overwritten before the start of this query [24, 38, 32, 37, 51]. However, a query that runs for a long time, either because it is complicated or because it has been delayed, will force the system to retain many unneeded intermediate versions between the oldest required version and the current one. This has been observed to be a major bottleneck for database systems with Hybrid Transaction and Analytical Processing (HTAP) workloads [16] (i.e., many small updates concurrent with some large analytical queries). To address this problem in the context of software transactional memory, Lu and Scott [35] proposed a non-blocking algorithm that can reclaim intermediate versions. Blocking techniques were later proposed by the database community [16, 34]. However, these techniques add significant time overhead in worst-case executions.

We present a wait-free MVGC scheme that achieves good time and space bounds, using $O(1)$ time[1] on average per allocated version and maintaining only a constant factor more versions than needed (plus an additive term). The scheme is very flexible and it can be used in a variety of multiversioning implementations. It uses a three-step approach that involves 1) identifying versions that can be reclaimed, including intermediate versions, 2) unlinking them from the version lists, and 3) reclaiming their memory. To implement these three steps efficiently, we develop two general components—a *range-tracking* data structure and a *version-list* data structure—that could be of independent interest beyond MVGC.

The range-tracking data structure is used to identify version list nodes that are no longer needed. It supports an `announce` operation that is used by a query to acquire the current timestamp t as well as protect any versions that were current at t from being reclaimed. A corresponding `unannounce` is used to indicate when the query is finished. The data structure also supports a `deprecate` operation that is given a version and its time interval, and indicates that the version is no longer the most recent—i.e., is safe to reclaim once its interval no longer includes any announced timestamp. When a value is updated with a new version, the previous version is deprecated. A call to `deprecate` also returns a list of versions that had previously been `deprecated` and are no longer cover any announced timestamp—i.e., are now safe to reclaim. We provide a novel implementation of the range-tracking data structure for which the amortized number of steps per operation is $O(1)$. We also bound the number of versions on which `deprecate` has been called, but have not yet been returned. If H is the maximum, over all configurations, of the number of needed deprecated versions, then the number of deprecated versions that have not yet been returned is at most $2H + O(P^2 \log P)$, where P is the number of processes. To achieve these time and space bounds, we borrow some

[1] For time/space complexity, we count both local and shared memory operations/objects.

ideas from real-time garbage collection [7, 12], and add several new ideas such as batching and using a shared queue.

The second main component of our scheme is a wait-free version-list data structure that supports efficient (amortized constant time) removals of nodes from anywhere in the list. When the `deprecate` operation identifies an unneeded version, we must splice it out of its version list, without knowing its current predecessor in the list, so we need a doubly-linked version list. Our doubly-linked list implementation has certain restrictions that are naturally satisfied when maintaining version lists, for example nodes may be appended only at one end. The challenge is in achieving constant amortized time per remove, and bounded space. Previously known concurrent doubly-linked lists [49, 45] do not meet these requirements, requiring at least $\Omega(P)$ amortized time per remove. We first describe the implementation of our version list assuming a garbage collector and then we show how to manually reclaim removed nodes while maintaining our desired overall time and space bounds.

To delete elements from the list efficiently, we leverage some recent ideas from randomized parallel list contraction [13], which asynchronously removes elements from a list. To avoid concurrently splicing out adjacent elements in the list, which can cause problems, the approach defines an implicit binary tree so that the list is an in-order traversal of the tree. Only nodes corresponding to leaves of the tree, which cannot be adjacent in the list, may be spliced out. Directly applying this technique, however, is not efficient in our setting. To reduce space overhead, we had to develop intricate helping mechanisms for splicing out internal nodes rather than just leaves. To achieve wait-freedom, we had to skew the implicit tree so that it is right-heavy. The final algorithm ensures that at most $2(L - R) + O(P \log L_{max})$ nodes remain reachable in an execution with L appends and R removes across an arbitrary number of version lists, and at most L_{max} appends on a single version list. This means the version lists store at most a constant factor more than the $L - R$ required nodes plus an additive term shared across all the version lists. Combining this with the bounds from the range tracker, our MVGC scheme ensures that at most $O(V + H + P^2 \log P + P \log L_{max})$ versions are reachable from the V version lists. This includes the current version for each list, H needed versions, plus additive terms from the range tracking and list building blocks.

After a node has been spliced out of the doubly-linked list, its memory must be reclaimed. This step may be handled automatically by the garbage collector in languages such as Java, but in non-garbage-collected languages, additional mechanisms are needed to safely reclaim memory. The difficulty in this step is that while a node is being spliced out, other processes traversing the list might be visiting that node. We use a reference counting reclamation scheme and this requires modifying our doubly-linked list algorithm slightly to maintain the desired space bounds. We apply an existing concurrent reference counting implementation [2] that employs a local hash table per process which causes the time bounds of our reclamation to become amortized $O(1)$ *in expectation*. It also requires an additional fetch-and-add instruction, whereas the rest of our algorithms require only read and CAS.

We apply our MVGC scheme to a specific multiversioning scheme [51] to generate end-to-end bounds for a full multiversioning system. This multiversioning scheme takes a given CAS-based concurrent data structure and transforms it to support complex queries (e.g., range queries) by replacing each CAS object with one that maintains a version list. Overall, we ensure that the memory usage of the multiversion data structure is within a constant factor of the needed space, plus $O(P^2 \log P + P^2 \log L_{max})$. In terms of time complexity, our garbage collection scheme takes only $O(1)$ time on average for each allocated version.

Detailed proofs of correctness and of our complexity bounds appear in the full version [9].

2 Related Work

Garbage Collection. One of the simplest, oldest techniques for garbage collection is reference counting (RC) [18, 19, 30]. In its basic form, RC attaches to each object a counter of the number of references to it. An object is reclaimed when its counter reaches zero. Some variants of RC are wait-free [2, 48]. In Section 6, we apply the RC scheme of [2] to manage version list nodes as it adds only constant time overhead (in expectation) and it is the only concurrent RC scheme that maintains our desired time bounds.

Epoch-based reclamation (EBR) [25, 17] employs a counter that is incremented periodically and is used to divide the execution into epochs. Processes read and announce the counter value at the beginning of an operation. An object can be reclaimed only if it was retired in an epoch preceding the oldest announced. EBR is often the preferred choice in practice, as it is simple and exhibits good performance. However, a slow or crashed process with timestamp t can prevent the reclamation of *all* retired objects with timestamps larger than t. EBR, or variants, are used in a variety of MVGC schemes [24, 38, 51] to identify versions that are older than any query. An advantage of these schemes is that identified versions can be immediately reclaimed without first being unlinked from the version lists because the section of the version list they belong to is old enough to never be traversed. However, they inherit the same problem as EBR and are not able to reclaim intermediate versions between the oldest needed version and the current version when a long-running query holds on to an old epoch. This can be serious for multiversioned systems since EBR works best when operations are short, but a key motivation for multiversioning is to support lengthy queries.

Hazard pointers (HP) [30, 36] can be used to track which objects are currently being accessed by each process and are therefore more precise. Combinations of HP and EBR have been proposed (e.g. [43, 52]) with the goal of preserving the practical efficiency of EBR while lowering its memory usage. However, unlike EBR, none of these techniques directly solve the MVGC problem. Other memory reclamation schemes have been studied that require hardware support [1, 20] or rely on the signaling mechanism of the operating system [17, 47]. Hyaline [39] implements a similar interface to EBR and can be used for MVGC, but like EBR, it cannot reclaim intermediate versions.

We are aware of three multiversioning systems based on version lists that reclaim intermediate versions: GMV [35], HANA [34] and Steam [16]. To determine which versions are safe to reclaim, all three systems merge the current version list for an object with the list of active timestamps to check for overlap. The three schemes differ based on when they decide to perform this merging step and how they remove and reclaim version list nodes. In GMV, when an update operation sees that memory usage has passed a certain threshold, it iterates through all the version lists to reclaim versions. Before reclaiming a version, it has to help other processes traverse the version list to ensure traversals remain wait-free. HANA uses a background thread to identify and reclaim obsolete versions while Steam scans the entire version list whenever a new version is added to it. In HANA and Steam, nodes are removed by locking the entire version list, whereas in GMV, nodes are removed in a lock-free manner by first logically marking a node for deletion, as in Harris's linked list [28]. If a `remove` operation in GMV experiences contention (i.e., fails a CAS), it restarts from the head of the version list. None of these three techniques ensure constant-time removal from a version list. Both Steam and GMV ensure $O(PM)$ space where M is the amount of space required in an equivalent sequential execution. In comparison, we use a constant factor more than the required space plus an additive term of $O(P^2 \log P + P^2 \log L_{max})$, where L_{max} is the maximum number of versions added to a single version list. This can be significantly less

than $O(PM)$ in many workloads.

Lock-Free Data Structures and Query Support. We use doubly-linked lists to store old versions. Singly-linked lists had lock-free implementations as early as 1995 [50]. Several implementations of doubly-linked lists were developed later from multi-word CAS instructions [6, 26], which are not widely available in hardware but can be simulated in software [29, 27]. Sundell and Tsigas [49] gave the first implementation from single-word CAS, although it lacks a full proof of correctness. Shafiei [45] gave an implementation with a proof of correctness and amortized analysis. Existing doubly-linked lists are not efficient enough for our application, so we give a new implementation with better time bounds.

Fatourou, Papavasileiou and Ruppert [23] used multiversioning to add range queries to a search tree [21]. Wei et al. [51] generalized this approach (and made it more efficient) to support wait-free queries on a large class of lock-free data structures. Nelson, Hassan and Palmieri [37] sketched a similar scheme, but it is not non-blocking. In Appendix 7, we apply our garbage collection scheme to the multiversion system of [51].

3 Preliminaries

We use a standard model with asynchronous, crash-prone processes that access shared memory using CAS, read and write instructions. For our implementations of data structures, we bound the number of steps needed to perform operations, and the number of shared objects that are allocated but not yet reclaimed.

We also use *destination objects* [15], which are single-writer objects that store a value and support `swcopy` operations in addition to standard reads and writes. A `swcopy(ptr)` atomically reads the value pointed to by `ptr`, and copies the value into the destination object. Only the owner of a destination object can perform `swcopy` and `write`; any process may `read` it. Destination objects can be implemented from CAS so that all three operations take $O(1)$ steps [15]. They are used to implement our range-tracking objects in Section 4.

Pseudocode Conventions. We use syntax similar to C++. The type `T*` is a pointer to an object of type `T`. `List<T>` is a `List` of objects of type `T`. If `x` stores a pointer to an object, then `x->f` is that object's member `f`. If `y` stores an object, `y.f` is that object's member `f`.

4 Identifying Which Nodes to Disconnect from the Version List

We present the *range-tracking* object, which we use to identify version nodes that are safe to disconnect from version lists because they are no longer needed. To answer a query, a slow process may have to traverse an entire version list when searching for a very old version. However, we need only maintain list nodes that are the potential target nodes of such queries. The rest may be spliced out of the list to improve space usage and traversal times.

We assign to each version node `X` an interval that represents the period of time when `X` was the current version. When the next version `Y` is appended to the version list, `X` ceases to be the current version and becomes a potential candidate for removal from the version list (if no query needs it). Thus, the left endpoint of `X`'s interval is the timestamp assigned to `X` by the multiversioning system, and the right endpoint is the timestamp assigned to `Y`.

We assume that a query starts by announcing a timestamp t, and then proceeds to access, for each relevant object o, its corresponding version at time t, by finding the first node in the

version list with timestamp at most t (starting from the most recent version). Therefore, an announcement of t means it is unsafe to disconnect any nodes whose intervals contain t.

As many previous multiversioning systems [24, 34, 37, 38, 51] align with the general scheme discussed above, we define the range-tracking object to abstract the problem of identifying versions that are not needed. We believe this abstraction is of general interest.

▶ **Definition 1** (Range-Tracking Object). *A range-tracking object maintains a multiset A of integers, and a set O of triples of the form* `(o,low,high)` *where* `o` *is an object of some type T and* `low` $\leq$ `high` *are integers. Elements of A are called* active announcements. *If* `(o,low,high)` $\in O$ *then* `o` *is a* deprecated object *with associated half-open interval* $[$`low`$,$`high`$)$. *The range-tracking object supports the following operations.*

- `announce(int* ptr)` *atomically reads the integer pointed to by* `ptr`*, adds the value read to A, and returns the value read.*
- `unannounce(int i)` *removes one copy of* `i` *from A, rendering the announcement inactive.*
- `deprecate(T* o, int low, int high)`*, where* `low` $\leq$ `high`*, adds the triple* `(o,low,high)` *to O and returns a set S, which contains the deprecated objects of a set $O' \subseteq O$ such that for any $o \in O'$, the interval of o does not intersect A, and removes O' from O.*

The specification of Definition 1 should be paired with a progress property that rules out the trivial implementation in which `deprecate` always returns an empty set. We do this by bounding the number of deprecated objects that have not been returned by `deprecate`.

▶ **Assumption 2.** To implement the range-tracking object, we assume the following.

1. A process's calls to `deprecate` have non-decreasing values of parameter `high`.
2. If, in some configuration G, there is a pending `announce` whose argument is a pointer to an integer variable `x`, then the value of `x` at G is greater than or equal to the `high` argument of every `deprecate` that has been invoked before G.
3. For every process p, the sequence of invocations to `announce` and `unannounce` performed by p should have the following properties: a) it should start with `announce`; b) it should alternate between invocations of `announce` and invocations of `unannounce`; c) each `unannounce` should have as its argument the integer returned by the preceding `announce`.
4. Objects passed as the first parameter to `deprecate` operations are distinct.

In the context we are working on, we have a non-decreasing integer variable that works as a global timestamp, and is passed as the argument to every `announce` operation. Moreover, the `high` value passed to each `deprecate` operation is a value that has been read from this variable. This ensures that parts 1 and 2 of Assumption 2 are satisfied. The other parts of the assumption are also satisfied quite naturally for our use of the range-tracking object, and we believe that the assumption is reasonably general. Under this assumption, we present and analyze a linearizable implementation of the range-tracking object in Section 4.1.

4.1 A Linearizable Implementation of the Range-Tracking Object

Our implementation, RangeTracker, is shown in Figure 1. Assumption 2.3 means that each process can have at most one active announcement at a time. So, RangeTracker maintains a shared array `Ann` of length P to store active announcements. `Ann[p]` is a destination object (defined in Section 3) that is owned by process `p`. Initially, `Ann[p]` stores a special value $\perp$. To announce a value, a process `p` calls `swcopy` (line 28) to copy the current timestamp into `Ann[p]` and returns the announced value (line 29). To deactivate an active announcement, p writes $\perp$ into `Ann[p]` (line 31). Under Assumption 2.3, the argument to

```
1  class Range { T* t, int low, int high; };
2  class RangeTracker {
3    // global variables
4    Destination Ann[P];
5    Queue<List<Range>> Q; //initially empty
6    // thread local variables
7    List<Range> LDPool; // initially empty
8    Array<int> sortAnnouncements() {
9      List<int> result;
10     for(int i = 0; i < P; i++) {
11       int num = Ann[i].read();
12       if(num != ⊥) result.append(num); }
13     return sort(toArray(result)); }

15   List<T*>, List<Range> intersect(
16       List<Range> MQ, Array<int> ar) {
17     Range r; int i = 0;
18     List<T*> Redundant;
19     List<Range> Needed;
20     for(r in MQ) {
21       while(i < ar.size() &&
22             ar[i] < r.high) i++;
23       if(i == 0 || ar[i-1] < r.low)
24         Redundant.append(r.t);
25       else Needed.append(r); }
26     return <Redundant, Needed>; }

27 int Announce(int* ptr) {
28   Ann[p].swcopy(ptr);
29   return Ann[p].read(); }

31 void unannounce() { Ann[p].write(⊥); }

33 List<T*> deprecate(T* o, int low, int high) {
34   List<T*> Redundant;
35   List<Range> Needed, Needed1, Needed2;
36   // local lists are initially empty
37   LDPool.append(Range(o, low, high));
38   if(LDPool.size() == B) {
39     List<Range> MQ = merge(Q.deq(),Q.deq());
40     Array<int> ar = sortAnnouncements();
41     Redundant, Needed = intersect(MQ, ar);
42     if(Needed.size() > 2*B) {
43       Needed1, Needed2 = split(Needed);
44       Q.enq(Needed1);
45       Q.enq(Needed2); }
46     else if(Needed.size() > B) {
47       Q.enq(Needed); }
48     else {
49       LDPool = merge(LDPool,Needed); }
50     Q.enq(LDPool);
51     LDPool = empty list; }
52   return Redundant; } };
```

Figure 1 Code for process p for our linearizable implementation of a range-tracking object.

unannounce must match the argument of the process's previous announce, so we suppress unannounce's argument in our code. An announce or unannounce performs O(1) steps.

A Range object (line 1) stores the triple (o,low,high) for a deprecated object o. It is created (at line 37) during a deprecate of o. RangeTracker maintains the deprecated objects as *pools* of Range objects. Each pool is sorted by its elements' high values. Each process maintains a local pool of deprecated objects, called LDPool. To deprecate an object, a process simply appends its Range to the process's local LDPool (line 37). Assumption 2.1 implies that objects are appended to LDPool in non-decreasing order of their high values.

We wish to ensure that most deprecated objects are eventually returned by a deprecate operation so that they can be freed. If a process p with a large LDPool ceases to take steps, it can cause all of those objects to remain unreturned. Thus, when the size of p's LDPool hits a threshold B, they are flushed to a shared queue, Q, so that other processes can also return them. The elements of Q are pools that each contain B to $2B$ deprecated objects. For the sake of our analysis, we choose $B = P \log P$. When a flush is triggered, p dequeues two pools from Q and processes them as a batch to identify the deprecated objects whose intervals do not intersect with the values in Ann, and return them. The rest of the dequeued objects, together with those in LDPool, are stored back into Q. We call these actions (lines 38–51), the *flush phase* of deprecate. A deprecate without a flush phase returns an empty set.

During a flush phase, a process p dequeues two pools from Q and merges them (line 39) into a new pool, MQ. Next, p makes a local copy of Ann and sorts it (line 40). It then uses the intersect function (line 41) to partition MQ into two sorted lists: Redundant contains objects whose intervals do not intersect the local copy of Ann, and Needed contains the rest. Intuitively, a deprecated object in MQ is put in Redundant if the low value of its interval is larger than the announcement value immediately before its high value. Finally, p enqueues the Needed pool with its LDPool into Q (lines 44–47 and line 50). To ensure that the size of each pool in Q is between B and $2B$, the Needed pool is split into two halves if it is too large

(line 43), or is merged with `LDPool` if it is too small (line 49). A flush phase is performed once every $P \log P$ calls to `deprecate`, and the phase executes $O(P \log P)$ steps. Therefore, the amortized number of steps for `deprecate` is $O(1)$.

The implementation of the concurrent queue `Q` should ensure that an element can be enqueued or dequeued in $O(P \log P)$ steps. The concurrent queue presented in [22] has step complexity $O(P)$ and thus ensures these bounds. To maintain our space bounds, the queue nodes must be reclaimed. This can be achieved if we apply hazard-pointers on top of the implementation in [22]. If `Q` is empty, then `Q.deq()` returns an empty list.

We sketch the proofs of the following three theorems. For detailed proofs, see [9].

▶ **Theorem 3.** *If Assumption 2 holds, then* RangeTracker *is a linearizable implementation of a range-tracking object.*

The linearization points used in the proof of Theorem 3 are defined as follows. An `announce` is linearized at its `swcopy` on line 28. An `unannounce` is linearized at its `write` on line 31. A `deprecate` is linearized at line 50 if it executes that line, or at line 37 otherwise.

The most interesting part of the proof concerns a `deprecate` operation I with a flush phase. I dequeues two pools from `Q` as `MQ` and decides which objects in `MQ` to return based on the local copy of `Ann` array. To show linearizability, we must also show that intervals of the objects returned by I do not intersect the `Ann` array at the linearization point of I. Because of Assumption 2.2, values written into `Ann` after the pools are dequeued cannot be contained in the intervals in `MQ`. Thus, if an object's interval does not contain the value I read from `Ann[i]`, it will not contain the value in `Ann[i]` at I's linearization point.

▶ **Theorem 4.** *In the worst case,* `announce` *and* `unannounce` *take* $O(1)$ *steps, while* `deprecate` *takes* $O(P \log P)$ *steps. The amortized number of steps performed by each operation is* $O(1)$.

Let H be the maximum, over all configurations in the execution, of the number of *needed* deprecated objects, i.e., those whose intervals contain an active announcement.

▶ **Theorem 5.** *At any configuration, the number of deprecated objects that have not yet been returned by any instance of* `deprecate` *is at most* $2H + 25P^2 \log P$.

At any time, each process holds at most $P \log P$ deprecated objects in `LDPool` and at most $4P \log P$ that have been dequeued from `Q` as part of a flush phase. We prove by induction that the number of deprecated objects in `Q` at a configuration G is at most $2H + O(P^2 \log P)$. Let G' be the latest configuration before G such that all pools in `Q` at G' are dequeued between G' and G. Among the dequeued pools, only the objects that were needed at G' are re-enqueued into `Q`, and there are at most H such objects. Since we dequeue two pools (containing at least B elements each) each time we enqueue B *new* objects between G' and G, this implies that the number of such new objects is at most half the number of objects in `Q` at G' (plus $O(P^2 \log P)$ objects from flushes already in progress at G'). Assuming the bound on the size of `Q` holds at G', this allows us to prove the bound at G.

The constant multiplier of H in Theorem 5 can be made arbitrarily close to 1 by dequeuing and processing k pools of `Q` in each flush phase instead of two. The resulting space bound would be $\frac{k}{k-1} \cdot H + \frac{(2k+1)(3k-1)}{k-1} \cdot P^2 \log P$. This would, of course, increase the constant factor in the amortized number of steps performed by `deprecate` (Theorem 4).

5 Maintaining Version Lists

We use a restricted version of a doubly-linked list to maintain each version list so that we can more easily remove nodes from the list when they are no longer needed. We assume each node has a timestamp field. The list is initially empty and provides the following operations.

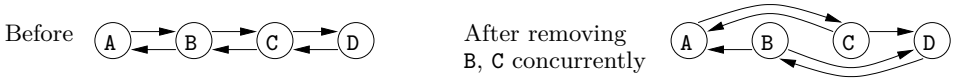

Figure 2 An example of incorrect removals.

- `tryAppend(Node* old, Node* new)`: Adds `new` to the head of the list and returns true if the current head is `old`. Otherwise returns false. Assumes `new` is not null.
- `getHead()`: Returns a pointer to the `Node` at the head of the list (or `null` if list is empty).
- `find(Node* start, int ts)`: Returns a pointer to the first `Node`, starting from `start` and moving away from the head of the list, whose timestamp is at most `ts` (or `null` if no such node exists).
- `remove(Node* n)`: Given a previously appended `Node`, removes it from the list.

To obtain an efficient implementation, we assume several preconditions, summarized in Assumption 6 (and stated more formally in the full version [9]). A version should be removed from the object's version list only if it is not current: either it has been superseded by another version (6.1) or, if it is the very last version, the entire list is no longer needed (6.2). Likewise, a version should not be removed if a `find` is looking for it (6.3), which can be guaranteed using our range-tracking object. We allow flexibility in the way timestamps are assigned to versions. For example, a timestamp can be assigned to a version after appending it to the list. However, some assumptions on the behaviour of timestamps are needed to ensure that responses to `find` operations are properly defined (6.4, 6.5).

▶ **Assumption 6.**

1. Each node (except the very last node) is removed only after the next node is appended.
2. No `tryAppend`, `getHead` or `find` is called after a `remove` on the very last node.
3. After `remove(X)` is invoked, no pending or future `find` operation should be seeking a timestamp in the interval between `X`'s timestamp and its successor's.
4. Before trying to append a node after a node `B` or using `B` as the starting point for a `find`, `B` has been the head of the list and its timestamp has been set. A node's timestamp does not change after it is set. Timestamps assigned to nodes are non-decreasing.
5. If a `find(X,t)` is invoked, any node appended after `X` has a higher timestamp than `t`.
6. Processes never attempt to append the same node to a list twice, or to remove it twice.

5.1 Version List Implementation

Pseudocode for our list implementation is in Figure 4. A `remove(X)` operation first marks the node `X` to be deleted by setting a `status` field of `X` to `marked`. We refer to the subsequent physical removal of `X` as *splicing* `X` out of the list.

Splicing a node `B` from a doubly-linked list requires finding its left and right neighbours, `A` and `C`, and then updating the pointers in `A` and `C` to point to each other. Figure 2 illustrates the problem that could arise if adjacent nodes `B` and `C` are spliced out concurrently. The structure of the doubly-linked list becomes corrupted: `C` is still reachable when traversing the list towards the left, and `B` is still reachable when traversing towards the right. The challenge of designing our list implementation is to coordinate splices to avoid this situation.

