

Built-in Coloring for Highly-Concurrent Doubly-Linked Lists*

Hagit Attiya and Eshcar Hillel
Department of Computer Science
Technion

October 17, 2010

Abstract

This paper presents a novel approach for lock-free implementations of highly-concurrent doubly-linked lists, based on dynamically maintaining a *coloring* of the nodes in the list. Highly-concurrent implementations allow operations to proceed without interfering with each other, if they are applied to non-adjacent nodes in the list. Roughly speaking, the operations are implemented by acquiring locks on nodes in the operation's data set and then making the changes atomically; locks are acquired according to a coloring order that decreases the length of waiting chains and increases concurrency. The legality of the coloring is, in turn, preserved by having operations correctly update the colors of the nodes they modify.

We use this approach in two new algorithms: **CAS-Chromo** uses an unary conditional primitive, **CAS**, and allows insertions anywhere in the linked list and removals only at the ends, while **DCAS-Chromo** allows insertions and removals anywhere but uses a stronger primitive, **DCAS**. Both algorithms ensure progress in close neighborhoods of processes that keep taking steps, providing high throughput.

Keywords: local nonblocking, nonblocking implementations, CAS, DCAS, doubly-linked list, double-ended queue, priority queue.

*This research was supported by the *Israel Science Foundation* (grant number 953/06). A preliminary version of this paper has appeared in the proceedings of the 20th International Symposium on Distributed Computing (DISC'06), pages 31–45.

1 Introduction

Many core problems in asynchronous multiprocessing systems revolve around the coordination of access to shared resources and can be captured as *concurrent data structures*—abstract data structures that are concurrently accessed by asynchronous processes. A prominent example is provided by list-based data structures: A *double-ended queue* (*deque*) supports operations that insert and remove nodes at the two ends of the queue; it can be used as a producer-consumer job queue [3]. A *priority queue* can be implemented as a doubly-linked list where removals are allowed only at the ends, while nodes can be inserted anywhere at the queue; it can be used to queue process identifiers for scheduling purposes. Finally, a generic *doubly-linked list* (hereafter, called simply a *linked list*) allows insertions and removals of nodes anywhere in the linked list.

Concurrent data structures are implemented by applying *primitives*—provided by the hardware or the operating system—to memory locations. The implementation may use locks, but this causes problems like deadlock, convoying, and priority-inversion. Alternatively, primitives like CAS (*compare and swap*) and its multi-location variant, *kCAS*, are used. In both cases, however, it is hard to get these implementations right; even for relatively simple, key data structures, like deques, implementations make significant compromises: Some implementations may contain garbage nodes [17], others statically limit the data structure’s size [19] or do not allow concurrent operations on both ends of the queue [22]. Even when DCAS (i.e., 2CAS) is used, existing implementations either are inherently sequential [14, 15] or allow access to chains of garbage nodes [12].

Implementing concurrent data structures is simpler if an arbitrary number of locations can be accessed atomically. For example, removing a node from a doubly-linked list is easy if one can atomically access three nodes—the node to be removed and the two nodes before and after it (cf. [12]). Known methods, such as [7, 25, 28], simulate this atomicity in software by using CAS to acquire locks on the nodes—one lock at a time. In these methods, the memory addresses of the nodes serve as their identifiers. The order in which locks are acquired is then determined by these identifiers. Unfortunately, in symmetric scenarios the resulting implementations may have long waiting chains, creating interference among operations and reducing the implementation’s throughput. Other methods avoid long waiting chains and improve the locality by using colors [1, 4], randomization [16, 24] or DCAS [5] to break the symmetry (examples appear in Section 7).

This paper presents an approach for improving the locality of concurrent linked-list implementations, by exploiting the fact that operations on the linked-list access its constituent nodes in a predictable, well-organized manner, i.e., two or three consecutive nodes in the list. In this case, the operations can determine the locking order by identifiers that are not the addresses of the nodes. If a *small set of colors* is built into the nodes, then nodes can be locked by the order of colors. To avoid deadlocks while ensuring short waiting chains, the operation guarantees that the modifications it applies to the data structure preserve the legality of the nodes’ coloring; this is possible since the implementation initializes the data structure and provides operations that are the only means for manipulating it.

Our implementations are *local nonblocking*, namely, they ensure that when operations access distant parts of the data structure, or are separated in time, they do not interfere with each other. Formalizing this notion relies on defining the *distance* between operations to be the length of the shortest path between them in the *conflict graph* (Section 3). In *d-local nonblocking* implementations, whenever an operation *op* takes an infinite number of steps, some operation, within distance *d* from *op*, completes. This guarantees progress in components of conflicting operations of diameter *d*, and ensures that operations are effectively *isolated* from other operations at distance $> d$.