We begin with an idea that has been used for parallel list contraction [46]. We assign each node a priority value and splice a node out only if its priority is greater than both of its neighbours' priorities. This ensures that two adjacent nodes cannot be spliced concurrently.

Conceptually, we can define a *priority tree* corresponding to a list of nodes with priorities as follows. Choose the node with minimum priority as the root. Then, recursively define

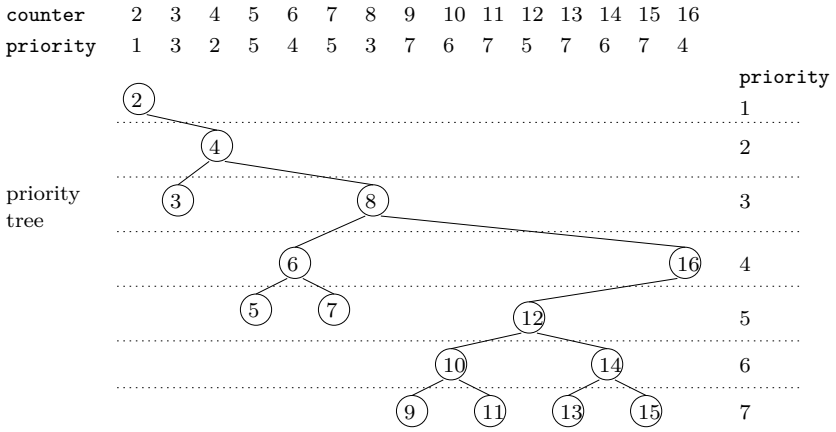

Figure 3 A list and its priority tree.

the left and right subtrees of the root by applying the same procedure to the sublists to the left and right of the root node. The original list is an in-order traversal of the priority tree. See Figure 3 for an example. We describe below how we choose priorities to ensure that (1) there is always a unique minimum in a sublist corresponding to a subtree (to be chosen as the subtree's root), and (2) if L nodes are appended to the list, the height of the priority tree is $O(\log L)$. We emphasize that the priority tree is not actually represented in memory; it is simply an aid to understanding the design of our implementation.

The requirement that a node is spliced out of the list only if its priority is greater than its neighbours corresponds to requiring that we splice only nodes whose descendants in the priority tree have all already been spliced out of the list. To remove a node that still has unspliced descendants, we simply mark it as logically deleted and leave it in the list. If `X`'s descendants have all been spliced out, then `X`'s parent `Y` in the priority tree is the neighbour of `X` in the list with the larger priority. An operation that splices `X` from the list then attempts to *help* splice `X`'s parent `Y` (if `Y` is marked for deletion and `Y` is larger than its two neighbours), and this process continues up the tree. Conceptually, this means that if a node `Z` is marked but not spliced, the last descendant of `Z` to be spliced is also responsible for splicing `Z`.

In this scheme, an unmarked node can block its ancestors in the priority tree from being spliced out of the list. For example, in Figure 3, if the nodes with counter values 10 to 16 are all marked for deletion, nodes 11, 13 and 15 could be spliced out immediately. After 13 and 15 are spliced, node 14 could be too. The unmarked node 9 prevents the remaining nodes 10, 12 and 16 from being spliced, since each has a neighbour with higher priority. Thus, an unmarked node could prevent up to $\Theta(\log L)$ marked nodes from being spliced out of the list.

Improving this space overhead factor to $O(1)$ requires an additional, novel mechanism. If an attempt to remove node `B` observes that `B`'s left neighbour `A` is unmarked and `B`'s priority is greater than `B`'s right neighbour `C`'s priority, we allow `B` to be spliced out of the list using a special-purpose routine called `spliceUnmarkedLeft`, even if `A`'s priority is greater than `B`'s. In the example of the previous paragraph, this would allow node 10 to be spliced out after 11. Then, node 12 can be spliced out after 10 and 14, again using `spliceUnmarkedLeft`, and finally node 16 can be spliced out. A symmetric routine `spliceUnmarkedRight` applies if `C` is unmarked and `B`'s priority is greater than `A`'s. This additional mechanism of splicing out nodes when one neighbour is unmarked allows us to splice out all nodes in a string of consecutive marked nodes, except possibly one of them, which might remain in the list if both its neighbours are unmarked and have higher priority. However, during the

`spliceUnmarkedLeft` routine that is splicing out `B`, `A` could become marked. If `A`'s priority is greater than its two neighbours' priorities, there could then be simultaneous splices of `A` and `B`. To avoid this, instead of splicing out `B` directly, the `spliceUnmarkedLeft` installs a pointer to a *Descriptor* object into node `A`, which describes the splice of `B`. If `A` becomes marked, the information in the Descriptor is used to *help* complete the splice of `B` before `A` itself is spliced. Symmetrically, a `spliceUnmarkedRight` of `B` installs a Descriptor in `C`.

Multiple processes may attempt to splice the same node `B`, either because of the helping coordinated by Descriptor objects or because the process that spliced `B`'s last descendant in the priority tree will also try to splice `B` itself. To avoid unnecessary work, processes use a CAS to change the status of `B` from `marked` to `finalized`. Only the process that succeeds in this CAS has the responsibility to recursively splice `B`'s ancestors. (In the case of the `spliceUnmarkedLeft` and `spliceUnmarkedRight` routines, only the process that successfully installs the Descriptor recurses.) If one process responsible for removing a node (and its ancestors) stalls, it could leave $O(\log L)$ marked nodes in the list; this is the source of an *additive* $P \log L$ term in the bound we prove on the number of unnecessary nodes in the list.

We now look at the code in more detail. Each node `X` in the doubly-linked list has `right` and `left` pointers that point toward the list's head and away from it, respectively. `X` also has a `status` field that is initially `unmarked` and `leftDesc` and `rightDesc` fields to hold pointers to Descriptors for splices happening to the left and to the right of `X`, respectively. `X`'s `counter` field is filled in when `X` is appended to the right end of the list with a value that is one greater than the preceding node. To ensure that the height of the priority tree is $O(\log L)$, we use the `counter` value c to define the `priority` of `X` as $p(c)$, where $p(c)$ is either k if c is of the form 2^k, or $2k + 1 -$ (number of consecutive 0's at the right end of the binary representation of c), if $2^k < c < 2^{k+1}$. The resulting priority tree has a sequence of nodes with priorities $1, 2, 3, \ldots$ along the rightmost path in the tree, where the left subtree of the ith node along this rightmost path is a complete binary tree of height $i - 1$, as illustrated in Figure 3. (Trees of this shape have been used to describe search trees [10] and in concurrent data structures [4, 5].) This assignment of priorities ensures that between any two nodes with the same priority, there is another node with lower priority. Moreover, the depth of a node with counter value c is $O(\log L)$. This construction also ensures that `remove` operations are wait-free, since the priority of a node is a bound on the number of recursive calls that a `remove` performs.

A Descriptor of a splice of node `B` out from between `A` and `C` is an object that stores pointers to the three nodes `A`, `B` and `C`. After `B` is marked, we set its Descriptor pointers to a special Descriptor `frozen` to indicate that no further updates should occur on them.

To append a new node `C` after the head node `B`, the `tryAppend(B,C)` operation simply fills in the fields of `C`, and then attempts to swing the `Head` pointer to `C` at line 36. `B`'s `right` pointer is then updated at line 37. If the `tryAppend` stalls before executing line 37, any attempt to append another node after `C` will first help complete the append of `C` (line 32). The boolean value returned by `tryAppend` indicates whether the append was successful.

A `remove(B)` first sets `B`'s `status` to `marked` at line 44. It then stores the `frozen` Descriptor in both `B->leftDesc` and `B->rightDesc`. The first attempt to store `frozen` in one of these fields may fail, but we prove that the second will succeed because of some handshaking, described below. `B` is *frozen* once `frozen` is stored in both of its Descriptor fields. Finally, `remove(B)` calls `removeRec(B)` to attempt the real work of splicing `B`.

The `removeRec(B)` routine manages the recursive splicing of nodes. It first calls `splice`, `spliceUnmarkedLeft` or `spliceUnmarkedRight`, as appropriate, to splice `B`. If the splice of `B` was successful, it then recurses (if needed) on the neighbour of `B` with the larger priority.

The actual updates to pointers are done inside the `splice(A,B,C)` routine, which is called

```
1  class Node {
2    Node *left, *right; // initially null
3    enum status {unmarked,marked,finalized};
4        // initially unmarked
5    int counter; // used to define priority
6    int priority; // defines implicit tree
7    int ts; // timestamp
8    Descriptor *leftDesc, *rightDesc;
9        // initially null
10 };

12 class Descriptor { Node *A, *B, *C; };
13 Descriptor* frozen = new Descriptor();

15 class VersionList {
16   Node* Head;
17   // public member functions:
18   Node* getHead() {return Head;}

20   Node* find(Node* start, int ts) {
21     VNode* cur = start;
22     while(cur != null && cur->ts > ts)
23       cur = cur->left;
24     return cur; }

26   bool tryAppend(Node* B, Node* C) {
27     // B can be null iff C is the initial node
28     if(B != null) {
29       C->counter = B->counter+1;
30       Node* A = B->left;
31       // Help tryAppend(A, B)
32       if(A != null) CAS(&(A->right), null, B);
33     } else C->counter = 2;
34     C->priority = p(C->counter);
35     C->left = B;
36     if(CAS(&Head, B, C)) {
37       if(B != null) CAS(&(B->right), null, C);
38       return true;
39     } else return false; }

41   // public static functions:
42   void remove(Node* B) {
43     // B cannot be null
44     B->status = marked;
45     for F in [leftDesc, rightDesc] {
46       repeat twice {
47         Descriptor* desc = B->F;
48         help(desc);
49         CAS(&(B->F), desc, frozen); } }
50     removeRec(B); }

52   // private helper functions:
53   bool validAndFrozen(Node* D) {
54     // rightDesc is frozen second
55     return D != null && D->rightDesc == frozen; }

57   void help(Descriptor* desc) {
58     if(desc != null && desc != frozen)
59       splice(desc->A, desc->B, desc->C); }

61   int p(int c) {
62     k = floor(log2(c));
63     if(c == 2^k) return k;
64     else return 2k + 1 - lowestSetBit(c); }

65 // private helper functions continued:
66 void removeRec(Node* B) {
67   // B cannot be null
68   Node* A = B->left;
69   Node* C = B->right;
70   if(B->status == finalized) return;
71   int a, b, c;
72   if(A != null) a = A->priority;
73   else a = 0;
74   if(C != null) c = C->priority;
75   else c = 0;
76   b = B->priority;
77   if(a < b > c) {
78     if(splice(A, B, C)) {
79       if(validAndFrozen(A)) {
80         if(validAndFrozen(C) && c > a) removeRec(C);
81         else removeRec(A); }
82       else if(validAndFrozen(C)) {
83         if(validAndFrozen(A) && a > c) removeRec(A);
84         else removeRec(C); } } }
85   else if(a > b > c) {
86     if(spliceUnmarkedLeft(A, B, C) &&
87     validAndFrozen(C)) {
88       removeRec(C); } }
89   else if(a < b < c) {
90     if(spliceUnmarkedRight(A, B, C) &&
91     validAndFrozen(A)) {
92       removeRec(A); } } } }

94 bool splice(Node* A, Node* B, Node* C) {
95   // B cannot be null
96   if(A != null && A->right != B) return false;
97   bool result = CAS(&(B->status), marked, finalized);
98   if(C != null) CAS(&(C->left), B, A);
99   if(A != null) CAS(&(A->right), B, C);
100  return result; }

102 bool spliceUnmarkedLeft(Node* A, Node* B, Node* C) {
103   // A, B cannot be null
104   Descriptor* oldDesc = A->rightDesc;
105   if(A->status != unmarked) return false;
106   help(oldDesc);
107   if(A->right != B) return false;
108   Descriptor* newDesc = new Descriptor(A, B, C);
109   if(CAS(&(A->rightDesc), oldDesc, newDesc)) {
110     // oldDesc != frozen
111     help(newDesc);
112     return true;
113   } else return false; }

115 bool spliceUnmarkedRight(Node* A, Node* B, Node* C) {
116   // B, C cannot be null
117   Descriptor* oldDesc = C->leftDesc;
118   if(C->status != unmarked) return false;
119   help(oldDesc);
120   if(C->left != B || (A != null && A->right != B))
121     return false;
122   Descriptor* newDesc = new Descriptor(A, B, C);
123   if(CAS(&(C->leftDesc), oldDesc, newDesc)) {
124     // oldDesc != frozen
125     help(newDesc);
126     return true;
127   } else return false; } };
```

Figure 4 Linearizable implementation of our doubly-linked list.

after reading `A` in `B->left` and `C` in `B->right`. The routine first tests that `A->right` = `B` at line 96. This could fail for two reasons: `B` has already been spliced out, so there is no need to proceed, or there is a `splice(A,D,B)` that has been partially completed; `B->left` has been updated to `A`, but `A->right` has not yet been updated to `B`. In the latter case, the `remove` that is splicing out `D` will also splice `B` after `D`, so again there is no need to proceed with the splice of `B`. If `A->right` = `B`, `B`'s `status` is updated to `finalized` at line 97, and the pointers in `C` and `A` are updated to splice `B` out of the list at line 98 and 99.

The `spliceUnmarkedLeft(A,B,C)` handles the splicing of a node `B` when `B`'s left neighbour `A` has higher priority but is unmarked, and `B`'s right neighbour `C` has lower priority. The operation attempts to CAS a Descriptor of the splice into `A->rightDesc` at line 109. If there was already an old Descriptor there, it is first helped to complete at line 106. If the new Descriptor is successfully installed, the `help` routine is called at line 111, which in turn calls `splice` to complete the splicing out of `B`. The `spliceUnmarkedLeft` operation can fail in several ways. First, it can observe that `A` has become marked, in which case `A` should be spliced out before `B` since `A` has higher priority. (This test is also a kind of handshaking: once a node is marked, at most one more Descriptor can be installed in it, and this ensures that one of the two attempts to install `frozen` in a node's Descriptor field during the `remove` routine succeeds.) Second, it can observe at line 107 that `A->right` $\neq$ `B`. As described above for the `splice` routine, it is safe to abort the splice in this case. Finally, the CAS at line 109 can fail, either because `A->rightDesc` has been changed to `frozen` (indicating that `A` should be spliced before `B`) or another process has already stored a new Descriptor in `A->rightDesc` (indicating either that `B` has already been spliced or will be by another process).

The `spliceUnmarkedRight` routine is symmetric to `spliceUnmarkedLeft`, aside from a slight difference in line 120 because `splice` changes the `left` pointer before the `right` pointer. The return values of `splice`, `spliceUnmarkedLeft` and `spliceUnmarkedRight` say whether the calling process should continue recursing up the priority tree to splice out more nodes.

5.2 Properties of the Implementation

Detailed proofs of the following results appear in the full version [9]. We sketch them here.

▶ **Theorem 7.** *Under Assumption 6, the implementation in Figure 4 is linearizable.*

Since the implementation is fairly complex, the correctness proof is necessarily quite intricate. We say that `X` $<_c$ `Y` if node `X` is appended to the list before node `Y`. We prove that `left` and `right` pointers in the list always respect this ordering. Removing a node has several key steps: marking it (line 44), freezing it (second iteration of line 49), finalizing it (successful CAS at line 97) and then making it unreachable (successful CAS at line 99). We prove several lemmas showing that these steps take place in an orderly way. We also show that the steps make progress. Finally, we show that the coordination between `remove` operations guarantees that the structure of the list remains a doubly-linked list in which nodes are ordered by $<_c$, except for a temporary situation while a node is being spliced out, during which its left neighbour may still point to it after its right neighbour's pointer has been updated to skip past it. To facilitate the inductive proof of this invariant, it is wrapped up with several others, including an assertion that overlapping calls to `splice` of the form `splice(W,X,Y)` and `splice(X,Y,Z)` never occur. The invariant also asserts that unmarked nodes remain in the doubly-linked list; no `left` or `right` pointer can jump past a node that has not been finalized. Together with Assumption 6.3, this ensures a `find` cannot miss the node that it is supposed to return, regardless of how `find` and `remove` operations are linearized. We linearize `getHead` and `tryAppend` when they access the `Head` pointer.

▶ **Theorem 8.** *The number of steps a* `remove(X)` *operation performs is* $O($`X->priority`$)$ *and the* `remove` *operation is therefore wait-free.*

Proof. Aside from the call to `removeRec(X)`, `remove(X)` performs $O(1)$ steps. Aside from doing at most one recursive call to `removeRec`, a `removeRec` operation performs $O(1)$ steps. Each time `removeRec` is called recursively, the node on which it is called has a smaller priority. Since priorities are non-negative integers, the claim follows. ◀

▶ **Theorem 9.** *The* `tryAppend` *and* `getHead` *operations take* $O(1)$ *steps. The amortized number of steps for* `remove` *is* $O(1)$*.*

Consider an execution with R `remove` operations. Using the argument for Theorem 8, it suffices to bound the number of calls to `removeRec`. There are at most R calls to `removeRec` directly from `remove`. For each of the R nodes `X` that are removed, we show that at most one call to `removeRec(X)` succeeds either in finalizing `X` or installing a Descriptor to remove `X`, and only this `removeRec(X)` can call `removeRec` recursively.

We say a node is *lr-reachable* if it is reachable from the head of the list by following `left` or `right` pointers. A node is *lr-unreachable* if it is not lr-reachable.

▶ **Theorem 10.** *At the end of any execution by* P *processes that contains* L *successful* `tryAppend` *operations and* R `remove` *operations on a set of version lists, and a maximum of* L_{max} *successful* `tryAppend`*s on a single version list, the total number of lr-reachable nodes across all the version lists in the set is at most* $2(L - R) + O(P \log L_{max})$*.*

Theorem 10 considers a set of version lists to indicate that the $O(P \log L_{max})$ additive space overhead is shared across all the version lists in the system. A node `X` is *removable* if `remove(X)` has been invoked. We must show at most $(L - R) + O(P \log L_{max})$ removable nodes are still lr-reachable. We count the number of nodes that are in each of the various phases (freezing, finalizing, making unreachable) of the removal. There are at most P removable nodes that are not yet frozen, since each has a pending `remove` operation on it. There are at most P finalized nodes that are still lr-reachable, since each has a pending `splice` operation on it. To bound the number of nodes that are frozen but not finalized, we classify an unfinalized node as Type 0, 1, or 2, depending on the number of its subtrees that contain an unfinalized node. We show that each frozen, unfinalized node `X` of type 0 or 1 has a pending `remove` or `removeRec` at one of its descendants. So, there are $O(P \log L_{max})$ such nodes. We show that at most half of the unfinalized nodes are of type 2, so there are at most $L - R + O(P \log L_{max})$ type-2 nodes. Summing up yields the bound.

6 Memory Reclamation for Version Lists

We now describe how to safely reclaim the nodes spliced out of version lists and the Descriptor objects that are no longer needed. We apply an implementation of Reference Counting (RC) [2] with amortized expected $O(1)$ time overhead to a slightly modified version of our list. To apply RC in Figure 4, we add a reference count field to each Node or Descriptor and replace raw pointers to Nodes or Descriptors with reference-counted pointers. Reclaiming an object clears all its reference-counted pointers, which may lead to recursive reclamations if any reference count hits zero. This reclamation scheme is simple, but not sufficient by itself because a single pointer to a spliced out node may prevent a long chain of spliced out nodes from being reclaimed (see Figure 5, discussed later). To avoid this, we modify the `splice` routine so that whenever the `left` or `right` pointer of an node `Y` points to a descendant in

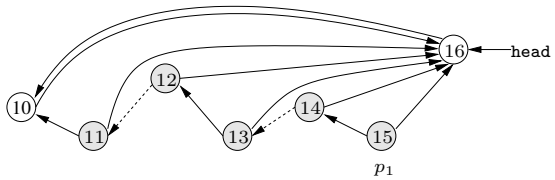

Figure 5 A portion of a version list where shaded nodes 15, 14, ..., 11 have been removed, in that order. Dotted pointers represent left and right pointers set to $\top$ by our modified `splice` routine. Node labels are counter values and vertical positioning represents nodes' priorities (cf. Figure 3).

the implicit tree, we set the pointer to $\top$ after `Y` is spliced out. Thus, only `left` and `right` pointers from spliced out nodes to their ancestors in the implicit tree remain valid. This ensures that there are only $O(\log L)$ spliced out nodes reachable from any spliced out node.

This modification requires some changes to `find`. When a `find` reaches a node whose `left` pointer is $\top$, the traversal moves right instead; this results in following a valid pointer because whenever `splice(A, B, C)` is called, it is guaranteed that either `A` or `C` is an ancestor of `B`. For example in Figure 5, a process p_1, paused on node 15, will next traverse nodes 14, 16, and 10. Breaking up chains of removed nodes (e.g., from node 15 to 11 in Figure 5) by setting some pointers to $\top$ is important because otherwise, such chains can become arbitrarily long and a process paused at the head of a chain can prevent all of its nodes from being reclaimed. In the full version of the paper, we prove that traversing backwards does not have any significant impact on the time complexity of `find`. Intuitively, this is because backwards traversals only happen when the `find` is poised to read a node that has already been spliced out and each backwards traversal brings it closer to a non-removed node.

Using the memory reclamation scheme described above, we prove Theorems 11 and 12 that provide bounds similar to Theorems 9 and 10 in [9]. Both theorems include the resources needed by the RC algorithm, such as incrementing reference counts, maintaining retired lists, etc. Since the RC algorithm uses process-local hash tables, the amortized time bounds in Theorem 9 become amortized *in expectation* in Theorem 11. Using this scheme requires that `getHead` and `find` return reference counted pointers rather than raw pointers. Holding on to these reference counted pointers prevents the nodes that they point to from being reclaimed. For the space bounds in Theorem 12, we consider the number of reference counted pointers K, returned by version list operations that are still used by the application code. In most multiversioning systems (including the one in Appendix 7), each process holds on to a constant number of such pointers, so $K \in O(P)$.

▶ **Theorem 11.** *The amortized expected time complexity of* `tryAppend`*,* `getHead`*,* `remove`*, and creating a new version list is* $O(1)$*. The amortized expected time complexity of* `find(V, ts)` *is* $O(n + \min(d, \log c))$*, where* n *is the number of version nodes with timestamp greater than* `ts` *that are reachable from* `V` *by following* `left` *pointers (measured at the start of the* `find`*),* d *is the depth of the VNode* `V` *in the implicit tree and* c *is the number of successful* `tryAppend` *from the time* `V` *was the list head until the end of the* `find`*. All operations are wait-free.*

▶ **Theorem 12.** *Assuming there are at most* K *reference-counted pointers to VNodes from the application code, at the end of any execution that contains* L *successful* `tryAppend` *operations,* R `remove` *operations and a maximum of* L_{max} *successful* `tryAppend`*s on a single version list, the number of VNodes and Descriptors that have been allocated but not reclaimed is* $O((L - R) + (P^2 + K) \log L_{max})$*.*

In RC, cycles must be broken before a node can be reclaimed. While there are cycles in our version lists, we show that VNodes that have been spliced out are not part of any cycle.

7 Application to Snapshottable Data Structures

We present a summary of the multiversioning scheme of Wei et al. [51], and describe how the techniques in this paper can be applied to achieve good complexity bounds.

The Multiversioning Scheme. Wei et al. [51] apply multiversioning to a concurrent data structure (DS) implemented from CAS objects to make it *snapshottable*. It does so by replacing each CAS object by a VersionedCAS object which stores a version list of all earlier values of the object. VersionedCAS objects support `vRead` and `vCAS` operations, which behave like ordinary read and CAS. They also support a `readVersion` operation which can be used to read earlier values of the object. Wei et al. present an optimization for avoiding the level of indirection introduced by version lists. For simplicity, we apply our MVGC technique to the version without this optimization.

Wei et al. also introduce a *Camera* object which is associated with these *VersionedCAS* objects. The Camera object simply stores a timestamp. A `takeSnapshot` operation applied to the Camera object attempts to increment the timestamp and returns the old value of the timestamp as a snapshot handle. To support read-only query operations on the concurrent DS (such as range-query, successor, filter, etc.), it suffices to obtain a snapshot handle s, and then read the relevant objects in the DS using `readVersion`(s) to get their values at the linearization point of the `takeSnapshot` that returned s. This approach can be used to add arbitrary queries to many standard data structures.

For multiversion garbage collection, Wei et al. [51] uses a variation of EBR [25], inheriting its drawbacks. Applying our range-tracking and version-list data structures significantly reduces space usage, resulting in bounded space without sacrificing time complexity.

Applying Our MVGC Scheme. Operations on snapshottable data structures (obtained by applying the technique in [51]) are divided into *snapshot queries*, which use a snapshot handle to answer queries, and *frontier operations*, which are inherited from the original non-snapshottable DS. We use our doubly-linked list algorithm (with the memory reclamation scheme from Section 6) for each VersionedCAS object's version list, and a range-tracking object `rt` to announce timestamps and keep track of required versions by ongoing snapshot queries. We distinguish between objects inherited from the original DS (DNodes) and version list nodes (VNodes). For example, if the original DS is a search tree, the DNodes would be the nodes of the search tree. See [9] for the enhanced code of [51] with our MVGC scheme.

At the beginning of each snapshot query, the taken snapshot is announced using `rt.announce()`. At the end of the query, `rt.unannounce()` is called to indicate that the snapshot that it reserved is no longer needed. Whenever a `vCAS` operation adds a new VNode `C` to the head of a version list, we `deprecate` the previous head VNode `B` by calling `rt.deprecate(B, B.timestamp, C.timestamp)`. Our announcement scheme prevents VNodes that are part of any ongoing snapshot from being returned by `deprecate`.

Once a VNode is returned by a `deprecate`, it is removed from its version list and the reclamation of this VNode and the Descriptors that it points to is handled automatically by the reference-counting scheme of Section 6. Thus, we turn our attention to DNodes. A DNode can be reclaimed when neither frontier operations nor snapshot queries can access it.

We assume that the original, non-snapshottable DS comes with a memory reclamation scheme, MRS, which we use to determine if a DNode is needed by any frontier operation. We assume that this scheme calls `retire` on a node X when it becomes unreachable from the roots of the DS, and `free` on X when no frontier operations need it any longer. This

assumption is naturally satisfied by many well-known reclamation schemes (e.g., [30, 43, 25]).

Even when MRS `frees` a DNode, it may not be safe to reclaim it, as it may still be needed by ongoing snapshot queries. To solve this problem, we tag each DNode with a birth timestamp and a retire timestamp. A DNode's birth timestamp is set after a DNode is allocated but before it is attached to the data structure. Similarly, a DNode's retire timestamp is set when MRS calls `retire` on it. We say that a DNode is *necessary* if it is not yet freed by MRS, or if there exists an announced timestamp in between its birth and retire timestamp. We track this using the same range-tracking data structure `rt` that was used for VNodes. Whenever MRS `frees` a DNode `N`, we instead call `rt.deprecate(N, N.birthTS, N.retireTS)`. When a DNode gets returned by a `deprecate`, it is no longer needed so we reclaim its storage space.

We say that a VNode is *necessary* if it is pointed to by a DNode that has not yet been deprecated (i.e. freed by MRS) or if its interval contains an announced timestamp. Let D and V be the maximum, over all configurations in the execution, of the number of necessary DNodes and VNodes, respectively. Theorem 13 bounds the overall memory usage of our memory-managed snapshottable data structure. Theorem 14 is an amortized version of the time bounds proven in [51].

▶ **Theorem 13.** *Assuming each VNode and DNode takes $O(1)$ space, the overall space usage of our memory-managed snapshottable data structure is $O(D + V + P^2 \log P + P^2 \log L_{max})$, where L_{max} is the maximum number of successful* `vCAS` *operations on a single* `VCAS` *object.*

▶ **Theorem 14.** *A snapshot query takes amortized expected time proportional to its sequential complexity plus the number of* `vCAS` *instructions concurrent with it. The amortized expected time complexity of frontier operations is the same as in the non-snapshottable DS.*

References

1. D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. StackTrack: An automated transactional approach to concurrent memory reclamation. In *Proc. 9th European Conference on Computer Systems*, pages 25:1–25:14, 2014.
2. D. Anderson, G. E. Blelloch, and Y. Wei. Concurrent deferred reference counting with constant-time overhead. In *Proc. 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 526–541, 2021.
3. S. E. Anderson. Bit twiddling hacks, 2005. Available from `http://graphics.stanford.edu/~seander/bithacks.html`. Retrieved Feb 12, 2021.
4. J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In *Proc. 28th ACM Symposium on Principles of Distributed Computing*, pages 36–45, 2009.
5. H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.
6. H. Attiya and E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems*, 52(4):729–762, 2013.
7. H. G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, Apr. 1978.
8. D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy. KiWi: A key-value map for scalable real-time analytics. *ACM Trans. Parallel Comput.*, 7(3):16:1–16:28, June 2020.
9. N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, Y. Sun, and Y. Wei. Space and time bounded multiversion garbage collection, 2021. Available from https://arxiv.org/abs/2108.02775.
10. J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.*, 5(3):82–86, 1976.
11. P. A. Bernstein and N. Goodman. Multiversion concurrency control–theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.
12. G. E. Blelloch and P. Cheng. On bounding time and space for multiprocessor garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 104–117, 1999.
13. G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures*, pages 89–102, 2020.
14. G. E. Blelloch and Y. Wei. Brief announcement: Concurrent fixed-size allocation and free in constant time. In *Proc. 34th International Symposium on Distributed Computing*, volume 179 of *LIPICS*, pages 51:1–51:3, 2020.
15. G. E. Blelloch and Y. Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In *Proc. 34th International Symposium on Distributed Computing*, volume 179 of *LIPICS*, pages 5:1–5:17, 2020.
16. J. Böttcher, V. Leis, T. Neumann, and A. Kemper. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment*, 13(2):128–141, 2019.
17. T. A. Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.
18. A. Correia, P. Ramalhete, and P. Felber. Orcgc: automatic lock-free memory reclamation. In *Proc. of the 26th ACM Symp. on Principles and Practice of Parallel Programming*, pages 205–218, 2021.
19. D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele. Lock-free reference counting. In *Proc. 20th ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001.
20. A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proc. 30th ACM Symposium on Principles of Distributed Computing*, pages 99–108, 2011.

21 F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010.

22 P. Fatourou and N. D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.

23 P. Fatourou, E. Papavasileiou, and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019.

24 S. M. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–188, 2011.

25 K. Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

26 M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, pages 260–269, 2002.

27 R. Guerraoui, A. Kogan, V. J. Marathe, and I. Zablotchi. Efficient multi-word compare and swap. In *34th International Symposium on Distributed Computing*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

28 T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

29 T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.

30 M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, May 2005.

31 I. Keidar and D. Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *LNCS*, pages 150–165. Springer, 2015.

32 J. Kim, A. Mathew, S. Kashyap, M. K. Ramanathan, and C. Min. MV-RLU: Scaling read-log-update with multi-versioning. In *Proc. 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 779–792, 2019.

33 P. Kumar, S. Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *Proc. Int. Conference on Distributed Computing and Networking*, pages 212–226, 2014.

34 J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, and W.-S. Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *Proc. International Conference on Management of Data*, page 1307–1318, 2016.

35 L. Lu and M. L. Scott. Generic multiversion STM. In *Proc. International Symposium on Distributed Computing*, pages 134–148. Springer, 2013.

36 M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

37 J. Nelson, A. Hassan, and R. Palmieri. Poster: Bundled references: An abstraction for highly-concurrent linearizable range queries. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 448–450, 2021.

38 T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

39 R. Nikolaev and B. Ravindran. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM International Conference on Programming Language Design and Implementation*, pages 987–1002, 2021.

40 C. H. Papadimitriou and P. C. Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems*, 9(1):89–99, 1984.

41 D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. SMV: Selective multi-versioning STM. In *Proc. International Symposium on Distributed Computing*, pages 125–140, 2011.

42 D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 16–25, 2010.

43 P. Ramalhete and A. Correia. Brief announcement: Hazard eras–non-blocking memory reclamation. In *Proc. 29th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 367–369, 2017.

44 D. Reed. Naming and synchronization in a decentralized computer system. Technical Report LCS/TR-205, EECS Dept., MIT, Sept. 1978.

45 N. Shafiei. Non-blocking doubly-linked lists with good amortized complexity. In *Proc. 19th Int. Conference on Principles of Distributed Systems*, volume 46 of *LIPIcs*, pages 35:1–35:17, 2015.

46 J. Shun, Y. Gu, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *Proc. 26th ACM-SIAM Symposium on Discrete Algorithms*, pages 431–448, 2015.

47 A. Singh, T. Brown, and A. Mashtizadeh. NBR: Neutralization based reclamation. In *Proc. 26th ACM Symp. on Principles and Practice of Parallel Programming*, pages 175–190, 2021.

48 H. Sundell. Wait-free reference counting and memory management. In *Proc. 19th IEEE Symposium Parallel and Distributed Processing*, 2005.

49 H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *J. Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

50 J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

51 Y. Wei, N. Ben-David, G. E. Blelloch, P. Fatourou, E. Ruppert, and Y. Sun. Constant-time snapshots with applications to concurrent data structures. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021. A full version is available from `https://arxiv.org/abs/2007.02372`.

52 H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott. Interval-based memory reclamation. In *Proc. 23rd ACM Symp. on Principles and Practice of Parallel Programming*, pages 1–13, 2018.

53 Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. of the VLDB Endowment*, 10(7):781–792, Mar. 2017.

A Model

We consider a standard concurrent system where P *processes* execute asynchronously and may fail by crashing. The system provides a set of *base objects*, which have a state and support atomic operations to read or update their state, such as `read`, `write`, and/or `Compare&Swap` (`CAS`). An *implementation* of a data structure (also known as *concurrent data structure*) provides, for each process p, an algorithm for implementing the operations supported by the data structure (using base objects). More complex objects (or data structures) can be implemented using the base objects.

We follow a standard formalism to model the system. A *configuration* is a vector that represents an instantaneous snapshot of the system (i.e., it provides the state of each process and the value stored in each base object) at some point in time. In an *initial configuration*, all processes are in an initial state and each object stores an initial value. A *step* by a process p is comprised of a single operation applied by p on a base object (and may also contain some local computation performed by p). The sequence of steps that each process performs depends on its program. An *execution* is an alternating sequence of configurations and steps, starting from an initial configuration.

Fix any execution E. The *execution interval* of an operation starts with the step that invokes the operation and finishes with the step performing its last instuction. We use a standard definition of linearizability: E is linearizable if there is a sequential execution σ_E, which contains all complete operations in E (and some of the incomplete ones) and the

following hold: σ_E respects the real-time ordering imposed by the execution intervals of operations in E and each operation in E that appears in σ_E returns the same response in both executions.

An implementation is *wait-free* is each process completes the execution of every operation it invokes within a finite number of steps. It is *lock-free* if in every infinite execution, an infinite number of operations complete; thus, a lock-free implementation ensures that the system as whole makes progress, but individual processes may starve. The *step complexity* of an operation is the maximum number of steps, over all executions, that a thread performs to complete any instance of the operation; we sometimes use the term *time complexity* (or simply *time*) to refer to the step complexity of an operation. Saying that the expected amortized number of steps for an operation is $O(f(n))$ means that the expected total number of steps performed by m invocations of the operation is $O(m \cdot f(n))$.

B Correctness and Analysis of RangeTracker

Here, we provide the proofs for Theorems 3, 4 and 5 from Section 4. These show that RangeTracker is linearizable and give time and space bounds for it.

B.1 Correctness of RangeTracker

In this section, we prove Theorem 3, which says that RangeTracker (Figure 1) is a linearizable implementation of the range-tracking object. Consider any execution E. We first prove that the pools maintained by the algorithm are disjoint and have the right size.

▶ **Invariant 15.**

1. *Among all the local* `LDPool`*s and pools in* `Q`*, no object appears more than once (with one temporary exception: if a process is poised at line 51, its* `LDPool` *may contain elements that also appear in pools of* `Q`*).*
2. *Each* `LDPool` *contains at most B objects (with one temporary exception: if a process has executed line 49, but not yet executed line 51, then its* `LDPool` *may contain up to $2B$ objects).*
3. *Each pool in* `Q` *contains between B and $2B$ objects.*
4. *Each pool is sorted by* `high`*.*

Proof. 1. Assumption 2.4 ensures that each object added to a local `LDPool` is distinct from objects that are in other processes' `LDPool`s or in one of `Q`'s pools. The flush phase of a `deprecate` rearranges the objects of a few pools, and may remove some of the objects to be returned, but does not add or duplicate any objects (with the temporary exception that a copy of `LDPool` is added to `Q` at line 50 before `LDPool` is emptied on the next line).
2. The test at line 38 ensures that `LDPool` is emptied (at line 51) whenever it reaches size B.
3. We show that the modifications to `Q` made by `deprecate` preserve this claim. Since the two pools dequeued at line 39 have at most $2B$ objects each, `Needed` has at most $4B$ objects. If `Needed` has more than $2B$ objects, it is split into two pools whose sizes are between B and $2B$ before those pools are enqueued. `LDPool` is of size at least B when it is enqueued at line 50. If two pools are merged at line 49, then each of the two is of size at most B, so the resulting size is at most $2B$.
4. Assumption 2.1 ensures that elements are appended to `LDPool` in sorted order. All steps that split or merge pools maintain their sorted order. ◀

Recall that we assign linearization points to operations as follows.

- An `announce` is linearized at its `swcopy` on line 28.
- An `unannounce` is linearized at its `write` on line 31.
- A `deprecate` is linearized at line 50 if it executes that line, or at line 37 otherwise.

Operations that do not execute the line specified above are not included in the linearization. We say that an object `o` is *deprecated* at some configuration of an execution if a `deprecate` of object `o` is linearized before that configuration.

▶ **Invariant 16.** *All objects in the following pools are deprecated:*

1. *all pools in* `Q`*,*
2. *the* `Needed` *and* `Redundant` *pools of any* `intersect` *or* `deprecate` *operation, and*
3. *each process p's* `LDPool`*, except for object* `o` *if p is executing* `deprecate(o,low,high)`*.*

Proof. `Q` and each process's `LDPool` are initially empty and no calls to `deprecate` or `intersect` are initially active, so the claim holds trivially in the initial state. We proceed inductively by showing every step s preserves the invariant. Since `Q` is a linearizable queue, we treat the execution of `Q.enq()` and `Q.deq()` as atomic steps.

We first show that if s is a `Q.enq()` performed by a `deprecate` on some object `o`, it preserves claim 1. The pools enqueued at line 44, 45 or 47, contain objects stored in `Needed`, which are deprecated by induction hypothesis 2. The pool enqueued at line 50 results from merging `LDPool` with `Needed`. By the induction hypotheses 2 and 3, all objects in `Needed` and all objects in `LDPool` other than `o` are deprecated. Since s is the linearization point of the `deprecate` on `o`, `o` is also deprecated in the configuration after s. So, claim 1 holds after s.

We next show that if s causes a `deprecate` by process p on object `o` to terminate, it preserves claim 3. By induction hypothesis 3, we must just show that if `o` is in p's `LDPool`, then `o` is deprecated. If `o` is in `LDPool`, then line 51 was not executed, so the `deprecate` of `o` was linearized at line 37 before s. Thus, `o` is deprecated, by the definition of deprecated objects.

We next show that if s is the assignment of the result of `merge` into `LDPool` on line 49 of a `deprecate` on `o`, it preserves claim 3. By induction hypotheses 2 and 3, all objects in `Needed` and all objects in `LDPool` other than `o` are deprecated. Since s merges the objects in `LDPool` with those in `Needed` and assigns the result to `LDPool`, it follows that all nodes in `LDPool` other than `o` are deprecated after s.

Finally, assume that s is an execution of line 24 or line 25 of `intersect`, which appends an object drawn from `MQ` to `Redundant` or `Needed`. We show that s preserves claim 2. Before `intersect` was called at line 41, `MQ` was created on line 39 by merging two pools that were dequeued from `Q`. By induction hypothesis 1, the object s appends to `Redundant` or `Needed` is deprecated. ◀

To show that `deprecate` returns a correct response, we first prove some lemmas.

▶ **Lemma 17.** *If* `intersect(MQ, ar)` *is called with sorted arguments, the* `Redundant` *and* `Needed` *sets it returns partition the elements of* `MQ`*, where* `Needed` *contains exactly those elements whose intervals intersect* `ar`*.*

Proof. We consider the test on line 23. If $\texttt{i} = 0$, then by the exit condition of the while loop, we have $\texttt{r.high} \leq \texttt{ar[0]} \leq \texttt{ar[p]}$ for all `p`, and `r` is placed in `Redundant`. Assume now that $\texttt{i} > 0$. Since `i` was last incremented during an iteration of the for loop on object $\texttt{r}'$, we have $\texttt{ar[i-1]} < \texttt{r}'\texttt{.high} \leq \texttt{r.high}$. Using the exit condition of the while loop, we therefore have $\texttt{ar[i-1]} < \texttt{r.high} \leq \texttt{ar[i]}$. Since `ar` is sorted, `r`'s interval $[\texttt{r.low}, \texttt{r.high})$ intersects `ar` if

and only if `ar[i-1]` $\geq$ `r.low` and this is exactly the test used to determine whether `r` is added to `Needed` or `Redundant`. ◀

We say that an object `o` is *redundant* at some configuration G if it is deprecated at G and its interval does not intersect `Ann`. If a deprecated object is not redundant at configuration G, we say that it is *needed* at G.

The following observation is a direct consequence of Assumption 2.2.

▶ **Observation 18.** *Any non-*$\perp$ *value written into* `Ann` *after a* `deprecate(o,low,high)` *is invoked is greater than or equal to* `high`.

▶ **Lemma 19.** *All objects returned by a* `deprecate` *are redundant in all configurations after the* `deprecate` *is linearized. The size of the set returned by* `deprecate` *is at most* $4P \log P$.

Proof. Consider any instance I of `deprecate`. If I returns the empty set, then the claim holds trivially. So, assume I performs a flush phase and returns a non-empty set `Redundant`. During the flush phase, I dequeues two pools from `Q` and puts their elements in `MQ`. Then, I calls `sortAnnouncements` at line 40 to make a local copy of the announcement array `Ann` in the array `ar`, and calls `intersect` at line 41 to compute `Redundant`.

Let `r` be an element of `Redundant` and `p` be any process. By Invariant 16, all elements of `MQ` are deprecated. So, `r` is deprecated before `sortAnnouncements` is called. By Observation 18, no value written into `Announce[p]` after that will intersect `r`'s interval. By Lemma 17, `ar[p]` is not in `r`'s interval either. Thus at all times after `sortAnnouncements` reads the value `ar[p]` from `Announce[p]`, the value in `Announce[p]` is not in `r`'s interval. Since I is linearized after this read for every `p`, the first claim follows.

All objects returned by the `deprecate` are drawn from two pools dequeued from `Q`. It follows from Invariant 15, that at most $4P \log P$ objects are returned. ◀

The flush phase of a `deprecate` removes all the objects it returns from the pools maintained by RangeTracker. Thus, the sets of nodes returned by any pair of calls to `deprecate` are disjoint. Given the way linearization points are assigned, Invariants 15 and 16, and Lemma 19, then imply the following theorem.

▶ **Theorem 3.** *If Assumption 2 holds, then* RangeTracker *is a linearizable implementation of a range-tracking object.*

B.2 Time and Space Bounds for RangeTracker

In this section, we prove Theorems 4 and 5. In particular, we show that, in any execution E, `deprecate` takes $O(P \log P)$ steps in the worst case, and the amortized number of steps per `deprecate` is constant. We also show that at any time, at most $2H_{max} + 5P^2 \log P$ redundant intervals have not yet been returned by any instance of `deprecate`, where H_{max} is the highest number of objects needed at any point in time during E; this will help us provide space bounds on the overall garbage collection mechanism we construct in this paper.

▶ **Theorem 4.** *In the worst case,* `announce` *and* `unannounce` *take* $O(1)$ *steps, while* `deprecate` *takes* $O(P \log P)$ *steps. The amortized number of steps performed by each operation is* $O(1)$.

Proof. Recall that `swcopy`, `read`, and `write` on destination objects take a constant number of steps. Thus, `announce` and `unannounce` take a constant number of steps in the worst case. Aside from the flush phase, a `deprecate` takes $O(1)$ steps. By Invariant 15, all pools have size $O(P \log P)$. Enqueueing or dequeueing a pool from `Q` takes $O(P \log P)$ steps, including

the time to copy elements in the pool. Thus, as discussed in Section 4.1, the flush phase of a `deprecate` takes $O(P \log P)$ steps. Since each process performs the flush phase once every $P \log P$ calls to `deprecate`, the amortized bound follows. ◀

We next bound the number of deprecated objects that have not yet been returned by any instance of `deprecate`.

▶ **Lemma 20.** *If an object* `o` *is redundant at some configuration, then it remains redundant in every following configuration.*

Proof. Consider any object `o` that is redundant at some configuration G. Since `o` is redundant at G, the elements of `Ann` at G do not intersect `o`'s interval. By Observation 18, no value in `o`'s interval can be written into `Ann` after G. It follows that `o` is redundant in every configuration after G. ◀

▶ **Lemma 21.** *Consider any instance I of* `deprecate` *that executes the flush phase. Let G be the configuration just after I dequeues a list L from* `Q` *(on line 39). Then, I returns all objects in L that were redundant at G.*

Proof. Consider any object `o` in L that is redundant at G. By Lemma 20, `o` is still redundant when I's call to `sortAnnouncements` (line 40) reads each entry from `Announce` and stores the value in the local copy `ar`. Thus each value copied into `ar` cannot be in `o`'s interval. By Lemma 17, `o` is included in the `Redundant` set returned by I. ◀

We denote by H_G the number of needed objects at configuration G. We denote by H_{max} the maximum number of needed objects over all configurations of the execution E, i.e., $H_{max} = \max\{H_G \mid G \in E\}$.

▶ **Lemma 22.** *In the configuration after a dequeue from* `Q`*,* `Q` *contains at most $2H_{max} + 15P^2 \log P$ objects. In the configuration after an enqueue,* `Q` *contains at most $2H_{max} + 20P^2 \log P$ objects.*

Proof. We prove the claim by induction. The claim is clearly true after the first enqueue or dequeue in the execution (since an enqueue can add at most $2P \log P$ objects to the initially empty queue, by Invariant 15). We assume the claim is true for all configurations prior to some operation on `Q`, and prove that it holds in the configuration G after that operation.

First, consider a dequeue from `Q`. If `Q` is empty, then the claim is clearly true. So, suppose the dequeue removes some pool L from `Q`. Let G' be the configuration after L was enqueued.

To bound the number of objects in `Q` at G we first bound the number of objects that are in `Q` at both of the configurations G' and G. Consider any such object `o`. The pool $L_{\texttt{o}}$ containing `o` was dequeued from `Q` by some `deprecate` between G' and G. By Lemma 21, the `deprecate` re-enqueues into `Q` only objects that are needed at the time $L_{\texttt{o}}$ was dequeued. So, `o` was needed at that time. By Lemma 20, this means `o` was needed at the earlier configuration G'. So, there are at most H_{max} such objects `o`.

Next, we count the objects that were dequeued from `Q` before G' and were re-enqueued between G' and G. All of these objects were held in the local memory of a `deprecate` at G'. Since each process holds at most $4P \log P$ dequeued objects at any time, there are at most $4P^2 \log P$ objects in this category.

Finally, we bound the number of *new* objects that are enqueued into `Q` (by line 50) for the first time between G' and G. Let N' be the number of objects in `Q` at G'. By Invariant 15, each pool in `Q` contains at least B objects, so there are at most $\frac{N'}{B}$ pools in `Q` at G'. Thus, the number of dequeues from `Q` between G' and G is at most $\frac{N'}{B}$. Between any pair

of enqueues by a process executing line 50, the process must do two dequeues at line 39. Thus, there are at most $P + \frac{N'}{2B}$ pools enqueued into `Q` between G' and G. Each pool contains exactly B new objects for a total of at most $BP + \frac{N'}{2}$ new objects. By the induction hypothesis, $N' \leq 2H_{max} + 20P^2 \log P$, so the number of new objects enqueued is at most $P^2 \log P + H_{max} + 10P^2 \log P$.

In total, the number of objects in `Q` at G is at most $H_{max} + 4P^2 \log P + (H_{max} + 11P^2 \log P)$, as required.

Next, consider the configuration G after an enqueue into `Q`. Let G' be the configuration after the preceding dequeue (or the initial configuration if there is no such dequeue). By the induction hypothesis, the number of objects in `Q` at G' is at most $2H_{max} + 15P^2 \log P$. Between G and G', each process can execute steps of at most one flush phase, and each flush phase can enqueue pools containing at most $5P \log P$ objects. So, the number of objects in `Q` at G is at most $2H_{max} + 15P^2 \log P + 5P^2 \log P$, as required. ◀

▶ **Theorem 5.** *At any configuration, the number of deprecated objects that have not yet been returned by any instance of* `deprecate` *is at most* $2H + 25P^2 \log P$.

Proof. By Lemma 22, there are at most $2H_{max} + 20P^2 \log P$ deprecated objects in `Q`, and a further $5P \log P$ in the local memory of any process, for a total of $2H_{max} + 25P^2 \log P$. ◀

C Correctness and Analysis of Version List Implementation

In this section, we prove Theorems 7, 9 and 10 from Section 5. These show that our List implementation is linearizable and give time and space bounds for it.

Throughout this section, we fix an execution. Let $\mathtt{V}_1, \mathtt{V}_2, \ldots$ be the nodes assigned to the `Head` pointer (in the order they were stored there), and let $\mathtt{ts}_1, \mathtt{ts}_2, \ldots$ be the timestamps assigned to those nodes. If a node $\mathtt{V}_i$ is never assigned a timestamp, then $\mathtt{ts}_i = \perp$. (The last node in the sequence, if there is one, might not be assigned a timestamp). We assume that the execution satisfies the following statements, which are more formal versions of those stated in Assumption 6.