Our first algorithm, *CAS-Chromo*, allows removals only at the ends of the linked list and uses CAS; it can be used as a simple deque or a priority queue. Our second algorithm, *DCAS-Chromo* implements a linked list that allows insertions and removals anywhere in the list. The algorithms we present are conceptually simple: A 3-coloring of the nodes is used by operations to determine the order in which locks are acquired. After acquiring locks, the operations apply their changes in isolation. In our algorithms, equally colored nodes are locked by their list order (from left to right) to ensure that there are no deadlocks and that some operation makes progress at any time.

Since the list is 3-colored, we can show that *CAS-Chromo* is *5-local nonblocking*, namely, an operation is delayed only due to operations on nodes close to its own nodes on the linked list. When insertions are limited to occur at the ends (i.e., a deque), the analysis can be further refined to show that the algorithm is 3-local nonblocking; this means that operations at the two ends of a deque containing at least three nodes do not interfere with each other.

DCAS-Chromo allows removals also from the middle of the linked list, which is more difficult: a remove operation locks three consecutive nodes, and in a legal coloring, two of these nodes may have the same color. Thus, removing a node might entail recoloring one of its neighbors while ensuring its neighbor's color is not changed concurrently. To do this without creating hold-and-wait chains we employ DCAS to *atomically lock two nodes with the same color*. *DCAS-Chromo* is *4-local nonblocking*, namely, it is an implementation of a linked list that allows insertions and removals anywhere in the list, and guarantees that only operations on nodes that are separated by less than three nodes may interfere with each other.

The rest of this paper is organized as follows. Section 2 presents the model of an asynchronous shared-memory system, while Section 3 defines the locality properties in a dynamic setting. *CAS-Chromo*, a CAS-based implementation of a priority queue allowing insertions anywhere and removals only from the end appears in Section 4.1. Section 4.2 outlines the modifications needed to obtain *DCAS-Chromo*, a DCAS-based implementation of a linked list supporting removals anywhere. The correctness proof of both algorithms is given in Section 5, while the proofs of their locality properties appear in Section 6. Finally, we discuss related work in Section 7 and conclude, in Section 8.

2 Preliminaries

We consider a standard model for a shared memory system [6] in which a finite set of *asynchronous processes* p_1, \dots, p_n communicate by applying *primitive* operations to shared *memory locations*, l_1, \dots, l_m . A *configuration* specifies the local state of each process and the value of each memory location. In the (unique) *initial configuration*, every process is in its initial state and every location contains its initial value.

An *event* is a computation step by a process consisting of some local computation and the application of a primitive to the memory. We employ the following primitives: $\text{READ}(l_j)$ returns the value v_j in location l_j ; $\text{WRITE}(l_j, v)$ sets the value of location l_j to v ; $\text{CAS}(l_j, \text{exp}, \text{new})$ writes the value new to location l_j if its value is equal to exp , and returns a success or failure indication; DCAS is similar to CAS, but operates on two independent memory locations (see Figure 1).

An *execution interval* α is a (finite or infinite) alternating sequence $C_0, \phi_0, C_1, \phi_1, C_2, \dots$, where C_k is a configuration, ϕ_k is an event and the application of ϕ_k to C_k results in C_{k+1} , for every $k = 0, 1, \dots$. An *execution* is an execution interval in which C_0 is the unique initial configuration.

A *data structure type* supports a set of operations that provide the only means to manipulate it. The sequential specification of an *operation* indicates how the data structure is modified when

operations are applied in a serial manner (in isolation). It is given as a sequence of read and write instructions executed on a set of items, which constitute the operation’s *data set*.

An *implementation* of a data structure type provides a specific data-representation for its instances as a set of memory locations, and protocols that processes must follow to carry out its operations, defined in terms of primitives applied to memory locations.

This paper considers a *doubly-linked list* data structure; the items are the *nodes* composing the linked list. Each node has links to its left and right neighboring nodes. Two special *anchor* nodes serve as the leftmost and rightmost nodes in the doubly-linked list, denoted *LA* and *RA*; they cannot be removed from it, and hold no left link or no right link, respectively. A node is *valid* if it is either an anchor, or both its left link and right link pointers are not null.