6.1 For any node `B`, if there are calls to both `tryAppend(B,*)` and `remove(B)` then some call to `tryAppend(B,*)` returns true before the invocation of `remove(B)`.

6.2 If `remove(B)` is invoked before any call to `tryAppend(B,*)` returns true, then there is no `tryAppend` or `find` that is either pending or invoked after the `remove(B)` is invoked.

6.3 If `remove(`$\mathtt{V}_i$`)` is invoked and $\mathtt{ts}_{i+1} \neq \perp$, then there is no `find(*,ts)` with $\mathtt{ts}_i \leq \mathtt{ts} < \mathtt{ts}_{i+1}$ that is either pending or invoked after the `remove(`$\mathtt{V}_i$`)` is invoked.

6.4 Before `tryAppend(B,C)` or `find(B,*)` is invoked, `B` is returned by some `getHead` operation and `B`'s timestamp field has been set. Once a node's timestamp is set, it never changes. If `tryAppend(B,C)` succeeds, the timestamp eventually assigned to `C` is greater than or equal to `B`'s timestamp.

6.5 If `find(`$\mathtt{V}_i$`,t)` is invoked and `tryAppend(`$\mathtt{V}_i$`,`$\mathtt{V}_{i+1}$`)` succeeds, then $\mathtt{ts}_{i+1} > \mathtt{t}$.

6.6 There are no two invocations of `remove(B)` or of `tryAppend(*,C)`.

In our usage in version lists, version nodes are removed only after they have been superseded by a later version (6.1). The exception is the last version node, which is removed when the list is no longer needed, and hence no more calls to `tryAppend` or `find` should be executing after this happens (6.2).

C.1 Correctness of Version Lists

We say a node is *active* if it has been successfully added to the list. More precisely, a node becomes active when the `Head` pointer is changed to point to it at line 36 and it remains active forever after that.

In our proof, we use the notation `X` $\rightarrow$ `Y` to mean `X->right` $=$ `Y` and `X` $\not\rightarrow$ `Y` to mean `X->right` $\neq$ `Y`. Similarly, `X` $\leftarrow$ `Y` means `Y->left` $=$ `X` and `X` $\not\leftarrow$ `Y` means `Y->left` $\neq$ `X`. Finally, `X` $\leftrightarrow$ `Y` means `X` $\leftarrow$ `Y` and `X` $\rightarrow$ `Y`.

We define two orderings on all the nodes that become active during the execution: if `X` and `Y` are pointers to nodes, we write `X` $<_p$ `Y` if `X->priority` $<$ `Y->priority` and `X` $<_c$ `Y` if `X->counter` $<$ `Y->counter`. The `counter` and `priority` fields of a node are initialized in the `tryAppend` routine before the node becomes active at line 36. So, the `counter` and `priority` fields of an active node are well-defined and never change. Hence, the orderings $<_c$ and $<_p$ are fixed. In particular, we have `V`$_1$ $<_c$ `V`$_2$ $<_c \cdots$. For convenience, we also define `X` $>_p$ `null` if `X` is a pointer to any node.

We start with some simple lemmas that follow easily from the code. The following straightforward lemma ensures that, whenever the code reads `X->F` for some field `F` of a node pointer `X`, `X` is not `null`.

▶ **Lemma 23.** *If* `splice`*,* `spliceUnmarkedLeft` *or* `spliceUnmarkedRight` *is called with arguments* `A`*,* `B`*,* `C`*, or if there is a Descriptor (other than* `frozen`*) containing* `A`*,* `B`*,* `C`*, then* `B` *is not* `null` *and there was a previous time when* `A` $\leftarrow$ `B` *and a previous time when* `B` $\rightarrow$ `C`*.*
Every call to `removeRec` *has a non-*`null` *argument.*
Every call to `spliceUnmarkedLeft(A,B,C)` *has a non-*`null` *argument for* `A`*.*
Every call to `spliceUnmarkedRight(A,B,C)` *has a non-*`null` *argument for* `C`*.*

Proof. Assume the claim holds in some prefix of the execution. We prove that it holds after one more step s.

If s creates a Descriptor at line 108 or 122, the claim follows from the precondition of the `spliceUnmarkedLeft` or `spliceUnmarkedRight` that performs s.

If s is an invocation of `spliceUnmarkedLeft(A,B,C)` at line 86 of `removeRec`, then `B` is not `null` by the precondition of `removeRec`. The test on the preceding line ensures `a` is not 0, and hence `A` is not `null`. Furthermore, `A` $\leftarrow$ `B` was previously true at line 68 and `B` $\rightarrow$ `C` was true at line 69.

If s is an invocation of `spliceUnmarkedRight` at line 90 of `removeRec`, the proof is symmetric to the call of `spliceUnmarkedLeft`.

If s is an invocation of `splice` in `removeRec`, then the claim follows from the execution of lines 68–69 that precedes s, and the precondition of `removeRec`.

If s is an invocation of `splice` in the `help` routine, then the claim follows from the induction hypothesis on the Descriptor passed as an argument to `help`, because the test on line 58 ensures that the Descriptor is neither `null` nor `frozen`.

If s is a recursive call to `removeRec` inside `removeRec`, it is preceded by a call to `validAndFrozen` that ensures the node used as the argument is not `null`.

If s is a call to `removeRec` inside `remove`, the claim follows from the precondition that the argument of a `remove` is not `null`. ◀

We next observe that Descriptors are correctly created by `spliceUnmarkedLeft` and `spliceUnmarkedRight` routines.

▶ **Lemma 24.** *If* `X->rightDesc` *is neither* `null` *nor* `frozen`*, then the Descriptor it points to contains* (`X`, `Y`, `Z`) *for some* `Y` *and* `Z` *satisfying* `X` $>_p$ `Y` $>_p$ `Z`*.*

Similarly, if `Z->leftDesc` *is neither* `null` *nor* `frozen`*, then the Descriptor contains* (`X`, `Y`, `Z`) *for some* `X` *and* `Y` *satisfying* $\texttt{X} <_p \texttt{Y} <_p \texttt{Z}$.

Proof. We prove the first claim; the second is symmetric. The Descriptor can only be stored in `X->right` at line 109. It was created at line 108 in a call to `spliceUnmarkedLeft(X,Y,Z)` for some `Y` and `Z`. The Descriptor contains (`X`, `Y`, `Z`) and by the test preceding the call to `spliceUnmarkedLeft(X,Y,Z)` on line 86, we have $\texttt{X} >_p \texttt{Y} >_p \texttt{Z}$. ◀

The next lemma shows that nodes of the list remain in the order they were appended, and that whenever a `left` or `right` pointer is updated, it points further along the list than it did before the change (i.e., nodes are spliced out). It also ensures that there are no cycles of `left` pointers or cycles of `right` pointers and that `left` and `right` pointers are not subject to ABA problems.

▶ **Lemma 25.** *Let* `X` *be any node.*
When `X->left` *is first set to a non-*`null` *value* `W`*,* `W` *is an active node and* $\texttt{W} <_c \texttt{X}$.
Whenever the value of `X->left` *changes from a value* `W` *to a non-*`null` *value* `U`*,* `U` *is an active node and* $\texttt{U} <_c \texttt{W} <_c \texttt{X}$.
Similarly, When `X->right` *is first set to a non-*`null` *value* `Y`*,* `Y` *is an active node and* $\texttt{X} <_c \texttt{Y}$.
Whenever the value of `X->right` *changes from a non-*`null` *value* `Y` *to* `Z`*,* `Z` *is an active node and* $\texttt{X} <_c \texttt{Y} <_c \texttt{Z}$.

Proof. A CAS on `X->left` at line 98 uses a non-`null` expected value, by Lemma 23. So, `X->left` is first set to a non-`null` value `W` at line 35 of tryAppend. By Assumption 6.4, `W` is active. Moreover, `X->counter` = `W->counter` + 1.

`X->left` is never changed by this line again because `tryAppend(*,X)` is called at most once, by Assumption 6.6. Consider an update of `X->left` from value `W` to a non-`null` value `U`, which can only happen at line 98 of a `splice(U,W,X)`. Assume the claim holds in the prefix of the execution prior to this update. By Lemma 23, `W` is not `null` and there was an earlier time when $\texttt{U} \leftarrow \texttt{W}$. By the induction hypothesis, if `U` is not `null`, then `U` is active and $\texttt{U} <_c \texttt{W} <_c \texttt{X}$.

A CAS on `X->right` at line 99 uses a non-`null` expected value, by Lemma 23. So, `X->right` is first set to a non-`null` value `Y` at line 32 or 37 of `tryAppend`. Prior to this step, we have $\texttt{X} \leftarrow \texttt{Y}$ at line 30 or 35, respectively. As proved above, this means that $\texttt{X} <_c \texttt{Y}$. By assumption 6.4, the new value `Y` used by the CAS at line 32 is an active node. The new value `Y` used by the CAS at line 37 is an active node, since it was successfully stored in `Head` at line 36.

Consider an update to `X->right` from a non-`null` value `Y` to a non-`null` value `Z`, which can only happen at line 99 of `splice(X,Y,Z)`. Assume the claim holds in the prefix of the execution prior to this update. There was an earlier time when $\texttt{Y} \rightarrow \texttt{Z}$, by Lemma 23. Thus, by the induction hypothesis, `Z` is active and $\texttt{X} <_c \texttt{Y} <_c \texttt{Z}$. ◀

The following lemma and its corollary are used to ensure that a node is spliced out of the list only if a remove operation has been called on it.

▶ **Lemma 26.** *Before* `removeRec(Y)`*,* `splice(X,Y,Z)`*,* `spliceUnmarkedLeft(X,Y,Z)` *or* `spliceUnmarkedRight(X,Y,Z)` *is called,* `Y->status` *is set to* `marked`*.*

Proof. By Lemma 23, `Y` is not `null`. `Y->status` is initially `unmarked`. The only way that it can change from `unmarked` to another value is by setting it to `marked` at line 44 so the first execution of that line on the node will succeed.

If `removeRec(Y)` is called at line 50 of `remove(Y)`, `Y->status` was previously set to `marked` at line 44. If `removeRec(Y)` is called recursively inside the `removeRec` routine, the test `validAndFrozen(Y)` must first have returned true. So, at some earlier time, `Y->rightDesc` was set to `frozen` at line 49, which can only happen after `Y->status` is set to `marked` at line 44.

The routines `spliceUnmarkedLeft(X,Y,Z)` and `spliceUnmarkedRight(X,Y,Z)` can only be called by `removeRec(Y)`, and it has already been shown that `Y->status` is set to `marked` before `removeRec(Y)` is called. The same applies if `splice(X,Y,Z)` is called in `removeRec(Y)`.

If `splice(X,Y,Z)` is called by `help`, then `help` was called on a Descriptor containing (`X`, `Y`, `Z`). This Descriptor was created either by a `spliceUnmarkedLeft(X,Y,Z)` or `spliceUnmarkedRight(X,Y,Z)`. So, as shown above, `Y->status` was already set to `marked`. ◀

▶ **Corollary 27.** *If a node* `Y` *is finalized, some process has called* `remove(Y)`*.*

Proof. `Y` can only be finalized by a call to `splice(*,Y,*)`. By Lemma 26 `Y->status` is set to `marked` before the `splice` is called, and this can only be done at line 44 of a `remove(Y)`. ◀

Next, we show that the first CAS step that attempts to finalize a node succeeds, and permanently changes the node's status to `finalized`.

▶ **Lemma 28.** *At all times after line 97 is performed on a node* `X`*,* `X->status` *is* `finalized`*.*

Proof. Consider the first time the CAS on line 97 is performed on `X`. By Lemma 26, `X->status` was set to `marked` prior to this CAS. There is no way for `X->status` to change to any other value, except `finalized`. So the CAS will succeed and change `X->status` to `finalized`. After this occurs, no execution of line 44 can change the value of `X->status`, since `remove(X)` is called at most once by assumption 6.6, and no other step can change the value of `X->status`. ◀

The next two lemmas describe freezing the fields of a node that can store Descriptors. After a `remove(X)` marks `X`, it performs two attempts to change each of its Descriptor fields to `frozen`. Although the first attempt may fail, we prove that no other Descriptor can be written into these fields after that first attempt. This, in turn, guarantees that the second attempt will succeed, and the Descriptor fields remain `frozen` forever after the second attempt. After this, it is safe to go on to try to splice `X` out of the list.

▶ **Lemma 29.** *Let* `F` $\in$ {`rightDesc`, `leftDesc`}*. No non-*`frozen` *Descriptor can be stored in* `X->F` *after a* `remove(X)` *performs line 49 on* `X->F`*.*

Proof. We prove the claim for `X->rightDesc`. The proof for `X->leftDesc` is symmetric. Let cas_1 be the first execution of line 49 on `X->rightDesc` and let r_1 be the preceding read of `X->rightDesc` at line 47. To derive a contradiction, assume there is some step that stores a non-`frozen` value in `X->rightDesc` after cas_1. Let cas_2 be the first such step. Then, cas_2 is an execution of line 109 in a call to `spliceUnmarkedLeft(X,Y,Z)` for some `Y` and `Z`. Let r_2 be the preceding read of `X->rightDesc` at line 104.

Prior to r_1, `X->status` is set to `marked` at line 44, and `X->status` remains `marked` or `finalized` forever after. So, r_2 must occur before r_1, since the line after r_2 observes that `X->status` is unmarked. Since r_2 is before cas_1, which is the first attempt to CAS `frozen` into `X->rightDesc`, it must read a value D different from `frozen`. Since cas_2 succeeds, `X->rightDesc` still has the value D when cas_2 occurs. Since at most one CAS attempts to store D in `X->rightDesc`, this means `X->rightDesc` has the value D at all times between

r_2 and cas_2. But r_1 and cas_1 are within this interval of time, so r_1 would read D and cas_1 would succeed in changing the value from D to `frozen`. This contradicts the fact that `X->rightDesc` does not change between r_2 and cas_2. ◀

▶ **Lemma 30.** *If* `removeRec(X)` *has been called, then both* `X->rightDesc` *and* `X->leftDesc` *are* `frozen`.

Proof. Let `F` $\in$ {`leftDesc`, `rightDesc`}. Lemma 29 says that no non-`frozen` Descriptor can be stored in `X->F` after line 49 has been performed on `X->F`. Thus, the second execution of line 49 on `X->F` stores `frozen` in `X->F`, and `X->F` can never change thereafter. So, if `removeRec(X)` is called at line 50, `frozen` has already been stored in `X->F`.

If `removeRec(X)` is called recursively from the `removeRec` routine, then there is a preceding test that ensures `X->rightDesc` is `frozen`. Since `frozen` can be written to `X->rightDesc` only after two executions of line 49 on `X->leftDesc`, it follows that `X->leftDesc` has also been frozen. Thus, `frozen` has already been stored in `X->F` before the call to `removeRec(X)`.

By Lemma 29, once `frozen` is stored in `X->F` (at line 49), `X->F` can never be changed again. ◀

Since the `Head` pointer can only change at line 36, we have the following observation. It implies that there is no ABA problem on the `Head` pointer.

▶ **Observation 31.** *If* `Head` *changes from node* `X` *to node* `Y`, `Y->counter` = `X->counter` + 1.

A *waning configuration* C is one where, for some node `X`, a `remove(X)` was called before C and before any `tryAppend(X,*)` had returned true. A configuration is called a *waxing configuration* if it is not a waning configuration. Once the execution switches from waxing to waning configurations, there can no longer be any `tryAppend` operations, by precondition 6.2. Thus, the list can grow while the execution is in waxing configurations, but can only shrink once it enters a waning configuration.

▶ **Lemma 32.** *In any waxing configuration, the node that* `Head` *points to is not finalized.*

Proof. Suppose `Head` points to `X` in some waxing configuration C. By Observation 31, no `tryAppend(X,*)` has performed a successful CAS at line 36. Thus, no `tryAppend(X,*)` has returned true before C. Since C is a waxing configuration, this means that no `remove(X)` has been invoked before C. By Corollary 27, `X` is not finalized in C. ◀

The following invariant gives a fairly complete description of the state of the doubly-linked list at any time. All of the unfinalized nodes are in the list. There can be at most one finalized node between any two unfinalized nodes (or to the left of the first unfinalized node or to the right of the last unfinalized node) if there is an ongoing splice of that finalized node. Moreover, the `left` and `right` pointers of any nodes that have already been spliced out of the list cannot skip over any unfinalized node. These properties are useful in showing that any traversal of the list cannot skip any unfinalized nodes and hence must visit every node that has not yet been removed.

Intuitively, part 7 of the invariant shows that we cannot concurrently splice out two adjacent nodes, which is important for avoiding the problem illustrated in Figure 2. We say that `(X,Y,Z)` is a *splice triple* in an execution if there is a call to `splice(X,Y,Z)` or if a Descriptor (other than `frozen`) that contains `(X,Y,Z)` is stored in some node's `leftDesc` or `rightDesc` field.

▶ **Invariant 33.** *In any configuration C, the following statements hold. Let* $\mathtt{U}_1, \mathtt{U}_2, \ldots, \mathtt{U}_k$ *be the sequence of active, non-finalized nodes in C, ordered by* $<_c$.

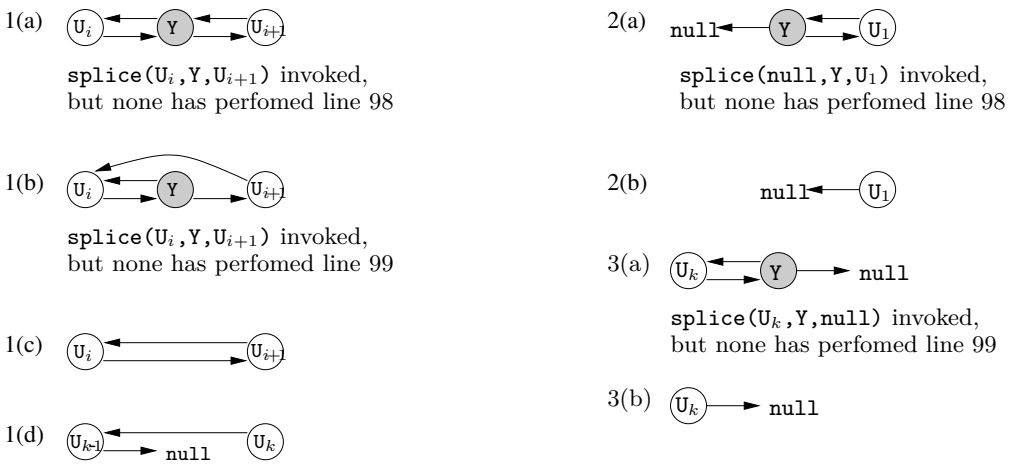

Figure 6 Possible configurations for list nodes, as described in Invariant 33, claims 1–3. Finalized nodes are grey.

1. *For $1 \leq i < k$, one of the following statements is true (refer to Figure 6).*
   - **(a)** *There is a finalized node* Y *such that* $\mathtt{U}_i \leftrightarrow \mathtt{Y} \leftrightarrow \mathtt{U}_{i+1}$ *and a* splice($\mathtt{U}_i$,Y,$\mathtt{U}_{i+1}$) *has been invoked, but no* splice($\mathtt{U}_i$,Y,$\mathtt{U}_{i+1}$) *has performed line 98.*
   - **(b)** *There is a finalized node* Y *such that* $\mathtt{U}_i \leftrightarrow \mathtt{Y} \rightarrow \mathtt{U}_{i+1}$ *and* $\mathtt{U}_i \leftarrow \mathtt{U}_{i+1}$ *and a* splice($\mathtt{U}_i$,Y,$\mathtt{U}_{i+1}$) *has been invoked, but no* splice($\mathtt{U}_i$,Y,$\mathtt{U}_{i+1}$) *has performed line 99.*
   - **(c)** $\mathtt{U}_i \leftrightarrow \mathtt{U}_{i+1}$.
   - **(d)** $i = k - 1$ *and* $\mathtt{U}_{k-1} \rightarrow$ null *and* $\mathtt{U}_{k-1} \leftarrow \mathtt{U}_k$ *and a* tryAppend($\mathtt{U}_{k-1}$,$\mathtt{U}_k$) *has performed line 36 successfully, but has not performed line 37 and no* tryAppend($\mathtt{U}_k$,*) *has performed line 32.*
2. *If $k \geq 1$ then either*
   - **(a)** *There is a finalized node* Y *such that* null $\leftarrow$ Y $\leftrightarrow$ $\mathtt{U}_1$ *and a* splice(null,Y,$\mathtt{U}_1$) *has been invoked, but no* splice(null,Y,$\mathtt{U}_1$) *has performed line 98, or*
   - **(b)** null $\leftarrow$ $\mathtt{U}_1$.
3. *If $k \geq 1$ then either*
   - **(a)** *There is a finalized node* Y *such that* $\mathtt{U}_k \leftrightarrow \mathtt{Y} \rightarrow$ null *and a* splice($\mathtt{U}_k$,Y,null) *has been invoked, but no* splice($\mathtt{U}_k$,Y,null) *has performed line 99, or*
   - **(b)** $\mathtt{U}_k \rightarrow$ null.
4. *If, before this configuration, either a* splice(X,Y,Z) *has been invoked or a Descriptor containing* (X,Y,Z) *has been stored in a node, then* $\mathtt{X} \leftarrow \mathtt{Y} \rightarrow \mathtt{Z}$.
5. **(a)** *Suppose* X *is finalized and* $\mathtt{X} \rightarrow \mathtt{Z}$. *If* $\mathtt{Z} \neq$ null, *then all nodes* Y *such that* $\mathtt{X} <_c \mathtt{Y} <_c \mathtt{Z}$ *are finalized. If* Z *is* null, *then all nodes* Y *such that* $\mathtt{X} <_c \mathtt{Y}$ *are finalized.*
   - **(b)** *Suppose* Z *is finalized and* $\mathtt{X} \leftarrow \mathtt{Z}$. *If* $\mathtt{X} \neq$ null, *then all nodes* Y *such that* $\mathtt{X} <_c \mathtt{Y} <_c \mathtt{Z}$ *are finalized. If* X *is* null, *then all nodes* Y *such that* $\mathtt{Y} <_c \mathtt{Z}$ *are finalized.*
6. *For any Descriptor* (X,Y,Z) *that was in the* rightDesc *or* leftDesc *field of a node in a previous configuration but is no longer in that field,*

(a) *if* `X` $\neq$ `null` *then* `X->right` $>_c$ `Y`*, and*
(b) *if* `Z` $\neq$ `null` *then* `Z->left` $<_c$ `Y`*.*

7. *There are no two splice triples of the form* `(W,X,Y)` *and* `(X,Y,Z)` *in the prefix of the execution up to* C*.*
8. *If a* `splice(X,Y,Z)` *has been called and* `Y` *is not finalized, then neither* `X` *nor* `Z` *are finalized nodes.*

Proof. First, we show that claim 8 follows from the others. Suppose claims 1, 2, 3, 4 and 7 hold for some configuration in which `Y` is not finalized and a `splice(X,Y,Z)` has been called. By claim 4, `X` $\leftarrow$ `Y` $\rightarrow$ `Z`. To derive a contradiction, suppose `X` is a finalized node. The proof for `Z` is symmetric. If `Y` is $\texttt{U}_1$ then by claim 2, a `splice(null,X,Y)` has been invoked since `null` $\not\leftarrow$ `Y`. Otherwise, `Y` is $\texttt{U}_{i+1}$ for some $i \geq 1$. Since `X` $\leftarrow$ `Y`, the non-finalized nodes $\texttt{U}_i$ and $\texttt{U}_{i+1}$ satisfy case 1a of claim 1, and a `splice(`$\texttt{U}_i$`,X,Y)` has been invoked. Either way, we have an invocation of `splice(X,Y,Z)` and `splice(*,X,Y)` contradicting claim 7.

It remains to show that for any step s if all the claims are satisfied in all configurations before s, then claims 1 to 7 are satisfied in the configuration after s. The only steps that could affect the truth of claims 1–7 are successful CAS steps, an invocation of `splice`, executions of line 32, 37, 98 or 99, that do not perform a successful CAS, and executions of line 44. So, we shall consider each of these types of steps in turn.

Case 1 Suppose s is a successful CAS at line 98 by a call to `splice(A,B,C)`. This step changes `C->left` from `B` to `A`.

To derive a contradiction, suppose `C` is finalized in the configuration before s. Consider the `splice(*,C,*)` operation that finalized `C`. By induction hypothesis 4, since `B` $\leftarrow$ `C`, it must have been a `splice(B,C,*)` operation. This contradicts induction hypothesis 7. Thus, `C` is not finalized when s occurs.

By Lemma 28, `B` is finalized when s occurs. In the configuration before s, we have `A` $\leftarrow$ `B` $\rightarrow$ `C` by induction hypothesis 4 and `B` $\leftarrow$ `C` since the CAS succeeds.

If `A` is `null`, then by the induction hypothesis, `C` must be the leftmost non-finalized node and case 2a of induction hypothesis 2 is satisfied. Thus, once s updates `C->left` to `null`, case 2b of claim 2 is satisfied.

If `A` is not `null`, then by the induction hypothesis, `A` is not finalized and the nodes `A`, `B`, `C` satisfy case 1a of induction hypothesis 1, so `A` $\rightarrow$ `B` in the configuration prior to s. Thus, once s updates `C->left` to `A`, case 1b of claim 1 will be satisfied for these three nodes.

Thus, claims 1, 2 and 3 are preserved by s.

Next, we verify that s preserves claim 4. Suppose `splice(X,Y,Z)` has been called before s. The only way that s can cause claim 4 to become false is if it changes `Y->left` from `X` to another value. If s did this, it would have to be part of a `splice(*,X,Y)` operation, but this would contradict induction hypothesis 7.

Next, we verify that s preserves claim 5. We consider two cases.

Suppose `A` is `null`. Since claim 5b holds before s, it suffices to check that there are no finalized nodes less than `B` in the $<_c$ order. This follows from induction hypothesis 5b, because when s occurs, `B` is finalized and `null` $\leftarrow$ `B`.

Suppose `A` is not `null`. Since claim 5b holds before s, it suffices to check that there are no finalized nodes between `A` and `B` in the $<_c$ order. This follows from induction hypothesis 5b, because when s occurs, `B` is finalized and `A` $\leftarrow$ `B`.

It follows from Lemma 25 that step s preserves claim 6.

Case 2 Suppose s is a successful CAS at line 99 by a call to `splice(A,B,C)`. This step changes `A->right` from `B` to `C`. (This case is nearly symmetric to 1.)

To derive a contradiction, suppose `A` is finalized in the configuration before s. Consider the `splice(*,A,*)` operation that finalized `A`. By induction hypothesis 4, since $\texttt{A} \rightarrow \texttt{B}$, it must have been a `splice(*,A,B)` operation. This contradicts induction hypothesis 7. Thus, `A` is not finalized when s occurs.

By Lemma 28, `B` is finalized when s occurs. In the configuration before s, we have $\texttt{A} \leftarrow \texttt{B} \rightarrow \texttt{C}$ by induction hypothesis 4 and $\texttt{A} \rightarrow \texttt{B}$ since the CAS succeeds.

If `C` is `null`, then by the induction hypothesis, `A` must be the rightmost non-finalized node and case 3a of induction hypothesis 3 is satisfied. Thus, once s updates `A->right` to `null`, case 3b of claim 3 is satisfied.

If `C` is not `null`, then by the induction hypothesis, `C` is not finalized and the nodes `A`, `B`, `C` satisfy case 1b of induction hypothesis 1 in the configuration before s. (Case 1a cannot be satisfied, since the `splice(A,B,C)` that performs s has already completed line 98.) So $\texttt{A} \leftarrow \texttt{C}$ in the configuration prior to s. Thus, once s updates `C->left` to `A`, case 1c of claim 1 will be satisfied with $\texttt{A} \leftrightarrow \texttt{C}$.

Thus, claims 1, 2 and 3 are preserved by s.

Next, we verify that s preserves claim 4. Suppose `splice(X,Y,Z)` has been called before s. The only way that s can cause claim 4 to become false is if it changes `Y->right` from `Z` to another value. If s did this, it would have to be part of a `splice(Y,Z,*)` operation, but this would contradict induction hypothesis 7.

Finally, we verify that s preserves claim 5. We consider two cases.

Suppose `C` is `null`. Since claim 5a holds before s, it suffices to check that there are no finalized nodes greater than `B` in the $<_c$ order. This follows from induction hypothesis 5a, because when s occurs, `B` is finalized and $\texttt{B} \rightarrow \texttt{null}$.

Suppose `C` is not `null`. Since claim 5a holds before s, it suffices to check that there are no finalized nodes between `B` and `C` in the $<_c$ order. This follows from induction hypothesis 5a, because when s occurs, `B` is finalized and $\texttt{B} \rightarrow \texttt{C}$.

It follows from Lemma 25 that step s preserves claim 6.

Case 3 Suppose s is a successful CAS at line 97 of `splice(A,B,C)`. Then, s finalizes `B`.

In the configuration before s, $\texttt{A} \leftarrow \texttt{B} \rightarrow \texttt{C}$ by induction hypothesis 4 and neither `A` nor `C` is a finalized node by induction hypothesis 8. Moreover, since the CAS s succeeds, it follows from Lemma 28 that no `splice(A,B,C)` has performed line 97 before, and hence no `splice(A,B,C)` has performed line 98 or 99.

If `C` is not `null`, then $\texttt{B} \leftarrow \texttt{C}$ by induction hypothesis 1 applied to the unfinalized nodes `B` and `C`.

If `A` is not `null`, we apply induction hypothesis 1 to the unfinalized nodes `A` and `B`. Since $\texttt{A} \leftarrow \texttt{B}$, they either satisfy case 1b or 1c. Suppose they satisfy case 1b. Then there is a finalized node `D` such that $\texttt{A} \rightarrow \texttt{D} \rightarrow \texttt{B}$. By Lemma 25, $\texttt{D} <_c \texttt{B}$. Also by Lemma 25, $\texttt{D} \geq_c \texttt{B}$, since $\texttt{A} \rightarrow \texttt{B}$ at the earlier step when line 96 is executed. This contradiction implies that if `A` is non-`null`, then `A` and `B` satisfy case 1c of claim 1, so $\texttt{A} \leftrightarrow \texttt{B}$ in the configuration before s.

Suppose both `A` and `C` are `null`. Since $\texttt{null} \leftarrow \texttt{B}$, `B` is the smallest active, non-finalized node with respect to $<_c$. Since $\texttt{B} \rightarrow \texttt{null}$, `B` satisfies either induction hypothesis 1d (with $k - 1 = 1$) or 3b (with $k = 1$) in the configuration before s. We next show that it cannot, in fact, be 1d. Since s finalizes `B`, a `remove(B)` has been invoked before s. If claim 1d were satisfied, then there would also be a pending `tryAppend(B,*)` that performed line 36 successfully, so no other `tryAppend(B,*)` could have returned true.

This contradicts assumption 6.1. Thus, B is the only active unfinalized node before s. So, after s finalizes B, there are no active, non-finalized nodes, so claims 1, 2 and 3 are trivially satisfied.

If A is null but C is not null, then null $\leftarrow$ B $\leftrightarrow$ C and B becomes finalized by s, leaving C as the leftmost active, non-finalized node. Thus, case 2a of claim 2 is satisfied after s.

Symmetrically, if C is null but A is not null, then A $\leftrightarrow$ B $\rightarrow$ null and B becomes finalized by s, leaving A as the rightmost active, non-finalized node. Thus, case 3a of claim 3 is satisfied after s.

If neither A nor C is null, A $\leftrightarrow$ B $\leftrightarrow$ C when s occurs. Thus, when s finalizes B, A and C become consecutive, non-finalized nodes that satisfy case 1a of claim 1.

In all cases, claims 1, 2 and 3 are preserved.

Step s trivially preserves claim 4 since it does not change any left or right pointers.

Finally, we verify that s preserves claim 5. As argued above, B's left and right pointers point to the next unfinalized node on either side of B in the order $<_c$, or, if they are null pointers, there are no such nodes. So, the claim is satisfied.

Case 4 Suppose s is a successful CAS at line 32 in tryAppend(B,C). This CAS modifies A->right from null to B. By Assumption 6.2, the configuration before s cannot be a waning configuration, since no tryAppend can be pending in a waning configuration. By Assumption 6.4, Head was equal to B prior to s. Since A was read from B->left at line 30, we have A $<_c$ B, by Lemma 25. Thus, A is not the last active node with respect to $<_c$. By Observation 31, A $<_c$ Head when s occurs. To derive a contradiction, suppose A is finalized when s occurs. Since A $\rightarrow$ null in the configuration before s, induction hypothesis 5.5a implies that there Head is finalized, contradicting Lemma 32. So, A is unfinalized when s occurs.

Since A $\rightarrow$ null in the configuration before s and is not the last active, unfinalized node (with respect to $<_c$), the induction hypothesis implies that A must be U_{k-1}, satisfying case 1d of induction hypothesis 1. Thus, some tryAppend(A,U_k) has performed line 36, but no tryAppend(U_k,*) has made another node greater then U_k active. Since B is an active node with B $>_c$ A, B must be U_k. Thus, after s updates A->right to B, we have A $\leftrightarrow$ B, or $U_{k-1} \leftrightarrow U_k$, satisfying claim 1c.

This step does not affect the other claims of the invariant.

Case 5 Suppose s is a successful CAS at line 37 in tryAppend(B,C). Then, s changes B->right from null to C. Prior to s, Head was changed from B to C at line 36, so B and Care active at s.

By Observation 31, there is no ABA problem on the Head pointer. So the tryAppend that performs s is the only tryAppend(B,*) that could return true. By Assumption 6.1, no remove(B) has been invoked before s. By Lemma 27, B is not finalized when s occurs.

Since Head $>_c$ B when s occurs and Head is not finalized, by Lemma 32, B is not the last active, unfinalized node with respect to $<_c$. In the configuration before s, B $\rightarrow$ null, so the induction hypothesis implies that B $= U_{k-1}$ and case 1d of induction hypothesis 1 is satisfied. Thus, some tryAppend(B,U_k) has performed line 36, but no tryAppend(U_k,*) has made another node greater then U_k active. Since C is an active node with C $>_c$ B, we must have C $= U_k$. Thus, after s updates B->right to C, we have B $\leftrightarrow$ C, or $U_{k-1} \leftrightarrow U_k$, satisfying claim 1c.

This step does not affect the other claims of the invariant.

Case 6 Suppose s is an invocation of splice(X,Y,Z) or a CAS at line 109 or 123 that

successfully stores a Descriptor containing `(X,Y,Z)`. This step does not affect the truth of claim 1, 2, 3 or 5.

We first prove that s preserves claim 7.

If s is an invocation of `splice` at line 59 of `help(desc)`, it gets its arguments from the Descriptor `desc`. Prior to the call to `help(desc)` at line 48, 119, 125, 106 or 125, `desc` was in the `rightDesc` or `leftDesc` field of a node at line 47, 104, 109, 117 or 123, respectively. Thus, Claim 7 follows from induction hypothesis 7.

So it remains to show that splice triples that arise from calls to splice on line 78 and from Descriptors satisfy the claim. To derive a contradiction, assume there are two such overlapping triples `(W,X,Y)` and `(X,Y,Z)`. We consider all possible cases and show that each leads to a contradiction.

Case i Suppose both `splice(W,X,Y)` and `splice(X,Y,Z)` have been called from line 78. Before `splice(W,X,Y)`, the test on line 77 ensures that $\texttt{X} >_p \texttt{Y}$. Similarly if `splice(X,Y,Z)` is called, then $\texttt{X} <_p \texttt{Y}$. A contradiction.

Case ii Suppose there are two Descriptors `descW` containing `(W,X,Y)` and `descX` containing `(X,Y,Z)` that have each been stored in a node. We consider the case where $\texttt{X} >_p \texttt{Y}$. The case where $\texttt{X} <_p \texttt{Y}$ is symmetric (and `X` and `Y` cannot have the same priority, by Lemma 24). By Lemma 24, `descW` and `descX` were stored in the `rightDesc` fields of `W` and `X` by calls to `spliceUnmarkedLeft(W,X,Y)` and `spliceUnmarkedLeft(X,Y,Z)`, respectively. Let these calls be $sul_{\texttt{X}}$ and $sul_{\texttt{Y}}$, and let $rr_{\texttt{X}}$ be the call to `removeRec(X)` that invokes $sul_{\texttt{X}}$. By Lemma 30, `descX` is removed from `X->rightDesc` at some point before $rr_{\texttt{X}}$ performs line 69. By induction hypothesis 6, this means that, at all times during $rr_{\texttt{X}}$, $\texttt{X} \not\rightarrow \texttt{Y}$. This contradicts the fact that $rr_{\texttt{X}}$ read `Y` from `X->right` at line 69.

Case iii Suppose `splice(W,X,Y)` has been called from line 78 and a Descriptor `desc` containing `(X,Y,Z)` has been stored in a node. Let rr be the `removeRec(X)` operation that calls `splice(W,X,Y)` at line 78. By the test at line 77, $\texttt{X} >_p \texttt{Y}$. By Lemma 24, `desc` must have been stored in `X->rightDesc`. By Lemma 30, `desc` is removed from `X->rightDesc` before rr is called. By induction hypothesis 6, this means that, at all times during rr, $\texttt{X} \not\rightarrow \texttt{Y}$. This contradicts the fact that rr read `Y` from `X->right` at line 69.

Case iv Suppose a Descriptor `desc` containing `(W,X,Y)` has been stored in a node and `splice(X,Y,Z)` is called from line 78. The proof is symmetric to Case 6iii.

This completes the proof of Claim 7.

Next, we verify that $\texttt{X} \leftarrow \texttt{Y} \rightarrow \texttt{Z}$ to ensure claim 4 is satisfied. We show that $\texttt{X} \leftarrow \texttt{Y}$ in the configuration after s; the proof that $\texttt{Y} \rightarrow \texttt{Z}$ is symmetric. By Lemma 23, there is a time before s when $\texttt{X} \leftarrow \texttt{Y}$. To derive a contradiction, suppose there is some step s' that changed `Y->left` from `X` to some other value `W` before s. Step s' must be an execution of line 98 in a `splice(W,X,Y)` operation, which contradicts Claim 7, proved above.

If s is a CAS at line 109 or 123, we verify that it preserves claim 6 in Case 10, below.

Case 7 Suppose s is a failed CAS on line 98 inside a `splice(A,B,C)`. This step is a failed attempt to change `C->left` from `B` to `A`. If this is not the first time line 98 is performed by a `splice(A,B,C)`, then there is nothing to check. If it is the first, then

we must verify that A, B and C do not satisfy case 1a of claim 1 or case 2a of claim 2. But this is obvious, since if they did, the CAS would succeed.

Case 8 Suppose s is a failed CAS on line 99 inside a `splice(A,B,C)`. This step is a failed attempt to change `A->right` from `B` to `C`. If this is not the first time line 99 is performed by a `splice(A,B,C)`, then there is nothing to check. If it is the first, then we must verify that `A`, `B` and `C` do not satisfy case 1b of claim 1 or case 3a of claim 3. But this is obvious, since if they did, the CAS would succeed.

Case 9 Suppose s is an execution of the CAS on line 32 or 37 of `tryAppend` that fails. We must show that s does not cause claim 1d to become false. Suppose step s attempts to CAS `X->right` from `null` to a node `Y`. The CAS fails, so $\texttt{X} \not\to \texttt{null}$ in the configuration before s. Thus, `X` cannot be $\texttt{U}_{k-1}$ in claim 1d, so s cannot cause that claim to become false.

Case 10 Suppose s is a successful CAS on line 49, 109 or 123 that removes a Descriptor `desc` containing `(X,Y,Z)` from the `leftDesc` or `rightDesc` field of a node. We must show that claim 6 holds in the configuration after s.

To prove claim 6a, suppose that $\texttt{X} \neq \texttt{null}$. Before `desc` was created at line 108 of `spliceUnmarkedLeft(X,Y,Z)` or line 122 of `spliceUnmarkedRight(X,Y,Z)`, we had $\texttt{X} \to \texttt{Y}$ at line 107 or 120. We show that, at some time between the creation of `desc` and s, $\texttt{X} \not\to \texttt{Y}$. Prior to s, `help(desc)` is called at line 48, 106 or 119, respectively. That call to `help` completes a `splice(X,Y,Z)` operation. If the `splice` terminates on line 96, then $\texttt{X} \not\to \texttt{Y}$. Otherwise, the `splice` performs line 99. Either $\texttt{X} \not\to \texttt{Y}$ before that CAS, or $\texttt{X} \to \texttt{Z}$ (where $\texttt{Z} \neq \texttt{Y}$) after the CAS. In all cases, $\texttt{X} \not\to \texttt{Y}$ at some point during the `help(desc)`, prior to the CAS that removes `desc` from `X->rightDesc`. Thus, `X->right` has changed from `Y` to some other value before s. It follows from Lemma 25 that $\texttt{X->right} >_c \texttt{Y}$ in the configuration after s.

To prove claim 6b, suppose that $\texttt{Z} \neq \texttt{null}$. The Descriptor `desc` was created at line 108 of `spliceUnmarkedLeft(X,Y,Z)` or line 122 of `spliceUnmarkedRight(X,Y,Z)`, each called by an instance $rr_{\texttt{Y}}$ of `removeRec(Y)`. Consider the configuration C' before $rr_{\texttt{Y}}$ executes line 69, when $\texttt{Y} \to \texttt{Z}$. We consider two cases.

If `Y` is finalized in C', then it was finalized by a `splice(X',Y,Z')`. Since, in the configuration before s, there has been a call to `splice(X',Y,Z')` and `desc` containing `(X,Y,Z)` has been installed in a node, we have $\texttt{X}' = \texttt{X}$ and $\texttt{Z}' = \texttt{Z}$ by induction hypothesis 4. By Lemma 24, the `splice` was not called at line 78 since either $\texttt{X} <_p \texttt{Y} <_p \texttt{Z}$ or $\texttt{X} >_p \texttt{Y} >_p \texttt{Z}$. So, the `splice` was called by `help(desc')` for some Descriptor `desc'`, also containing `(X,Y,Z)`, that was installed in the same node as `desc` prior to the creation of `desc`. Thus, `desc'` has been removed from a node before s. By induction hypothesis 6, the claim is true in the configuration before s, and therefore also in the configuration after s.

If `Y` is not finalized in C', then by induction hypotheses 1 and 3, we have $\texttt{Y} \leftarrow \texttt{Z}$ in C', when $rr_{\texttt{Y}}$ executes line 69. Later, at line 107 or 120 before `desc` is created, we have that either $\texttt{X} = \texttt{null}$ or $\texttt{X} \to \texttt{Y}$. Prior to s, `help(desc)` is called at line 48, 106 or 119, respectively. That call to `help` completes a `splice(X,Y,Z)` operation. If it performs line 98, then $\texttt{Y} \not\leftarrow \texttt{Z}$ after that CAS. Otherwise, it terminated at line 96. This means that $\texttt{X} \neq \texttt{null}$ and $\texttt{X} \not\to \texttt{Y}$. We know that, in an earlier configuration, $\texttt{X} \to \texttt{Y}$, so `X->right` has changed from `Y` to another value. This could only have been done by line 99 of a `splice(X,Y,Z')` operation. By induction hypothesis 4, $\texttt{Z} = \texttt{Z}'$. By Lemma 24, this `splice` could not have been called at line 78, so it was called by `help(desc')`. This `splice` also performed a `CAS(Z->left,Y,X)` at line 98. After

this step, Y $\not\leftarrow$ Z. If this CAS was performed after C' (when Y $\leftarrow$ Z), then it follows from Lemma 25 that Z->left $<_c$ Y at all times after the CAS. If this CAS was performed before C', then desc$'$ $\neq$ desc (since desc had not yet been created at C') and desc$'$, which also contains (X,Y,Z) has been removed from a node before s. In this case, the claim follows from the induction hypothesis.

Case 11 An execution of line 44 cannot change the status field of a node from finalized to marked, by Lemma 28. Hence, this step cannot cause any of the claims to become false. ◀

▶ **Theorem 7.** *Under Assumption 6, the implementation in Figure 4 is linearizable.*

Proof. We choose linearization points as follows.

- Each getHead is linearized when it reads Head.
- Each tryAppend is linearized when it performs the CAS on Head at line 36.
- Each remove and find can be linearized at any time during its execution.

Results returned by getHead and tryAppend are clearly consistent with this choice of linearization points. The remove operation does not return any result. So, it remains to show that find operations return results consistent with the linearization.

Recall that $V_1, V_2, \ldots$ are the active nodes in the order they are appended to the list, and $ts_1, ts_2, \ldots$ are their timestamps. Consider a find(V_j,t) operation. By assumption 6.4, V_j is active, so the local variable cur is initially set to an active node. Since cur is updated by following left pointers, cur is always either null or an active node, by Lemma 25.

We next prove the invariant that if cur $= V_i$ and $i < j$, then $ts_{i+1} >$ t. When cur is initialized at line 21, $i = j$, so the claim holds. Suppose cur is updated from node V_i to node $V_{i'}$ at line 23 (because $ts_i >$ t). Since $V_{i'}$ is read from V_i->left we have $i' < i$, by Lemma 25. If $i' + 1 = i$, then we know that $ts_{i'+1} >$ t. If $i' < i - 1$, then each of the nodes V_k with $i' < k < i$ are finalized, by Invariant 33. By Corollary 27, remove has been called on each of these nodes V_k. By Assumption 6.3, this means that t $\notin [ts_k, ts_{k+1})$ for each such k. So, t $\notin [ts_{i'+1}, ts_i)$ and we know that $ts_i >$ t. Hence, t $< ts_{i'+1}$, as required.

It follows that if the find(V_j,t) returns a node V_i where $i < j$, then V_i is the rightmost node ever appended to the list that is to the left of V_j and has $ts_i \leq$ t. Moreover, by assumption 6.4, no remove(V_i) has been invoked prior to the find's linearization point. Thus, the value returned by the find is consistent with its linearization point.

If the find(V_j,t) returns V_j, then $ts_j \leq$ t by the test on line 22. So, it suffices to show that no remove(V_j) is linearized before the find. If, at the linearization point of the find, V_j is the last node that has been appended to the list then Assumption 6.2 ensures that remove(V_j) has not been invoked. If a successful tryAppend(V_j, V_{j+1}) has been linearized before the find, then $ts_{j+1} > t$ by Assumption 6.5. By Assumption 6.3, this implies that remove(V_j) has not been invoked before the linearization point of the find.

Finally, we consider the case where find returns null. Consider the last execution of line 23 that changes cur from V_i to null. Then $ts_i >$ t and V_i->left $=$ null. By Invariant 33, for all $k < i$, V_k is finalized. By Corollary 27, remove has been called on each of these nodes V_k. By Assumption 6.3, this means that t $\notin [ts_k, ts_{k+1})$ for each such k. So, t $\notin [ts_1, ts_i)$. Since we know t $< ts_i$, it follows that t $< ts_1$, so no node has ever been appended to the list with a timestamp less than or equal to t. Thus, it is consistent with the linearization to return the result null. ◀

The freedom to linearize remove and find operations at any time during their execution results from Assumption 6.3, which essentially ensures there are no dangerous races between remove and find operations.

C.2 Amortized Time Bounds for Version Lists

The `tryAppend` operation performs $O(1)$ steps. Some clever tricks from [3] can be used to compute the function `p` using $O(1)$ machine instructions.

We consider a `tryAppend` operation to be *successful* if its CAS at line 36 successfully updates the `Head` pointer. By Theorem 8, the worst-case number of steps performed by a `remove` operation in an execution with L successful `tryAppend` operations is $O(\log L)$. We now prove that the amortized number of steps per `remove` operation is constant. To prove this, we first prove a lemma that implies at most one call to `removeRec(Y)` can recurse. (Interestingly, this lemma will also be useful later in bounding the amount of space used.)

▶ **Lemma 34.** *Among all calls to* `splice(*,Y,*)` *at line 78,* `spliceUnmarkedLeft(*,Y,*)` *and* `spliceUnmarkedRight(*,Y,*)` *in an execution, at most one performs a successful CAS at line 97, 109 or 123, and hence at most one can return true.*

Proof. It follows from Invariant 33.4 that any two such calls that perform a successful CAS at line 97, 109 or 123 must have the same triple of arguments `(X,Y,Z)`. If `splice(X,Y,Z)` is called at line 78, then `X` $<_p$ `Y` $>_p$ `Z`, by the test at line 77. It follows from Lemma 24, that there cannot be calls to two different routines that both return true. That is, the successful calls are all to `splice`, all to `spliceUnmarkedLeft`, or all to `spliceUnmarkedRight`.

By Lemma 28, only the first `splice(X,Y,Z)` to perform the CAS at line 97 can succeed in finalizing `Y`.

To derive a contradiction, suppose two calls to `spliceUnmarkedLeft(X,Y,Z)` perform a successful CAS at line 109. Let these two calls be sul_1 and sul_2, in the order that they perform their successful CAS. Let `desc` be the Descriptor installed in `X->rightDesc` by sul_1. We consider two cases.

First, suppose `desc` is replaced by another Descriptor before sul_2 reads `X->rightDesc` at line 104. By Invariant 33.6, `X` $\not\rightarrow$ `Y` when sul_2 performs line 107 so sul_2 cannot perform the CAS at line 109, a contradiction.