We concentrate on the following operations: *PushLeft*, *PopLeft* operations (and their counterparts *PushRight*, *PopRight*), which inserts a node or removes the node next to the left (right) anchor respectively; *InsertRight* operation (and its equivalent *InsertLeft*) and *Remove* operation applied to some *source* node in the linked list, which inserts a node to the left of, or removes, the source node. The sequential specification of these operations, following the description of the deque operations in [2], are described in Figure 2; missing specifications are symmetric.

In order to apply an operation, process p_i executes the protocol associated with the operation.

Two executions are *equivalent* if every process in these executions issues the same operations in the same order and gets the same result for each operation.

The *interval of an operation* op is the execution interval that starts at the first event of op and ends at the last event of op , if there is one; if the operation does not terminate, its interval is infinite. Two operations *overlap* if their intervals overlap.

An execution is *serial* if operations do not overlap; this means that every operation is executed to completion before another operation starts.

We require the implementation to be *linearizable* [20], that is, any execution can be extended by discarding some pending operations and completing the others, such that the extended execution is equivalent to some serial execution, called its *linearization*, which preserves the order of non-overlapping operations; there might be more than one linearization for an execution.

3 Locality Properties

The notion of a data set is crucial for defining the locality properties. Formalizing this notion requires care since the data structure—and hence the data sets of its operations—are dynamic.

Figure 3 shows several overlapping operations and their data sets in a specific configuration: op_1 is a *PushLeft* operation, op_2 is a *PopLeft* operation, op_5 is a *PushRight* operation, op_3 inserts a

<pre> boolean CAS(<i>l</i>, <i>exp</i>, <i>new</i>) { // Atomically if <i>l</i> = <i>exp</i> then <i>l</i> ← <i>new</i> return TRUE return FALSE } </pre>	<pre> boolean DCAS(<i>l</i>₁, <i>l</i>₂, <i>e</i>₁, <i>e</i>₂, <i>n</i>₁, <i>n</i>₂) { // Atomically if <i>l</i>₁ = <i>e</i>₁ and <i>l</i>₂ = <i>e</i>₂ then <i>l</i>₁ ← <i>n</i>₁ <i>l</i>₂ ← <i>n</i>₂ return TRUE return FALSE } </pre>
--	--

Figure 1: The CAS and DCAS primitives.