Now, suppose `desc` is still in `X->rightDesc` when sul_2 reads that field at line 104. Then, sul_2 calls `help(desc)` at line 106, which calls `splice(X,Y,Z)`. Prior to this call to `splice`, we had `X` $\rightarrow$ `Y` when sul_1 performed line 107. Then, during sul_2's call to `splice`, we have `X` $\not\rightarrow$ `Y` at line 96 or before or after the CAS at line 99 is performed. By Lemma 25, `X` $\not\rightarrow$ `Y` at all times after this `splice` returns. Thus, when sul_2 performs the test at line 107, it returns false and does not perform the CAS at line 109, a contradiction.

The proof that no two calls to `spliceUnmarkedRight(X,Y,Z)` can perform a successful CAS at line 123 is symmetric to the proof for `spliceUnmarkedLeft`. ◀

▶ **Theorem 9.** *The* `tryAppend` *and* `getHead` *operations take $O(1)$ steps. The amortized number of steps for* `remove` *is $O(1)$.*

Proof. As argued in the proof of Theorem 8, the total number of steps performed by `remove` operations is proportional to the number of calls to `remove` and `removeRec`. So, it suffices to show that there are at most $O(R)$ calls to `removeRec`.

There are at most R calls to `removeRec` at line 50 of the `remove` operation. Each recursive call to `removeRec` is preceded by a (distinct) call to `splice` at line 78, `spliceUnmarkedLeft` or `spliceUnmarkedRight` that returns true. Thus, it suffices to show that at most $O(R)$ such calls can return true. By Lemma 26, each call to `splice(*,Y,*)`, `spliceUnmarkedLeft(*,Y,*)` or `spliceUnmarkedRight(*,Y,*)` is on a marked node `Y`. Since there are at most R marked nodes, the claim follows from Lemma 34. ◀

C.3 Space Bound for Version Lists

In this section, we prove Theorem 10, which bounds the number of nodes that remain in the list. The proof of this theorem requires several lemmas. We say that a node `X` is *lr-reachable* in a configuration if it can be reached from an active, unfinalized node by following a sequence of `left` and `right` pointers.

Consider a finite execution. Define the priority tree T for the execution as follows. Consider the sequence of all nodes that become active during the execution, with the priorities assigned to them by the algorithm. Arrange the sequence into a priority tree by choosing the minimum priority node as the root and recursively constructing the left and right subtrees from the subsequences to the left and right of this node. As remarked earlier, our scheme for assigning priorities ensures that the choice of the root in this recursive construction is always unique. Furthermore, the sequence of active nodes ordered by $<_c$ is an in-order traversal of T. See Figure 3 for an example. T is a single tree defined for the entire execution. Whenever we refer to ancestors or descendants of nodes, we are referring to relationships in the tree T. By convention, we consider a node to be its own ancestor and descendant.

We say that a node is *removable* in a configuration if `remove` has been invoked on it before that configuration. As the execution proceeds, certain removable nodes within T will become marked and finalized. At any given configuration during the execution, we wish to bound the number of nodes in the tree that are removable but not finalized.

▶ **Lemma 35.** *If* `splice(X,Y,Z)` *is called, then* `X` $<_p$ `Y` $>_p$ `Z` *or* `X` $<_p$ `Y` $<_p$ `Z` *or* `X` $>_p$ `Y` $>_p$ `Z`*.*

Proof. If `splice(X,Y,Z)` is called on line 78, then `X` $<_p$ `Y` $>_p$ `Z` by the test at line 77. Otherwise, `splice(X,Y,Z)` is called at line 59 of a call to `help(desc)` on some Descriptor `desc` containing `X`,`Y`,`Z`. Since `desc` was created by a call to either `spliceUnmarkedLeft(X,Y,Z)` or `spliceUnmarkedRight(X,Y,Z)`, we must have either `X` $>_p$ `Y` $>_p$ `Z` or `X` $<_p$ `Y` $<_p$ `Z`, by the tests preceding those calls at line 86 and 90. ◀

▶ **Lemma 36.** *In any configuration, if* `X` *is an active node then* `X->left` *and* `X->right` *each point to a proper ancestor of* `X` *in* T*, a proper descendant of* `X` *in* T*, or* `null`*.*

Proof. Suppose the claim is false. Consider the first time it is violated. We consider all steps that can change `left` or `right` pointers.

Suppose the violation is caused by a CAS at line 98 or 99 of some `splice(A,B,C)` operation. This CAS either updates `A->right` to `C`, or `C->left` to `A`. Since this causes a violation of the lemma, `A` and `C` must be non-`null` and neither `A` nor `C` is an ancestor of the other. When the CAS occurs, `A` $\leftarrow$ `B` $\rightarrow$ `C`, by Invariant 33.4. Since no violation of the lemma occurred before this step, one of `A` and `B` must be a proper ancestor of the other, and one of `B` and `C` must be a proper ancestor of the other. The only way this can happen is if `B` is a proper ancestor of both `A` and `C`. Thus, `A` $>_p$ `B` $<_p$ `C`. This contradicts Lemma 35.

Suppose the violation is caused by line 32 which sets `A->right` to `B`. Since we have `A` $\leftarrow$ `B` earlier at line 30, this would violate the assumption that no violation occurred before this step.

The violation cannot be caused by line 35 or line 37 because they set a `left` or `right` pointer to an adjacent node in the $<_c$ ordering, and two adjacent nodes in an in-order traversal of T must be ancestors or descendants of each other. ◀

▶ **Corollary 37.** *If a call to* `removeRec(X)` *calls* `removeRec(Y)` *recursively, then* `Y` *is an ancestor of* `X`*.*

Proof. `Y` is read from `X->left` or `X->right`. The `validAndFrozen` test before any recursive call ensures that `Y` is not `null`. Thus, by Lemma 36, `Y` is either an ancestor or descendant of `X`. Since `removeRec(Y)` is called only after testing that `Y` $<_p$ `X`, `Y` must be an ancestor of `X`. ◀

Incidentally, it also follows from Lemma 36 that we never compare the priorities of two nodes with the same priority during the `removeRec` operation.

▶ **Lemma 38.** *When a node is finalized, either all nodes in its left subtree in T or all nodes in its right subtree in T are finalized.*

Proof. Consider the call to `splice(A,B,C)` that finalizes node `B`. By Lemma 35, either `A` $<_p$ `B` or `C` $<_p$ `B`. We prove the lemma for the case where `A` $<_p$ `B`; the other case is symmetric. By Lemma 23, there was an earlier configuration C when `A` $\leftarrow$ `B`. We consider two cases.

If `A` is `null`, then by Invariant 33 in the configuration when `null` $\leftarrow$ `B`, there are no non-finalized nodes `D` such that `D` $<_c$ `B`. In other words, `B` is the first non-finalized node in an in-order traversal of T. So, all nodes in the left subtree of `B` in T are finalized.

Now, suppose `A` is not `null`. By Lemma 36, `B` is a descendant of `A` in T since `A` $<_p$ `B`. By Lemma 25, `A` $<_c$ `B`, so `B` is in the right subtree of `A`. By Invariant 33, there is no non-finalized node `D` such that `A` $<_c$ `D` $<_c$ `B` in configuration C. In other words, there is no non-finalized node between `A` and `B` in an in-order traversal of T. Thus, every node in `B`'s left subtree in T is finalized in C. ◀

We want to bound the number of unfinalized nodes. We divide them into three types as follows.

- Type 0: neither of the node's two subtrees contains an unfinalized node.
- Type 1: exactly one of the node's two subtrees contains an unfinalized node.
- Type 2: both of the node's two subtrees contain an unfinalized node.

The following lemma shows that it will be sufficient to bound the number of unfinalized nodes of Types 0 and 1.

▶ **Lemma 39.** *For any configuration and any subtree of T that contains unfinalized nodes, the number of unfinalized nodes of Type 2 in the subtree is less than the number of unfinalized nodes of Type 0 in the subtree.*

Proof. We prove the claim by induction on the height of the subtree.

Base case: Consider a subtree of T consisting of a single leaf node. If the node is finalized, the claim holds trivially. If the node is not finalized, then it is of type 0, so the claim holds.

Induction step: Assume the claim holds for subtrees of height less than h. Consider a subtree of height h with root r. We consider several cases.

Case 1: Suppose r is finalized. By Lemma 38, either the left or right subtree of r contains only finalized nodes. The claim follows by applying the induction hypothesis to the other subtree.

Case 2: Suppose r is unfinalized and of Type 0. Then r is the only unfinalized node in the subtree, so the number of unfinalized nodes of Type 2 is 0.

Case 3: Suppose r is unfinalized and of Type 1. Then, either the left or right subtree r contains only finalized nodes. The claim follows by applying the induction hypothesis to the other subtree (which does contain unfinalized nodes).

Case 4: Suppose r is unfinalized and of Type 2. Let T_L and T_R be the left and right subtrees of r. By definition, both contain unfinalized nodes. So by the induction hypothesis,

$$\begin{aligned}
& \#\text{ Type-2 unfinalized nodes in the subtree} \\
= \;& 1 + (\#\text{ Type-2 unfinalized nodes in } T_L) + (\#\text{ Type-2 unfinalized nodes in } T_R) \\
\leq \;& 1 + (\#\text{ Type-0 unfinalized nodes in } T_L) - 1 + (\#\text{ Type-0 unfinalized nodes in } T_R) - 1 \\
= \;& (\#\text{ Type-0 unfinalized nodes in subtree}) - 1 \\
< \;& \#\text{ Type-0 unfinalized nodes in subtree.}
\end{aligned}$$

◀

▶ **Lemma 40.** *In each configuration, every active, unfinalized node is lr-reachable.*

Proof. Assume the configuration has some active, unfinalized nodes. Let $\mathtt{U}_1, \ldots, \mathtt{U}_k$ be the sequence of active, unfinalized nodes in the configuration, ordered by $<_c$.

We first show that $\mathtt{U}_k$ is reachable. We define a sequence of nodes $\mathtt{W}_1, \ldots, \mathtt{W}_\ell$ inductively. Let $\mathtt{W}_1$ be the value of `Head`. If $\mathtt{W}_i$ is not finalized or if $\mathtt{null} \leftarrow \mathtt{W}_i$, then $\ell = i$ and $\mathtt{W}_i$ is the last node in the sequence. Otherwise, $\mathtt{W}_i$ is finalized and we define $\mathtt{W}_{i+1} = \mathtt{W}_i\mathtt{->left}$. We show that $\mathtt{U}_k = \mathtt{W}_\ell$.

We prove by induction on i that $\mathtt{U}_k \leq_c \mathtt{W}_i$. For $i = 1$, ince $\mathtt{U}_k$ is active, $\mathtt{U}_k \leq_c \mathtt{W}_1$, by Observation 31. For $1 \leq i < \ell$ suppose $\mathtt{U}_k \leq_c \mathtt{W}_i$ to prove that $\mathtt{U}_k \leq_c \mathtt{W}_{i+1}$. Since $i < \ell$, $\mathtt{W}_i$ is finalized, so $\mathtt{U}_k \neq \mathtt{W}_i$. Thus, $\mathtt{U}_k <_c \mathtt{W}_i$. It follows from Invariant 33.5b that $\mathtt{U}_k \leq_c \mathtt{W}_{i+1}$.

Thus, we have $\mathtt{U}_k \leq_c W_\ell$. If $\mathtt{W}_\ell$ were finalized, then $\mathtt{null} \leftarrow W_\ell$, which would contradict Invariant 33.5b. Thus, $\mathtt{W}_\ell$ is not finalized, and we have $\mathtt{U}_k = \mathtt{W}_\ell$ since $\mathtt{U}_k$ is the largest unfinalized node with respect to $<_c$. Thus, $\mathtt{U}_k$ is lr-reachable via the sequence of nodes $\mathtt{W}_1, \ldots, \mathtt{W}_\ell = \mathtt{U}_k$.

By Invariant 33.1, if $\mathtt{U}_{i+1}$ is lr-reachable, then so is $\mathtt{U}_i$. The claim follows by induction. ◀

▶ **Lemma 41.** *If an active node* `X` *ceases being lr-reachable, it never subsequently becomes lr-reachable again.*

Proof. It follows from Invariant 33.4 that when a node's `left` or `right` pointer is modified, the node is not finalized. By Invariant 33, a `left` or `right` pointer can only be changed to an unfinalized node or `null`. (For example, Invariant 33.1 implies that $\mathtt{U}_i$`->right` can only be changed to $\mathtt{U}_{i+1}$ or `null`.) If the new value of the pointer is not `null`, it is an unfinalized node, which was already lr-reachable by definition, so this change does not cause any node to become lr-reachable. ◀

▶ **Lemma 42.** *After an instance of* `splice(*,Y,*)` *executes line 99,* `Y` *is not lr-reachable.*

Proof. Let $\mathtt{U}_i$ be the largest unfinalized node that is less than `Y` (with respect to $<_c$), or `null` if there is no such node. Similarly, let $\mathtt{U}_{i+1}$ be the smallest unfinalized node that is greater than `Y`, or `null` if there is no such node. By Invariant 33, if an unfinalized node points to a finalized node, the finalized node points only to unfinalized ones. Thus if there is a path of `left` or `right` pointers from an unfinalized node to a finalized node, there cannot be two finalized nodes in a row on the path. So, if `Y` is lr-reachable, there is some unfinalized node that points directly to it. Thus, by Invariant 33, `Y` can only be lr-reachable if either $\mathtt{U}_i \rightarrow \mathtt{Y}$ or $\mathtt{Y} \leftarrow \mathtt{U}_{i+1}$.

Since the `splice(*,Y,*)` has executed line 99, and therefore also line 97, `Y` is finalized, by Lemma 28. Since the `splice` has also executed lines 98 and 99, it follows from Invariant 33 that $\mathtt{U}_i \not\rightarrow \mathtt{Y}$ and $\mathtt{Y} \not\leftarrow \mathtt{U}_{i+1}$. So, by the claim proved in the first paragraph, `Y` is not lr-reachable. ◀

▶ **Lemma 43.** *At the time a node* `Y` *is finalized by a* `splice(X,Y,Z)`*, the following hold with respect to the ordering* $<_c$*.*
If $\texttt{X} \neq \texttt{null}$ *then* `X` *is the greatest unfinalized node that is less than* `Y` *and* $\texttt{X} \leftrightarrow \texttt{Y}$*.*
If $\texttt{X} = \texttt{null}$ *then* `Y` *is the smallest unfinalized node.*
If $\texttt{Z} \neq \texttt{null}$ *then* `Z` *is the least unfinalized node that is greater than* `Y` *and* $\texttt{Y} \leftrightarrow \texttt{Z}$*.*

Proof. Consider the configuration before the step that finalizes `Y`. By Invariant 33.4, we have $\texttt{X} \leftarrow \texttt{Y} \rightarrow \texttt{Z}$.

Since `Y` is not finalized, if `Z` is a finalized node, then a `splice(Y,Z,*)` must have been invoked by Invariant 33.1, but this would violate Invariant 33.7. So, `Z` is not a finalized node. By Invariant 33.1, if `Z` is not `null`, then it is the smallest unfinalized node greater than `Y` and $\texttt{Y} \leftarrow \texttt{Z}$.

Suppose `X` is not `null`. If `X` is finalized, then a `splice(*,X,Y)` must have been invoked, violating Invariant 33.7. So, `X` is not finalized. By Invariant 33.1 `X` is the greatest unfinalized node that is less than `Y`. We had $\texttt{X} \rightarrow \texttt{Y}$ at line 96 before `Y` is finalized, so when `Y` is finalized, we have $\texttt{X->right} \geq_c \texttt{Y}$, by Lemma 25. By Invariant 33.1, $\texttt{X->right} \leq_c \texttt{Y}$ since neither `X` nor `Y` is finalized. Thus, $\texttt{X} \rightarrow \texttt{Y}$.

If `X` is `null`, it follows from Invariant 33.2 that `Y` is the smallest unfinalized node. ◀

▶ **Lemma 44.** *If, in some configuration* C*,* `X` *is frozen and* `X->left` *is not a descendant of* `X`*, then* `X->left` *never changes after* C*.*

Proof. Let `V` be `X->left` in C. To derive a contradiction, suppose `X->left` changes from `V` to some other value after C. This change must be performed by a `splice(*,V,X)` operation. By Lemma 23, `V` is not `null`. By assumption, `V` is not a descendant of `X` so by Lemma 36, `V` is an ancestor of `X`. Since $\texttt{V} <_p \texttt{X}$, the `splice(*,V,X)` operation must have been called by a `help(desc)`, where `desc` is a Descriptor containing `(*,V,X)` that was stored in `X->leftDesc`. Since `X->leftDesc` was `frozen` before C, `desc` was removed from `X->leftDesc` before C. By Invariant 33.6, $\texttt{X->left} <_c \texttt{V}$ at some time before C. By Lemma 25, $\texttt{X->left} <_c \texttt{V}$ at C, a contradiction. ◀

We say a `removeRec` *is at* `X` if an instance of `removeRec(X)` has been called and it has neither terminated nor made a recursive call.

If `X` is a finalized node, we wish to define the instance of `removeRec(X)` that *caused* `X` to become finalized. `X` was finalized by a call to `splice(*,X,*)`. If this `splice` is called from line 78 of an instance of `removeRec(X)`, then that is the instance that caused `X` to become finalized. Otherwise, the `splice` must have been called by `help(desc)`, where `desc` is a Desciptor that was installed in a node by a call to `spliceUnmarkedLeft(*,X,*)` or `spliceUnmarkedRight(*,X,*)` at line 86 or 90 of an instance of `removeRec(X)`. In this case, that instance is the one that cause `X` to become finalized. By Lemma 34, there is a unique `removeRec(X)` that caused `X` to be finalized.

▶ **Lemma 45.** *If, in a configuration* C*, a node* `X` *is of type 0 or type 1 and is frozen but not finalized, then there is a* `remove` *or* `removeRec` *at a descendant of* `X` *in* C*.*

Proof. We prove the claim by induction on the height of the subtree rooted at `X`.

Base case (`X` is a leaf of T): To derive a contradiction, suppose the claim is false for `X`. Then, the call to `remove(X)` that froze `X` called `removeRec(X)` and that call either terminated or recursed to another node. Since `X` is not finalized in C, the test at line 70 did not cause `removeRec(X)` to terminate. By Lemma 36, the nodes $\texttt{X}_L$ and $\texttt{X}_R$ read at line 68 and 69

are either `null` or ancestors of `X`, so the test at line 77 succeeded and `splice(`X_L`,X,`X_R`)` was called at line 78.

Consider the test at line 96 of `splice(`X_L`,X,`X_R`)`. By Invariant 33, we cannot have X_L`->right` $>_c$ `X`, since `X` is not finalized. If X_L`->right` $=$ `W` $<_c$ `X`, then by Invariant 33, `W` is finalized and a `splice(`X_L`,W,X)` has been called. By Lemma 36, this means `W` $<_p$ `X`. So, the `splice(`X_L`,W,X)` was called by `help` on a Descriptor stored in `X->leftDesc`. By Lemma 29, this Descriptor was removed before the call to `removeRec(X)`. By Invariant 33.6, $X_L \not\to$ `W` when line 96 is executed, a contradiction.

Thus, the test at line 96 of `splice(`X_L`,X,`X_R`)` cannot cause the splice to terminate. Since the `removeRec(X)` is no longer at `X`, the `splice(`X_L`,X,`X_R`)` must execute line 97 before C. By Lemma 28, this means `X` is finalized before C, a contradiction.

Induction step (`X` is an internal node of T): We assume the claim holds for proper descendants of `X` and prove that it holds for `X`. To derive a contradiction, assume the claim does not hold at `X`. Thus, at configuration C, `X` is frozen but not finalized and there is no `remove` or `removeRec` at a descendant of `X`.

Let S be the set of proper descendants `Y` of `X` such that in C, `Y` is finalized and there are no unfinalized nodes between `Y` and `X` with respect to the ordering $<_c$. Let S_L and S_R be the subsets of S in `X`'s left and right subtree, respectively.

▷ Claim. There is a configuration C_L before C, such that

(a) in C_L, `X->left` does not point to any node in S_L,

(b) in C_L, if `X->left` $\neq$ `null` then `X->left->right` $=$ `X`, and

(c) a `removeRec(X)` is invoked between C_L and C.

Proof. If S_L is empty, let C_L be the configuration when the `right` pointer of `X`'s predecessor in $<_c$ is updated to point to `X`. (If `X` is the minimum node with respect to $<_c$, then let C_L be the configuration when the head pointer is changed to `X`.) The first part of the claim is vacuously true. In C_L, if `X->left` $\neq$ `null`, then `X->left->right` $=$ `X`. Since `X` is frozen in C, there must have been a call to `removeRec(X)` between C_L and C.

If S_L is non-empty, let `W` be the node in S_L that was finalized last. Consider the `removeRec(W)` $rr_{\texttt{W}}$ that causes `W` to be finalized. Let $\texttt{W}_L$ and $\texttt{W}_R$ be the values $rr_{\texttt{W}}$ reads from `W->left` and `W->right` at line 68 and 69. By Invariant 33.4, we have $\texttt{W}_L \leftarrow \texttt{W} \rightarrow \texttt{W}_R$ when `W` becomes finalized. By Lemma 43, $\texttt{W}_L$ and $\texttt{W}_R$ are the closest unfinalized nodes on either side of `W` in the $<_c$ ordering when `W` becomes finalized (or `null` if there is no such node). So, by definition $\texttt{W}_R = \texttt{X}$. By Lemma 36, $\texttt{W}_L$ is either an ancestor of `X` (or `null`) if S_L contains all nodes `X`'s left subtree, or a descendant of `X`, otherwise.

We consider two cases to show that some call to `splice(*,W,*)` executes line 99 before C. Suppose $rr_{\texttt{W}}$ calls `splice(`$\texttt{W}_L$`,W,`$\texttt{W}_R$`)` at line 78 and successfully performs line 97 to finalize `W`. Then, $rr_{\texttt{W}}$ must also perform line 99 before C, since there is no `removeRec` at `W` in C, by assumption.

Suppose $rr_{\texttt{W}}$ calls `spliceUnmarkedLeft` or `spliceUnmarkedRight` and installs a Descriptor containing `(`$\texttt{W}_L$`,W,`$\texttt{W}_R$`)`. Then, at line 107 or 120 we have $\texttt{W}_L \rightarrow \texttt{W}$ if $\texttt{W}_L$ is not `null`. After installing the Descriptor, $rr_{\texttt{W}}$ calls `splice(`$\texttt{W}_L$`,W,`$\texttt{W}_R$`)` via line 111 or 125. This call must complete before C, since there is no `removeRec` at `W` in C, by assumption. If this `splice` sees that $\texttt{W}_L$ is not `null` and $\texttt{W}_L \not\to \texttt{W}$ at line 96, then $\texttt{W}_L$`->right` must have been changed from `W` to another value. This can only be done by some `splice(*,W,*)` performing line 99. (Since `W` is not `null`, this change was not by a `tryAppend`.) Otherwise, $rr_{\texttt{W}}$'s call to `splice(`$\texttt{W}_L$`,W,`$\texttt{W}_R$`)` will perform line 99 itself.

As argued above, $rr_{\texttt{W}}$ calls `splice(`$\texttt{W}_L$`,W,X)`. By Invariant 33.4, *all* calls to `splice(*,W,*)` are of the form `splice(`$\texttt{W}_L$`,W,X)`. As argued above, some `splice(`$\texttt{W}_L$`,W,X)` executes line 99 before C. Let C_L be the configuration after the first execution of line 99 by any call to `splice(`$\texttt{W}_L$`,W,X)`. Prior to C_L, a `splice(`$\texttt{W}_L$`,W,X)` executes a `CAS(X->left,W,`$\texttt{W}_L$`)` at line 98.

By Lemma 43, $\texttt{W} \leftrightarrow \texttt{X}$ in the configuration before W was finalized. Thus, when the first CAS(`X->left,W,`$\texttt{W}_L$) is performed, it must succeed, since no other step can change `X->left`. By Lemma 25, this means that in all configurations after C_L, `X->left` $\leq_c \texttt{W}_L$. Since $\texttt{W}_L$ is not a finalized node when W is finalized, it must be either `null` or less than all nodes in S_L with respect to the order $<_c$. Thus, at all times after C_L, either `X->left` is `null` or `X->left` $<_c$ Y for all $\texttt{Y} \in S_L$. This completes the proof of Claim a.

We now prove Claim b. Assume `X->left` is not `null` in C_L.

We first show that if W_L is not `null`, then the execution of `CAS(`$\texttt{W}_L$`->right,W,X)` immediately before C_L is successful. Before W was finalized by line 97 of a `splice(`$\texttt{W}_L$`,W,X)` we have $\texttt{W}_L \rightarrow \texttt{W}$ at line 96. The only way that $\texttt{W}_L$`->right` can change is by line 99 of a call to `splice(`$\texttt{W}_L$`,W,X)`. (Since W is not `null`, this change cannot be by a `tryAppend`.) Thus, the first such execution succeeds. So, if $\texttt{W}_L \neq$ `null`, then we have $\texttt{W}_L \rightarrow \texttt{X}$ in C_L.

As argued above, after the first execution of line 98 by a call to `splice(`$\texttt{W}_L$`,W,X)`, we have $\texttt{W}_L \leftarrow \texttt{X}$. To derive a contradiction, suppose $\texttt{W}_L \not\leftarrow \texttt{X}$ at C_L. Then `X->left` has been changed by line 98 of a `splice(*,`$\texttt{W}_L$`,X)`, which means $\texttt{W}_L \neq$ `null`. When this change occurs, we have $\texttt{W}_L \rightarrow \texttt{X}$, by Invariant 33.4. But this cannot be true before the step preceding C_L, as argued in the previous paragraph. This contradiction shows that $\texttt{W}_L \leftarrow \texttt{X}$ in C_L. So, in C_L, `X->left` is $\texttt{W}_L$ (which is assumed to be non-`null`) and $\texttt{W}_L$`->right` is X, which establishes claim b.

We now prove Claim c. The routine $rr_{\texttt{W}}$ either calls `splice(`$\texttt{W}_L$`,W,`$\texttt{W}_R$`)` directly at line 78 or via `help` after successfully storing a Descriptor containing `(`$\texttt{W}_L$`,W,`$\texttt{W}_R$`)`. Thus, $rr_{\texttt{W}}$ terminates or recurses between C_L and C. We consider two cases.

If X is frozen in C_L, then $rr_{\texttt{W}}$ cannot terminate; it must recurse. If $\texttt{W}_L$ is `null`, then $rr_{\texttt{W}}$ will recurse to X. If $\texttt{W}_L$ is an ancestor of X, then $rr_{\texttt{W}}$ will recurse to X, because $\texttt{X} >_p \texttt{W}_L$. If $\texttt{W}_L$ is a descendant of X, then $\texttt{W}_L$ is unfinalized at C (by definition of S) and it is not of type 2, since all nodes in $\texttt{W}_L$'s right subtree are finalized at C, and hence by the induction hypothesis, $\texttt{W}_L$ is not frozen before C. Thus, $rr_{\texttt{W}}$ will also recurse to X in this case. In all cases, there is a call to `removeRec(X)` between C_L and C.

If X is not frozen in C_L, then it becomes frozen by a `remove(X)` between C_L and C. By assumption, there is no `remove(X)` at X in C, so the `remove(X)` that froze X must call `removeRec(X)` between C_L and C.

This completes the proof of the claim. ◁

▷ Claim. There is a configuration C_R before C, such that

(a) in C_R, `X->right` does not point to any node in S_R,

(b) in C_R, if `X->right` $\neq$ `null` then `X->right->left` $=$ X, and

(c) a `removeRec(X)` is invoked between C_R and C.

The proof of this claim is symmetric to the previous one.

Combining the two claims, there is an instance $rr_{\texttt{X}}$ of `removeRec(X)` that is invoked after both C_L and C_R and before C. By assumption, $rr_{\texttt{X}}$ must terminate or recurse before C. It cannot terminate at line 70 before C, since X is not finalized in C. Let $\texttt{X}_L$ and $\texttt{X}_R$ be the nodes that $rr_{\texttt{X}}$ reads at line 68 and 69. We consider three cases to show that X is finalized before C, which is the desired contradiction.

Case 1 Suppose X is of type 0 in C. Then, S contains all of X's proper descendants. By the claims proved above and Lemma 25, $\texttt{X}_L$ and $\texttt{X}_R$ are not descendants of X. By Lemma 36 they must each be either null or an ancestor of X. Thus, $rr_{\texttt{X}}$ calls splice($\texttt{X}_L$,X,$\texttt{X}_R$) at line 78.

Let V be the value of X->left when $rr_{\texttt{X}}$ is invoked. In the configuration C_L, which is before $rr_{\texttt{X}}$ is invoked, X->left does not point to any node in X's left subtree. By Lemma 25, X->left cannot point to any node in X's left subtree after C_L. Thus, V is not a descendant of X. By Lemma 30, X is frozen when $rr_{\texttt{X}}$ is invoked. By Lemma 44, $\texttt{V} \leftarrow \texttt{X}$ at all times after $rr_{\texttt{X}}$ begins. Thus, $\texttt{V} = \texttt{X}_L$ and $\texttt{X}_L \leftarrow \texttt{X}$ at all times after $rr_{\texttt{X}}$ begins.

If $\texttt{X}_L = \texttt{null}$, then line 97 of the splice($\texttt{X}_L$,X,$\texttt{X}_R$) called at line 78 of $rr_{\texttt{X}}$ finalizes X before C, by Lemma 28.

Otherwise, let U be the value of $\texttt{X}_L$->right read at line 96. To derive a contradiction, suppose $\texttt{U} <_c \texttt{X}$. At line 96, we have $\texttt{X}_L \leftarrow \texttt{X}$ and $\texttt{X}_L \rightarrow \texttt{U}$ and X is unfinalized. By Invariant 33, this means a splice($\texttt{X}_L$,U,X) has been called and $\texttt{U} \rightarrow \texttt{X}$. By Lemma 36, U is either an ancestor or descendant of X. Since X is unfinalized, $\texttt{X}_L$ and U are lr-reachable at line 96. Since all nodes in X's left subtree are not lr-reachable at the earlier configuration C_L, it follows from the Lemma 41 that U cannot be a descendant of X. Thus, U is an ancestor of X. By Lemma 24, the splice($\texttt{X}_L$,U,X) must have been called by help on a Descriptor containing ($\texttt{X}_L$,U,X) that was stored in X->leftDesc. Since X is frozen before $rr_{\texttt{X}}$ begins, the fact that $\texttt{X}_L \rightarrow \texttt{U}$ during $rr_{\texttt{X}}$ contradicts Invariant 33.6. Thus, $\texttt{U} \geq_c \texttt{X}$. If $\texttt{U} = \texttt{X}$ then by Lemma 28, line 97 finalizes X before C. If $\texttt{U} >_c \texttt{X}$ then X is finalized by Invariant 33.

Case 2 Suppose that in C, X is of type 1 and all nodes of its right subtree are finalized. Then, by Lemma 36 and the first claim above, $\texttt{X}_R$ is either null or an ancestor of X. In C_L, X->left does not point to any element of S_L by the claim above, and it cannot point past an unfinalized node by Invariant 33. Thus, $\texttt{X}_L$ is the maximum node (with respect to $<_c$) in X's left subtree that is not finalized in C_L. Since all nodes in S_L are finalized in C_L, $\texttt{X}_L$ is not finalized in C either. It follows from Invariant 33.1 that throughout the period between C_L and C, $\texttt{X}_L \leftrightarrow \texttt{X}$.

Since all nodes in $\texttt{X}_L$'s right subtree are finalized in C, $\texttt{X}_L$ is not of type 2 in C. It follows from the induction hypothesis that X_L is not frozen in C. Since there is no remove($\texttt{X}_L$) at $\texttt{X}_L$ in C, $\texttt{X}_L$ is unmarked in C.

So, we have $\texttt{X}_L >_p \texttt{X} >_p \texttt{X}_R$ and $rr_{\texttt{X}}$ completes a call $sul_{\texttt{X}}$ to spliceUnmarkedLeft($\texttt{X}_L$,X,$\texttt{X}_R$) before C. Since X_L is unmarked at C, $sul_{\texttt{X}}$ does not terminate at line 105. Since $\texttt{X}_L \rightarrow \texttt{X}$ throughout its execution, $sul_{\texttt{X}}$ does not terminate at line 107. Thus, $sul_{\texttt{X}}$ performs the CAS at line 109.

To derive a contradiction, assume that this CAS fails. Then some other call to spliceUnmarkedLeft installed a Descriptor ($\texttt{X}_L$,X′,$\texttt{X}'_R$) in $\texttt{X}_L$->rightDesc between $sul_{\texttt{X}}$'s executions of line 104 and 109. If $\texttt{X}' >_c \texttt{X}$ then there was an earlier time when $\texttt{X}_L \leftarrow \texttt{X}'$ with $\texttt{X}_L <_c \texttt{X} <_c \texttt{X}'$ which means that X was finalized, a contradiction. If $\texttt{X}' = \texttt{X}$ then the removeRec(X) that installed the Descriptor ($\texttt{X}_L$,X′,$\texttt{X}'_R$) will ensure that X is finalized before C. Otherwise, $\texttt{X}' <_c \texttt{X}$, so $\texttt{X}' \in S_L$. So, X′ is finalized before C_L. Hence, there was a removeRec(X′) that performed a successful CAS at line 97, 109 or 123 that caused X′ to be finalized before C_L. This contradicts Lemma 34.

So, $sul_{\texttt{X}}$ performs the CAS at line 109 successfully, and then calls splice($\texttt{X}_L$,X,$\texttt{X}_R$) via line 111 and that call to splice must complete before C. It cannot terminate at line 96 because $\texttt{X}_L \rightarrow \texttt{X}$ throughout its execution. So it performs line 97, which

finalizes X before C, by Lemma 28.

Case 3 Suppose that in C, X is of type 1 and all nodes of X's left subtree are finalized. Although this case is nearly symmetric to the previous one, there are some differences due to asymmetries in the code.

Let V be the value of X->left when $rr_{\mathtt{X}}$ is invoked. In the configuration C_L, which is before $rr_{\mathtt{X}}$ is invoked, X->left does not point to any node in X's left subtree. By Lemma 25, X->left cannot point to any node in X's left subtree after C_L. Thus, V is not a descendant of X. By Lemma 30, X is frozen when $rr_{\mathtt{X}}$ is invoked. By Lemma 44, $\mathtt{V} \leftarrow \mathtt{X}$ at all times after $rr_{\mathtt{X}}$ begins. Thus, $\mathtt{V} = \mathtt{X}_L$ and $\mathtt{X}_L \leftarrow \mathtt{X}$ at all times after $rr_{\mathtt{X}}$ begins.

We next show that if $\mathtt{X}_L \neq$ null then $\mathtt{X}_L \rightarrow \mathtt{X}$ at all times between the invocation of $rr_{\mathtt{X}}$ and C. To derive a contradiction, assume that, at some configuration C' between the invocation of $rr_{\mathtt{X}}$ and C, $\mathtt{X}_L$->right $\neq$ X. Let U be the value of $\mathtt{X}_L$->right at C'. By Invariant 33.1, we have $\mathtt{X}_L \leftrightarrow \mathtt{U} \rightarrow \mathtt{X}$ at C' and a splice($\mathtt{X}_L$,U,X) has been called. By Lemma 40, X and therefore $\mathtt{X}_L$ and U are lr-reachable at line 96. Since all nodes in X's left subtree are not lr-reachable at the earlier configuration C_L, it follows from the Lemma 41 that U is not a descendant of X. Since we have $\mathtt{U} \rightarrow \mathtt{X}$, it follows from Lemma 36 that U is an ancestor of X, so $\mathtt{U} <_p \mathtt{X}$. Thus, the call to splice($\mathtt{X}_L$,U,X) must have been called by help on a Descriptor containing ($\mathtt{X}_L$,U,X) that was stored in X->leftDesc. Since X is frozen before $rr_{\mathtt{X}}$ begins, the fact that $\mathtt{X}_L \rightarrow \mathtt{U}$ during $rr_{\mathtt{X}}$ contradicts Invariant 33.6.

Let Z be the minimum (with respect to $<_c$) node in X's right subtree that is not finalized at C. By the second claim above, X->right $\geq_c$ Z at C_R. Since Z is not finalized at C, it follows from Lemma 25 and Invariant 33 that $\mathtt{X} \rightarrow \mathtt{Z}$ at all times between C_R and C. Thus $\mathtt{Z} = \mathtt{X}_R$ and $\mathtt{X} \rightarrow \mathtt{X}_R$ at all times between C_R and C. By the second claim above, $\mathtt{X} \leftarrow \mathtt{X}_R$ at C_R. Since X is not finalized at C, it follows from Lemma 25 and Invariant 33 that $\mathtt{X} \leftarrow \mathtt{X}_R$ at all times between C_R and C.

At C, all nodes in $\mathtt{X}_R$'s left subtree are finalized, so $\mathtt{X}_R$ is not of type 2. It follows from the induction hypothesis that $\mathtt{X}_R$ is not frozen in C. Since there is no remove($\mathtt{X}_R$) at $\mathtt{X}_R$ in C, $\mathtt{X}_R$ is unmarked in C.

Since $\mathtt{X}_L <_p \mathtt{X} <_p \mathtt{X}_R$, $rr_{\mathtt{X}}$ calls spliceUnmarkedRight($\mathtt{X}_L$,X,$\mathtt{X}_R$) at line 90. Let $sur_{\mathtt{X}}$ be this call, which must terminate or recurse before C. It does not terminate at line 118, since $\mathtt{X}_R$ is unmarked at C. We have already shown that at all times between the invocation of $rr_{\mathtt{X}}$ and C, $\mathtt{X} \leftarrow \mathtt{X}_R$ and if $\mathtt{X}_L \neq$ null then $\mathtt{X}_L \rightarrow \mathtt{X}$. Thus, $sur_{\mathtt{X}}$ cannot terminate at line 120. So, it performs the CAS at line 123.

To derive a contradiction, assume that this CAS fails. Then some other call to spliceUnmarkedRight installed a Descriptor ($\mathtt{X}'_L$,X',$\mathtt{X}_R$) in $\mathtt{X}_R$->leftDesc between $sur_{\mathtt{X}}$'s executions of line 117 and 123. If $\mathtt{X}' <_c \mathtt{X}$ then there was an earlier time when $\mathtt{X}' \rightarrow \mathtt{X}'_R$ with $\mathtt{X}' <_c \mathtt{X} <_c \mathtt{X}_R$ which means that X was finalized (by Invariant 33), a contradiction. If $\mathtt{X}' = \mathtt{X}$ then the removeRec(X) that installed the Descriptor ($\mathtt{X}'_L$,X',$\mathtt{X}_R$) will ensure that X is finalized before C. Otherwise, $\mathtt{X}' >_c \mathtt{X}$, so $\mathtt{X}' \in S_R$. So, X' is finalized before C_R. Hence, there was a removeRec(X') that performed a successful CAS at line 97, 109 or 123 that caused X' to be finalized before C_R. This contradicts Lemma 34. ◀

Finally, we can present the proof of the overall space bound.

▶ **Theorem 10.** *At the end of any execution by P processes that contains L successful* tryAppend *operations and R* remove *operations on a set of version lists, and a maximum of*

L_{max} successful `tryAppend`*s on a single version list, the total number of lr-reachable nodes across all the version lists in the set is at most* $2(L-R)+O(P\log L_{max})$.

Proof. We divide the lr-reachable nodes into several categories in order to bound the number of them. There are $L-R$ nodes that are not removable. If a removable node `X` is not frozen, then the `remove(X)` operation is still at `X`, so there are at most P such nodes. If a node `X` is finalized but still lr-reachable, then there is a `splice(*,X,*)` that performed line 97, but not line 99, by Lemma 42, so there are at most P such nodes. It remains to bound the number of nodes that are frozen but not finalized. By Lemma 45, there are $O(P\log L)$ such nodes of types 0 and 1. By Lemma 39, the total number of unfinalized nodes of type 2 is at most $L-R+O(P\log L)$. Thus, the number of lr-reachable nodes is at most $2(L-R)+O(P\log L)$. ◀

D Details of Memory Reclamation for Version Lists

D.1 Pseudo-code

As described in Section 6, we must make some adjustments to the list implementation of Section 5 to allow for efficient reclamation of list nodes. The majority of the VersionList methods are implemented the same way as in Figure 4 except with raw pointers replaced by reference-counted pointers. Some methods require more involved changes; only these methods are shown in Figure 7, with the changes shown in blue.

Using the reference counting scheme from [2] involves adding a reference count field to each Node and Descriptor object and replacing raw pointers to Nodes or Descriptors with reference-counted pointers (of type `rc_ptr` and `arc_ptr`), which automatically manage the reference counts of the objects they point to. The `rc_ptr` type supports read, write and dereferencing just like raw pointers, and it is typically used for pointers stored in local memory. The `arc_ptr` (a.k.a. `atomic_rc_ptr` in [2]) type additionally supports CAS, and is typically used for pointers stored in shared memory.

D.2 Time Bounds for Memory-Managed Version Lists

In this section, we prove Theorem 11. We divide the proof into the following pieces.

- Lemma 46 shows that the amortized number of steps for reference counting operations is constant.
- Lemma 47 shows that the changes to `remove` do not affect its constant amortized time bound.
- Lemma 49 shows that the modified `find` operation does not go back to the same node multiple times.

We assume that memory can be allocated and de-allocated in constant time. Since nodes and Descriptor objects have a fixed size, we can make use of recent work on implementing concurrent fixed-sized `alloc` and `free` in constant time [14].

▶ **Lemma 46.** *Read-only operations on* `rc_ptr` *and* `arc_ptr` *(including reading and dereferencing) take O(1) steps. If each object contains* $O(1)$ *reference-counted pointers, the expected amortized number of steps for each update operation (store and CAS) on* `rc_ptr` *and* `arc_ptr` *is* $O(1)$.

Proof. Theorem 1 from [2] states that read-only operations perform $O(1)$ steps, and that the expected number of steps for update operations, ignoring the call to `delete`, is constant.

```
200 class Node {
201   arc_ptr<Node> left, right; // initially null
202   enum status {unmarked,marked,finalized};
203     // initially unmarked
204   int counter; // used to define priority
205   int priority; // defines implicit tree
206   int ts; // timestamp
207   arc_ptr<Descriptor> leftDesc, rightDesc;
208     // initially null
209   int refCount; };

211 class Descriptor {
212   arc_ptr<Node> A, B, C;
213   int refCount; };

215 class VersionList {
216   arc_ptr<Node> Head;

218   rc_ptr<Node> find(rc_ptr<Node> start,
219                     int ts) {
220     rc_ptr<Node> cur = start;
221     while(cur != null && cur->ts > ts) {
222       rc_ptr<Node> next = cur->left;
223       if(next == ⊤) next = cur->right;
224       cur = next; }
225     return cur; } };
```

```
226 bool tryAppend(rc_ptr<Node> B, rc_ptr<Node> C) {
227   if(B != null) {
228     C->counter = B->counter+1;
229     rc_ptr<Node> A = B->left;
230     if(A == ⊤) return false;
231     if(A != null) A->right.CAS(null, B);
232   } else C->counter = 2;
233   C->priority = p(C->counter);
234   C->left = B;
235   if(Head.CAS(B, C)) {
236     if(B != null) B->right.CAS(null, C);
237     return true;
238   } else return false; }

240 bool splice(rc_ptr<Node> A, rc_ptr<Node> B,
241                             rc_ptr<Node> C) {
242   // B cannot be null
243   if(A != null && A->right != B) return false;
244   bool result = CAS(&(B->status), marked, finalized);
245   if(C != null) C->left.CAS(B, A);
246   if(A != null) A->right.CAS(B, C);
247   if(A != null && A->priority > B->priority)
248       B->left = ⊤;
249   if(C != null && C->priority > B->priority)
250       B->right = ⊤;
251   return result; } };
```

Figure 7 Changes to the implementation of version lists to accommodate memory management.

The `delete` operation takes as input a pointer and it frees the object being pointed to. If the object contains any reference counted pointers, the reference count of those pointers are decremented. If these decrements cause any objects' reference counts to hit 0, then delete is recursively called on those objects.

It suffices to argue that the cost of `delete` can be amortized to earlier update operations. When the `delete` operation is called on an object O, for each reference counted pointer *ptr* that belongs to O, `delete` decrements the reference count of the object pointed to by *ptr*. After decrementing the reference counts, the `delete` operation frees the memory occupied by O, which takes constant time. If an object's reference count hits 0 due to one of the earlier decrements, `delete` is recursively called on that object. Since each object contains a constant number of reference counted pointers, the step complexity of a `delete` operation is proportional to the number of decrements performed by the `delete` operation. We can amortize these reference count decrements as follows. Suppose the `delete` operation decrements the reference count of object Y because it was pointed to by field F of object X. We charge the cost of this decrement to the update operation that caused $X.F$ to point to Y. In this way, we guarantee that each update operation is charged with at most one decrement from a `delete` operation. Therefore, each update operation on a reference-counted pointer takes an amortized expected constant number of steps. ◀

The following lemma proves both the correctness of the `remove` algorithm in Section 6 as well as its time bounds. It is convenient to combine these two arguments because they both boil down to arguing that the `remove` algorithm in Section 6 behaves the same way as in Section 5. We show that R remove operations takes $O(R)$ steps plus the cost of $O(R)$ operations on reference-counted pointers.

▶ **Lemma 47.** *Theorem 10 holds for the* `remove` *algorithm in Section 6. Similar to Theorem 9, R* `remove` *operations take $O(R)$ steps plus the cost of $O(R)$ operations on reference-counted*

pointers.

Proof. We begin by comparing Figure 7 with Figure 4 and arguing that the changes made in Figure 7 do not affect the behavior of the original algorithm. The major change is Lines 248 to 250 of `splice`, which set the `left` and `right` pointers of the spliced out node to $\top$ if they point to descendants, as well as the additional if statement on Line 230. This change only adds constant time overhead to `splice` and `tryAppend`, but we need to argue that it does not change the time complexity and correctness of other parts of `remove` or `tryAppend`. To do this, we need to go through every read and CAS on `left` or `right` pointers that could have potentially seen the value $\top$ and argue that the operation would have behaved the same way had it seen the value that was overwritten by $\top$.

We begin by reasoning about CAS operations performed on `left` and `right` fields by the algorithm of Figure 4. It follows from Lemma 23 and Lemma 25 that the CAS steps on line 98 and 99 of `splice` use distinct values for the expected and new value. By Invariant 33.4, `B->left` and `B->right` never change after `splice(*,B,*)` is invoked. Thus, CAS operations on the `left` and `right` pointers of `B` can never succeed. Therefore setting `B`'s `left` or `right` pointer to $\top$ during a `splice(*,B,*)` does not impact the return value of any other CAS operation.

Consider the reads on Lines 68 and 69. We check if `B` is finalized immediately after the two reads. If either of them were $\top$, then the check will fail, so the operation returns without making use of the invalid pointer.

Consider the read of `C->left` on line 120 of `spliceUnmarkedRight(A,B,C)`. If it returns $\top$, then `C` was marked and frozen, so even if the previous value of `C->left` was `B` the `spliceUnmarkedRight` would have returned false anyways after failing the next CAS on Line 123, by Lemma 29. The same argument applies to the read of `A->right` at line 107 of `spliceUnmarkedLeft(A,B,C)`. Now consider a read of `A->right` on line 120 of `spliceUnmarkedRight(A,B,C)` or on line 96 of `splice(A,B,C)`. If it returns `B`, then there cannot have been a call to `splice(*,A,*)` prior to this test, because by Invariant 33.4, it would have to be a `splice(*,A,B)`, which would violate Invariant 33.7. Thus, in the modified algorithm, `A->right` cannot have been changed to $\top$ before the read of `A->right` occurs. So the tests on lines 120, 107 and 96 evaluate to the same value with or without the modification to the `splice` routine. Therefore, the return value and behavior of `splice`, `spliceUnmarkedLeft` and `spliceUnmarkedRight` do not change.

For the read on Line 229, if `B->left` is $\top$, then `B` has already been finalized, so a newer node has already been installed, in this case the `tryAppend` is guaranteed to fail and it will return false on Line 230. Conversely, if `tryAppend` returns false on Line 230, then `B` is no longer at the head of the version list so the CAS on Line 235 is guaranteed to fail.

Therefore, adding Lines 248–250 and 230 to Figure 4 does not affect its behavior and this change just potentially causes some operations to return sooner, so the same arguments for correctness and time bounds apply (Theorems 9 and 10).

The only remaining changes is that some raw pointers in Figure 4 are replaced with reference-counted pointers in Figure 7. This does not affect the behavior (and therefore the correctness) of the algorithm as long as safety is ensured (i.e. a node or Descriptor is not accessed after it is freed). Our algorithm ensures safety because all access to nodes and Descriptors are done through reference counted pointers and these pointers prevent it from being freed. In terms of time bounds, `tryAppend` performs a constant number of reference-counted pointer operations and R `remove` operations perform at most $O(R)$ reference-counted pointer operations, so the lemma holds. ◀

▶ **Definition 48.** *An update to* `cur` *in the* `find` *routine made by following a* `left` *or* `right` *pointer is called a* traversal. *An* upward traversal *is one that follows a link from an finalized node to another node. A* forward traversal *is one that follows a link from a unfinalized node to another node.*

A forward traversal always follows a `left` pointer. We use the term upward traversal because most upward traversals move from a node to an ancestor. However, this is not necessarily the case: if a node has been finalized but its `left` pointer has not yet been set to $\top$, then an upward traversal from the node might go to a descendant.

▶ **Lemma 49.** *The source nodes of upward traversals in a* `find` *operation are all distinct. Similarly, the destination nodes of forward traversals in a* `find` *are all distinct.*

Proof. We first prove the claim about upward traversals. Suppose there is an upward traversal from $\mathtt{X}_0$ to $\mathtt{X}_1$. Let $\mathtt{X}_2, \mathtt{X}_3, \ldots$ be the sequence of nodes visited after $\mathtt{X}_1$. Assume there is another upward traversal after the one from $\mathtt{X}_0$ to $\mathtt{X}_1$, and let the first such upward traversal be from $\mathtt{X}_k$ to $\mathtt{X}_{k+1}$. We argue that $\mathtt{X}_k$ becomes finalized after $\mathtt{X}_0$ becomes finalized by considering two cases.