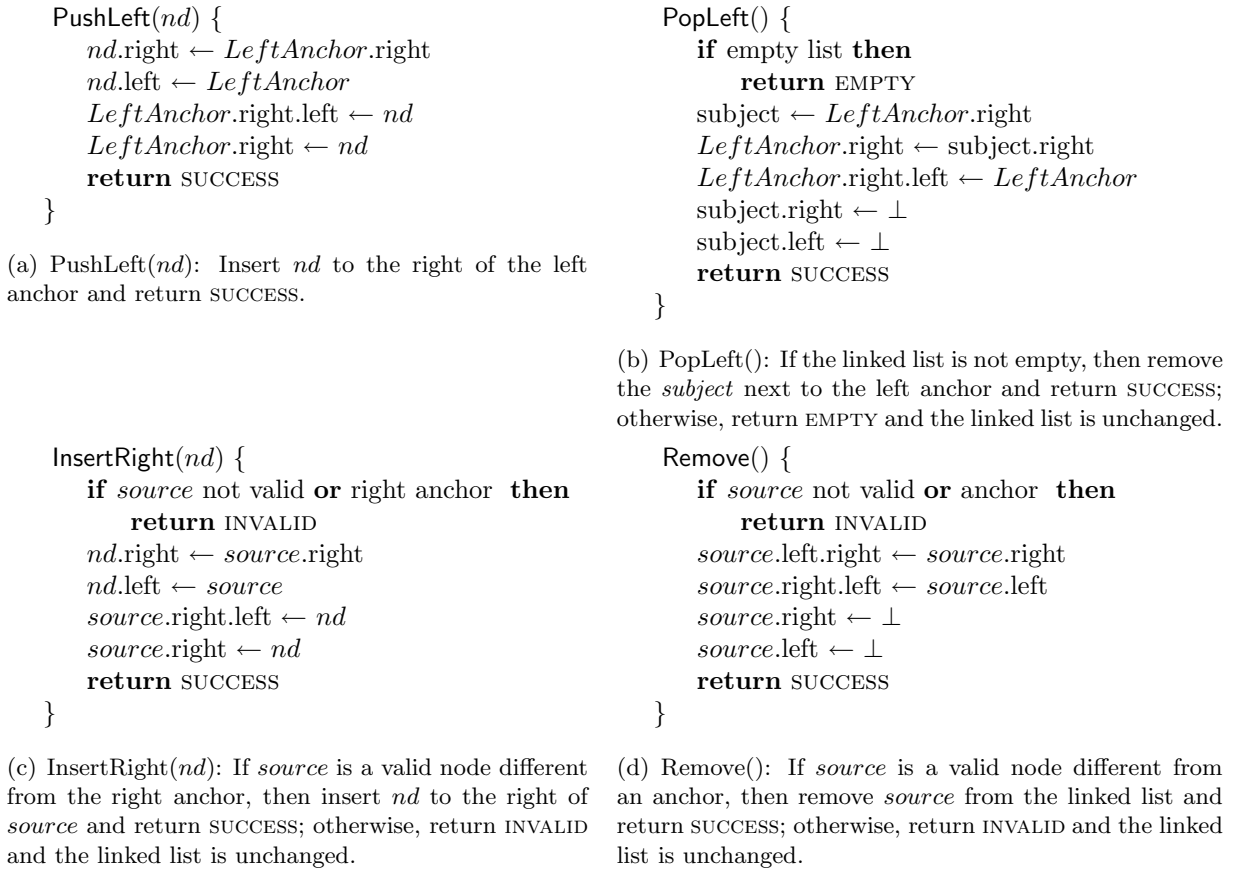


Figure 2: Sequential specification of the linked list operations.

new node to the right of m_2 , and op_4 removes m_4 . Consider, for example, the following execution $\alpha_1\alpha_2\alpha_3$ of these operations: in the execution interval α_1 every operation takes steps, but makes no changes to the memory; then, in the execution interval α_2 only op_2 takes steps until it successfully removes m_1 ; finally, in the execution interval α_3 only op_1 takes steps until it completes. Denote by C_1 , C_2 , and C_3 the configurations at the end of the execution intervals α_1 , α_2 , and α_3 , respectively. Figure 3 shows the linked list at configuration C_1 .

The *set of states* of the linked list after an execution α contains every state of the linked list that is reachable after some linearization of α .

Definition 1 *If ST is the set of states of the linked list at a configuration C , then the data set of an operation at C is the union, over every state s in ST , of the set of items accessed by the operation when executed from s . The data set of an operation is the union of the data sets in all the configurations during its execution.*

The *conflict graph* captures the distance between overlapping operations: Edges represent direct conflicts between two concurrent operations, while paths represent the transitive closure of these conflicts. More precisely, the *conflict graph* of a configuration C is an undirected graph in which vertices represent operations, and an edge connects two operations whose data sets intersect. The conflict graph of an execution interval α is the union of the conflict graphs of all configurations C

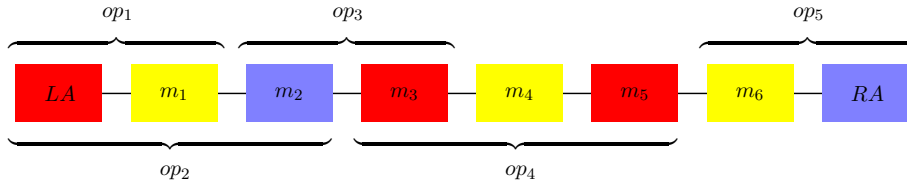


Figure 3: Overlapping operations of a doubly-linked list, LA and RA are left and right anchors respectively.

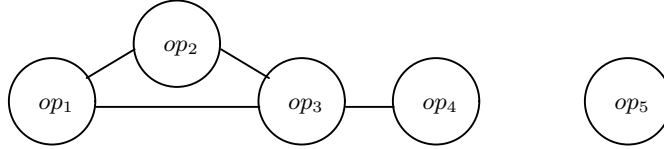


Figure 4: The conflict graph of the execution interval $\alpha_1\alpha_2\alpha_3$ of the operations from Figure 3.

in α ; that is, the vertices (edges, respectively) in the graph are the union of the vertices (edges, respectively) of all these conflict graphs.

The *distance* between two operations, $op \neq op'$, in a conflict graph, is the length (in edges) of the shortest path between op and op' . The *distance* from an operation to itself is zero. The distance between two operations is one if their data sets intersect; if there is no path between the operations, the distance is infinity.

For example, Figure 4 depicts the conflict graph of the interval of the operation op_1 from Figure 3, $\alpha_1\alpha_2\alpha_3$. Note that the data set of op_1 changes over time: in C_1 the data set is $\{LA, m_1\}$, while after m_1 is removed in C_2 , the data set is $\{LA, m_2\}$ and altogether the data set of the operation is $\{LA, m_1, m_2\}$. Therefore, in configuration C_1 , the distance between op_1 and op_2 is one, the distance between op_1 and op_3 is two, the distance between op_1 and op_4 is three, and the distance between op_1 and op_5 is ∞ . After op_2 completes in configuration C_2 , the distance between op_1 and other operations is decreased by one, since at this configuration the data set of op_1 includes m_2 . During the interval of op_1 , the distance between op_1 and op_3 is one, the distance between op_1 and op_4 is two, and the distance between op_1 and op_5 remains ∞ .