First, suppose $k = 1$. Consider the time when the `find` traverses from $\mathtt{X}_0$ to $\mathtt{X}_1$. Either $\mathtt{X}_0 \rightarrow \mathtt{X}_1$ or $\mathtt{X}_1 \leftarrow \mathtt{X}_0$. $\mathtt{X}_0$ was finalized by line 97 of a `splice(*,`$\mathtt{X}_0$`,*)`. By Invariant 33.4, that `splice` was of the form `splice(`$\mathtt{X}_2$`,`$\mathtt{X}_0$`,`$\mathtt{X}_1$`)` or `splice(`$\mathtt{X}_1$`,`$\mathtt{X}_0$`,`$\mathtt{X}_2$`)` for some $\mathtt{X}_2$. By Invariant 33, $\mathtt{X}_1$ was unfinalized when that pointer was swung to make $\mathtt{X}_0$ lr-unreachable.

Now, suppose $k > 1$. By definition, the traversal from $\mathtt{X}_{k-1}$ to $\mathtt{X}_k$ is a forward traversal. Thus, at the time the traversal is performed, $\mathtt{X}_{k-1}$, and therefore $\mathtt{X}_k$, is lr-reachable. Thus, $\mathtt{X}_k$ becomes lr-unreachable after $\mathtt{X}_0$.

If we now consider the sequence of source nodes of the upward traversals in the `find`, it follows that each becomes lr-unreachable earlier than the next one in the sequence, so the same node can never appear as the source of two upward traversals.

Next, we prove the claim about forward traversals. Suppose there is a forward traversal f from `Y` to `Z`. To derive a contradiction, assume that at some later configuration, there is another forward traversal f' from `X` to `Z`. (`X` might be equal to `Y`.) Consider the path of nodes $\pi = \mathtt{Z}, \mathtt{U}_1, \mathtt{U}_2, ..., \mathtt{U}_k = \mathtt{X}, \mathtt{Z}$ that the traversal visits between f and f'. When the forward traversal f' from `X` to `Z` occurs, `X` is lr-reachable and therfore `Z` is lr-reachable. Thus, `Z` is lr-reachable when the earlier traversal from `Z` to $\mathtt{U}_1$ occurs since once `Z` becomes lr-unreachable, it cannot become lr-reachable again, by Lemma 41. Therefore, the traversal from `Z` to $\mathtt{U}_1$ is a forward traversal. By Lemma 25, we know that $\mathtt{U}_1 <_c \mathtt{Z}$ because $\mathtt{U}_1$ was a left neighbor of `Z`. By the same lemma, we know that $\mathtt{X} >_c \mathtt{Z}$. Therefore, there are two consecutive nodes, `A` and `B` in π such that $\mathtt{A} \leq_c \mathtt{Z}$ and $\mathtt{B} >_c \mathtt{Z}$. By Lemma 25, the traversal from `A` to `B` must follow a `right` pointer and is therefore an upward traversal. $\mathtt{A} \neq \mathtt{Z}$ because only forward traversals can be performed from `Z` and these always go left. Therefore, $\mathtt{A} <_c \mathtt{Z}$. Since π performs an upward traversal from `A` to `B`, `A` must have been lr-unreachable (and therefore finalized) at the time of the traversal and $\mathtt{A} \rightarrow \mathtt{B}$ so by Invariant 33.5a, all nodes `D` such that $\mathtt{A} \leq_c \mathtt{D} \leq_c \mathtt{B}$ are unreachable. That would imply that `Z` is unreachable, which is a contradiction. Therefore, destination nodes of forward traversals are always distinct. ◀

▶ **Lemma 50.** *If* `find` *performs an upward traversal from* `B` *to* `C`*, then* `C` *is a proper ancestor of* `B`.

Proof. If `find` performs an upward traversal from `B` to `C`, then `B->left` is $\top$. So, `B` must have already been finalized by some `splice(A,B,C')` operation. By Invariant 33.4, $\mathtt{B} \rightarrow \mathtt{C}'$ at the time of the upward traversal from `B` to `C`, so $\mathtt{C} = \mathtt{C}'$. To complete the proof, we argue

that C is a proper ancestor of B. Since B->left was set to $\top$, we know that A $\geq_p$ B so by Lemma 35, we know that B $\geq_p$ C. By Lemma 36, B->right is either a proper ancestor or a proper descendant of B, so C is a proper ancestor of B. ◀

▶ **Lemma 51.** *Suppose* A *and* B *are two nodes in a list such that* A $<_c$ B*. If* C *is an ancestor of* A *and* C $>_c$ B *then* C *is also an ancestor of* B*.*

Proof. Let C be an ancestor of A such that C $>_c$ B. Our goal is to show that C is also an ancestor of B. Let D be the lowest common ancestor of A and B. Either A = D or A is in the left subtree of D. Similarly, either B = D or B is in the right subtree of D. So, each node E along the path from A to D satisfies E $\leq_c$ D $\leq_c$ B. Since C $>_c$ B, C is not one of the nodes on the path from A to D, so C is a proper ancestor of D, and therefore of B. ◀

▶ **Lemma 52.** *If the traversal of a* find *operation starts at node* V*, then the nodes visited by the traversal to the right of* V *are ancestors of* V*.*

Proof. Let $X_1, X_2, \ldots$ be the sequence of distinct nodes visited whose counter values are greater than V, in the order in which they were first visited. We proceed by induction on this set of nodes. Consider the node X_1. The traversal must have arrived at this node by leaving a node U, where U $\leq_c$ V. By Lemma 25 this traversal followed U->right, so it must have been an upward traversal. By Lemma 50, X_1 is an ancestor of U. By Lemma 51, X_1 must also be an ancestor of V. For the inductive step, suppose X_i is an ancestor of V for all $i < j$. It suffices to show that X_j is also an ancestor of V. There are two ways the traversal could have arrived at X_j: (1) from some node X_i where $i < j$, (2) from some node U with U $\leq_C$ V. Case (2) is the same as the base case, so we consider case (1). Since an upward traversal was performed from X_i to X_j, by Lemma 50, X_j must be an ancestor of X_i. Since X_i is an ancestor of V, X_j is also an ancestor of V. Therefore, $X_1, X_2, \ldots$ are all ancestors of V. ◀

▶ **Lemma 53.** *If the traversal of a* find *operation starts at node* V*, the number of distinct nodes visited to the right of* V *is bounded above by both the depth of* V *in the implicit tree and* $O(\log c)$*, where* c *is the number of successful* tryAppend *operations from the time* V *was the head of the list and the end of the* find *(on the same object).*

Proof. Let $X_1, X_2, ..., X_k$ be the k distinct nodes visited by the find operation whose counter values are larger than V's, sorted so that $X_1 <_p X_2 <_p \cdots <_p X_k$. Each of these k nodes must be an ancestor of V, so k is at most the depth of V. To prove the second part of the claim, it suffices show that c is at least 2^{k-1}. Each node X_i was added by a tryAppend between the time V was the head of the list and the end of the find. Since V $<_c$ X_i, V must be in the left subtree of X_i for all i. Thus, all of $X_2, \ldots, X_k$ are in the left subtree of X_1. This subtree (like all left subtrees) is a complete binary tree. Hence, X_2 is the root of a complete binary tree that contains all the nodes $X_2, X_3, ..., X_k$, so its height is at least $k - 1$. This means that the right subtree of X_2 has 2^{k-1} nodes and all of these nodes have counter values greater than V, which means that $c \geq 2^{k-1}$. ◀

▶ **Theorem 11.** *The amortized expected time complexity of* tryAppend*,* getHead*,* remove*, and creating a new version list is* $O(1)$*. The amortized expected time complexity of* find(V, ts) *is* $O(n + \min(d, \log c))$*, where* n *is the number of version nodes with timestamp greater than* ts *that are reachable from* V *by following* left *pointers (measured at the start of the* find*),* d *is the depth of the VNode* V *in the implicit tree and* c *is the number of successful* tryAppend *from the time* V *was the list head until the end of the* find*. All operations are wait-free.*

Proof. Since `tryAppend` and `getHead` have no loops, they perform $O(1)$ steps, plus $O(1)$ operations on reference-counted pointers. By Lemma 47, we know that `remove` takes amortized $O(1)$ time plus amortized $O(1)$ operations on reference counted pointers.

Next, we show that the number of steps and reference-counted pointer operations performed by a `find` is $O(n + \min(d, \log c))$. It suffices to show that the number of distinct nodes visited by the `find` operation is $O(n + \min(d, \log c))$ because it follows from Lemma 49 that the number of upward and forward traversals is at most the number of distinct nodes visited. Each forward and upward traversal contributes $O(1)$ steps plus the cost of $O(1)$ reference counted pointer operations.

By Lemma 53, we know that the number of distinct nodes visited whose counter values are greater than `V`'s is at most $O(\min(d, \log c))$. Let S be the set of nodes visited whose counter values are less than `V`'s. Next, we argue that $|S|$ is at most $O(n)$. This is because whenever we do a forward or upward traversal, we know that the timestamp of the source node is larger than the timestamp of the `find` operation. Thus, all nodes in S have timestamp greater than `ts` except one. All nodes in S must have also been reachable from `V` at the start of the `find` operation. Therefore, there are at most $O(n)$ nodes in S.

The final step is to argue that the reference counting operation performed by the version list operations all take $O(1)$ amortized expected time. This is a little tricky to show because there could be operations on reference counted pointers outside of the version list operations. However, if we example the proof of Lemma 46, no decrement operations will ever be charged to operations on reference counted pointers outside of the version list operations. Therefore, the amortized expected time complexity of `tryAppend`, `getHead`, `remove`, and creating a new version list is $O(1)$, and the amortized expected time complexity of `find(V, ts)` is $O(n + \min(d, \log c))$. ◀

D.3 Space Bounds for Memory-Managed Version Lists

The following observation follows from the fact that the RC scheme of [2] has $O(P^2)$ delayed decrements, as discussed in Section 6.

▶ **Observation 54.** *If there are a total of $O(K)$ reference-counted pointers to a particular type of object, then the number of objects of that type that have been allocated, but not freed is $O(K + P^2)$.*

▶ **Theorem 12.** *Assuming there are at most K reference-counted pointers to VNodes from the application code, at the end of any execution that contains L successful* `tryAppend` *operations, R* `remove` *operations and a maximum of L_{max} successful* `tryAppend`*s on a single version list, the number of VNodes and Descriptors that have been allocated but not reclaimed is $O((L - R) + (P^2 + K) \log L_{max})$.*

Proof. Let T be the total number of lr-reachable nodes across all the version lists. By Lemma 47, we know that Theorem 10 holds for the modified version list algorithm in Section 6, so $T \in O((L - R) + P \log L)$. To prove this theorem, it suffices to show that the number of VNodes and Descriptors is at most $O(T + (P^2 + K) \log L)$. We begin by defining some useful terminology. An *up-pointer* is a `left` or `right` pointer from a VNode `A` to another VNode `B` such that `B` is an ancestor of `A` in the implicit tree. Similarly, a *down-pointer* is a pointer from `A` to `B` such that `B` is a descendant of `A` in the implicit tree. By Lemma 36, every pointer between two active VNodes is either an up-pointer or a down-pointer. A *source node* is an lr-unreachable node that is not pointed to by any up-pointers.

We begin by proving the claim that all lr-unreachable nodes are reachable by following up-pointers starting from some source node. Starting from an lr-unreachable node `A`, consider

the following process: if `A` is a source node, then terminate; otherwise there exists an lr-unreachable node `B` with an up-pointer to `A` (`B` cannot be lr-reachable because that would mean `A` is lr-reachable as well), so we repeat this process starting from `B`. This process is guaranteed to terminate because up-pointers do not form a cycle, and this proves the claim.

Next, we show that there are at most $O(T + P^2)$ Descriptors that have been allocated but not freed. This is because there are at most $O(P)$ Descriptors pointed to by local pointers (these pointers are temporarily held by a process when it is in the middle of a version list operation), and $O(T)$ Descriptors pointed to by lr-reachable nodes. Nodes that are lr-unreachable do not point to any non-`frozen` Descriptor objects. This is because a VNode cannot point to any Descriptor object until it is active, and if a node is active and lr-unreachable, then it must have been finalized, by Lemma 40, so its descriptor fields have already been frozen, by Lemma 30. Therefore, there are at most $O(T + P)$ pointers to Descriptor objects, and by Observation 54, there are at most $O(T + P^2)$ Descriptor objects that have been allocated, but not freed.

It remains to show that there are $O(T + (P^2 + K) \log L)$ VNodes and we prove this by bounding the number of lr-unreachable VNodes by the same amount.

Next, we show that there are $O(T + P^2 + K)$ source nodes. This is because source nodes are either pointed to by (1) Descriptor objects, (2) a local pointer held during version list operations, (3) down-pointers or (4) user pointers. There are $O(T + P^2)$ pointers of type (1), $O(P)$ of type (2) and (3), and $O(K)$ of type (4). To show the bounds for type (3), there are only two ways for a lr-unreachable node to be pointed to by a down-pointer. First, it could be that the node pointing to it has been spliced out and its `left` or `right` pointer has not yet been set to $\top$. Second, it could be that the node pointing to it was newly allocated for a `tryAppend` that is now guaranteed to fail. There can only be a constant number of such cases per process, which proves the bound. The `Head` pointer of a version list can never point to a source node, since we set the `Head` pointer to `null` before calling `remove` on the head of a version list, which ensures `Head` always points to an lr-reachable node. Therefore, there is a total of $O(T + P^2 + K)$ reference-counted pointers to source nodes, which results in $O(T + P^2 + K)$ source nodes that have been allocated and not freed, by Observation 54.

From each source node, there are at most $O(\log L)$ nodes reachable by following up-pointers. This is because all the reachable nodes are ancestors of the source node, and each node can only have $O(\log L)$ ancestors. However, some of these $O(\log L)$ nodes are lr-reachable and we only want to count the lr-unreachable ones. It turns out that whenever a source node `C` is pointed to by a Descriptor installed in a VNode, the left and right neighbors of `C` are both lr-reachable, so all of the $O(\log L)$ nodes reachable from `C` by following up-pointers are lr-reachable. There are $O(T)$ source nodes that are pointed to by a Descriptor installed in a VNode and $O(P^2 + K)$ sources nodes that are not. For each source node in the latter category, there could be $O(\log L)$ lr-unreachable nodes that are reachable from that node by following up-pointers. Therefore, the number of lr-unreachable nodes is bounded by $O(T + (P^2 + K) \log L)$. ◀

E Details of Snapshottable Data Structure Application

E.1 Pseudo-code for Memory-Managed VersionedCAS Objects

Wei et al.'s implementation [51] of VersionedCAS objects appears as the black text of Figure 8. Additions we make to the algorithm for memory management are shown in blue. A `takeSnapshot` operation attempts to increment the counter of the Camera object and returns the old value of the counter as a snapshot handle. Each vnode contains a value

```
1  class Camera {
2    int timestamp;
3    RangeTracker<VNode*> rt;
4    Camera() { timestamp = 0; }

6    int takeSnapshot() {
7      int ts = rt.Announce(&timestamp);
8      CAS(&timestamp, ts, ts+1);
9      return ts; }

11   void unreserve() {
12     rt.unannounce(); } };

14 class VNode {
15   Value val; int ts;
16   arc_ptr<VNode> left, right;
17   // the remaining variables are only
18   // used by remove() and tryAppend()
19   int priority; int counter;
20   enum status; int refCount;
21   VNode(Value v) {
22     val = v; ts = TBD; } };

24 class VersionedCAS {
25   VersionList VList;
26   Camera* C;
27   VersionedCAS(Value v, Camera* c) {
28     C = c;
29     rc_ptr<VNode> node = new VNode(v);
30     VList.tryAppend(null, node);
31     initTS(node); }
```

```
32 void initTS(rc_ptr<VNode> n) {
33   if(n->ts == TBD) {
34     int curTS = C->timestamp;
35     CAS(&(n->ts), TBD, curTS); } }

37 Value readVersion(int ts) {
38   rc_ptr<VNode> node = VList.getHead();
39   initTS(node);
40   node = VList.find(node, ts);
41   return node->val; } // node cannot be null

43 Value vRead() {
44   rc_ptr<VNode> head = VList.getHead();
45   initTS(head);
46   return head->val; }

48 bool vCAS(Value oldV, Value newV) {
49   rc_ptr<VNode> head = VList.getHead();
50   initTS(head);
51   if(head->val != oldV) return false;
52   if(newV == oldV) return true;
53   rc_ptr<VNode> newN = new VNode(newV);
54   if(VList.tryAppend(head, newN)) {
55     initTS(newN);
56     List<VNode*> redundant = C.rt.deprecate(
57     toRawPtr(head), head.ts, newN.ts);
58     forevery(node in redundant) {
59       // convert from raw pointer to rc_ptr
60       VersionList.remove(rc_ptr<VNode>(node)); }
61     return true; }
62   else {
63     delete newN;
64     initTS(VList.getHead());
65     return false; } } };
```

Figure 8 Indirect versioned CAS implementation (black text) using the range-tracking and version list maintenance interface (blue text). Functions `announce`, `unannounce` and `deprecate` are defined in the range-tracking object (see Section 4). Functions `tryAppend` and `remove` are defined in the version list maintenance interface (see Section 5).

that has been stored in the VersionedCAS object, a pointer to the next older vnode and a timestamp value read from the associated Camera object. A `vRead` obtains the current value of the VersionedCAS object from the head of the list. A `vCAS`(old, new) operation checks that the head of the list represents the value *old* and, if so, attempts to add a new vnode to the head of the list to represent the value *new*. A `versionedRead` operation using a timestamp s returned by a `takeSnapshot` traverses the version list until it finds a vnode whose timestamp is less than or equal to s, and then returns the value stored in that vnode. To ensure linearizability, a number of helping mechanisms are incorporated into the implementation. See [51] for details.

To facilitate garbage collection, we add an `unreserve` operation to the Camera object, which a process can use to release a particular snapshot handle, guaranteeing that the process will not subsequently use it as an argument to `readVersion`.

E.2 Proof of time bounds

▶ **Observation 55.**
An execution with M calls to `vCAS` *contains M calls to* `deprecate` *and at most M calls to* `remove`.

▶ **Theorem 14.** *A snapshot query takes amortized expected time proportional to its sequential complexity plus the number of* `vCAS` *instructions concurrent with it. The amortized expected time complexity of frontier operations is the same as in the non-snapshottable DS.*

Proof. Each snapshot query begins with a `Camera.takeSnapshot` returning some timestamp `ts`, ends with a `Camera.unreserve` and performs some `readVersion(ts)` operations in between. By inspection of the pseudo-code and by Theorem 4, we can see that the `takeSnapshot` and `unreserve` operations take $O(1)$. If each `readVersion` operation took $O(1)$ time as well, then time complexity of the snapshot query would be the same as its sequential complexity. Unfortunately, `readVersion` does not take constant time because it calls `find`. We will prove the following claim: a `readVersion(ts)` on VersionedCAS object O has amortized expected time complexity at most the number of successful `vCAS` operations on object O concurrent with the snapshot query. Assuming that the snapshot query calls `readVersion` on each object at most once (which is true in [51] because values returned by `readVersion`s are cached so there's no need to read the same object twice), this means that the amortized expected time complexity of the `readVersion` is proportional to its sequential complexity plus the number of successful `vCAS` operations it is concurrent with. Therefore, to prove the desired time bound for `readVersion`, it suffices to prove the previous claim.

To prove the claim, we first note that a `readVersion(ts)` operation starts by performing `getHead` and this takes amortized expected constant time by Theorem 11. Then it calls `initTS` on V (which also takes constant time) followed by a `find(*, ts)`. We know that all nodes visited by the `find(*, ts)` operation, except for the last one, have timestamp greater than `ts`. The value of the global timestamp at the start of `readVersion` is less than or equal to `ts`, so the vCAS operations that added these nodes must have been linearized after the start of the `readVersion`. Therefore, the number of distinct nodes visited by the `find` operation is bounded by one plus the number of successful vCAS operations on the same object that were concurrent with the `readVersion`. By Lemma 49, we know that the total number of forward and upward traversals performed by a `find` is proportional the number of distinct nodes visited. Each forward or upward traversal takes amortized expected constant time, so the `readVersion` has amortized expected time complexity $O(C)$ where C is the number of `vCAS` operations on the same object that are concurrent with the snapshot query. Thus the claim holds.

Finally, the amortized expected time complexity of frontier operations remains unchanged because both `vRead` and `vCAS` take amortized expected constant time. ◀

E.3 Proof of space bounds

As in Section 7, we consider a DNode to be *necessary* if it is either not freed by the original data structure's memory reclamation scheme or if there exists an announced timestamp in between its birth (inclusive) and retire (exclusive) timestamp. We say that a VNode is *necessary* if it is pointed to by a DNode that has not yet been deprecated (i.e., freed by the original data structure's memory reclamation scheme) or if it has been deprecated but the interval it was assigned contains an announced timestamp. Let D_{max} and V_{max} be the maximum number of necessary DNodes and VNodes at any one time during an execution.

▶ **Theorem 56.** *In any configuration, there are $O(D_{max} + P^2 \log P)$ DNodes that have been allocated but not freed.*

Proof. We begin by noting that a DNode is deprecated as soon as it is freed by the original data structure's memory reclamation scheme. The set of DNodes that have been allocated

but not freed can be partitioned into three sets, one set A for DNodes that have not been freed by the original data structure's memory reclamation scheme, another set B for DNodes that have been passed to `deprecate` but not returned, and another set C for DNodes that have been returned by some `deprecate` but not freed. Since any DNode that is not yet deprecated is necessary, $|A| < D_{max}$. The nodes returned by `deprecate` are freed before the next call to `deprecate` by the same process, so by Lemma 19, each process contributes at most $4P \log P$ nodes to the set C, so $|C| \leq 4P^2 \log P$.

To complete the proof, it remains to bound $|B|$. Recall that a deprecated object is said to be *needed* if its interval contains an active announcement. This means that all needed deprecated VNodes are also necessary. Therefore D_{max} is an upper bound on the number of needed deprecated VNodes across all configurations of the execution. By Theorem 5, $|B| \leq 2D_{max} + 25P^2 \log P$, which completes the proof. ◀

Assuming each DNode is constant-sized, we can prove the following theorem.

▶ **Theorem 13.** *Assuming each VNode and DNode takes $O(1)$ space, the overall space usage of our memory-managed snapshottable data structure is $O(D + V + P^2 \log P + P^2 \log L_{max})$, where L_{max} is the maximum number of successful* `vCAS` *operations on a single* `VCAS` *object.*

Proof. Theorem 56 bounds the total number of DNodes by $O(D_{max} + P^2 \log P)$. To bound the number of VNode and Descriptor objects, we make use of Theorem 12. In Figure 8, we can see that each process holds on to a constant number of VNode pointers so we set the parameter K in Theorem 12 to $O(P)$. To show that there are not too many VNodes and Descriptors, we need to bound the number of successful `tryAppend`s (L) minus the number of `remove`s (R) by $O(D_{max} + V_{max} + P^2 \log P)$. This is equivalent to bounding the number of VNodes that have been appended, but not passed to remove. We divide these VNodes into disjoint sets and show that there are:

1. $O(D_{max})$ VNodes that have been appended and are still at the head of the version list,
2. $O(P)$ VNodes that have been replaced as the head of the version list but are not yet deprecated,
3. $O(V_{max} + P^2 \log P)$ VNodes that have been deprecated, but not yet returned, and
4. $O(P^2 \log P)$ VNodes that have been returned, but not yet passed to remove.

The first bound holds because each DNode contains a constant number of versionedCAS objects, which are essentially pointers to heads of version lists. The second bound holds because a VNode is deprecated immediately after a new head is appended and there can only be P processes stalled in between these two steps. The third bound holds because every needed deprecated VNode is also a necessary VNode, so V_{max} is an upper bound on H_{max} from Theorem 5. Therefore, by Theorem 5, at most $O(V_{max} + P^2 \log P)$ VNodes have been deprecated, but not yet returned. The final bound holds because each call to `deprecate` returns at most $4P \log P$ VNodes (Lemma 19) and VNodes are passed to remove as soon as they are returned by some `deprecate`.

Summing up over all the cases, we see that $(L - R) \in O(D_{max} + V_{max} + P^2 \log P)$. By Theorem 12, the number of VNodes and Descriptors that have been allocated and not freed is at most $O(D_{max} + V_{max} + P^2 \log P + P^2 \log L)$. Since each DNode, VNode and Descriptor takes a constant amount of space, the overall space usage of these objects is $O(D_{max} + V_{max} + P^2 \log P + P^2 \log L)$. ◀