The *d-neighborhood* of an operation op contains all the operations within distance d from op . In the example of Figure 4, the 1-neighborhood of op_1 includes op_1 , op_2 , and op_3 ; the 2-neighborhood of op_1 includes op_1 , op_2 , op_3 and op_4 ; there is no d such that op_5 is in the d -neighborhood of op_1 .

The next definition guarantees progress within the neighborhood of an operation.

Definition 2 *An algorithm is d -local nonblocking if whenever a process takes an infinite number of steps in an operation op , then some operation in the d -neighborhood of op completes.*

4 Algorithms using Built-in Coloring

4.1 CAS-Chromo: Priority Queue and Deque

CAS-Chromo is a doubly-linked list implementation that allows insertions anywhere (*PushLeft*, *PushRight*, *InsertRight*, and *InsertLeft*) and removals only at the ends (*PopLeft*, and *PopRight*); see Figure 2. Our description focuses on the *PushLeft* and *PopLeft* operations.

4.1.1 Overview

CAS-Chromo follows the common scheme [7, 25, 28] of having an operation first acquire *locks* on its data set (LOCK phase), and then apply its changes atomically on these nodes (APPLY phase); finally, the operation releases the locks (UNLOCK phase). The key novelty of our approach is in having nodes that are legally colored¹ with three ordered colors, $c_1 < c_2 < c_3$. (In Figure 3, for example, the items are legally colored with three colors, yellow < blue < red.)

An operation acquires locks on its data set in an increasing order of colors. During the LOCK phase, a *hold-and-wait* scenario happens when an operation holds locks on one or more nodes and waits for another operation to release the lock on another node. Since an operation acquires locks on its data set in an increasing order of colors, and the colors of neighboring nodes are different, hold-and-wait chains have (strictly) *increasing* colors; the chains are short, since the number of colors is small.

Push and insert operations access exactly two adjacent nodes in the list, which must have different colors. Pop operations access three consecutive nodes, two of which may have the same color. To avoid hold-and-wait cycles, pop operations acquire locks on monochromatic nodes according to their order in the list, from left to right. For example, consider a doubly-linked list containing only one node, where the two anchors have the same color. When a *PopLeft* operation and a *PopRight* operation compete on removing the single node in the list, both of them try to lock the left anchor first, and at least one of them (that succeeds in locking the left anchor) succeeds in popping the last item. Moreover, pop operations occur only at the ends of the list, and hence, each of them adds at most one edge to a hold-and-wait chain (one edge at each end of the chain) in addition to at most three edges connecting operations that have conflicts on nodes with increasing colors. This is used to show that the length of a hold-and-wait chain is at most 5 (see Theorem 10).

Maintaining a Legal Coloring: Since operations apply their changes in exclusion, they can ensure that the coloring is legal at all times, by careful color changes during the APPLY phase.

The operations use a temporary color $c_0 < c_1$, which is white in the figures, to simplify the task of maintaining the coloring legal: In the *PushLeft* operation, the color of the new node is c_0 . After the new node is in the list, and while the data set is still locked, the new node is assigned with a color different than its neighbors. In the *PopLeft* operation, after the data set is locked, the color of the right neighbor of the node to be removed is changed to c_0 . After the node is removed, the color of the right neighbor is changed to be different from its neighbors’.

Helping: The simple algorithm described so far may block if a process stops taking steps while holding a lock. An operation op_1 is *blocked* if one of its nodes is already locked by another *blocking* operation op_2 . *Helping* guarantees that some operation makes progress at any time while preserving the good locality properties of the implementation. This means that instead of waiting, the process executing op_1 helps op_2 to complete by executing its steps. Helping is recursive: if, while helping op_2 , the process executing op_1 discovers that op_2 is blocked by a third operation op_3 , then the process helps op_3 to complete, and so on. In this way, *hold-and-help* chains replace hold-and-wait chains, and it is possible to prove that their length is bounded by the number of colors (see Lemma 5). For example, assume that operation op_3 in Figure 3 locks m_2 and then tries to lock m_3 , with color red. If the lock on m_3 is already held by op_4 , then op_3 helps op_4 . However, red is the

¹Adjacent nodes in the list have different colors.

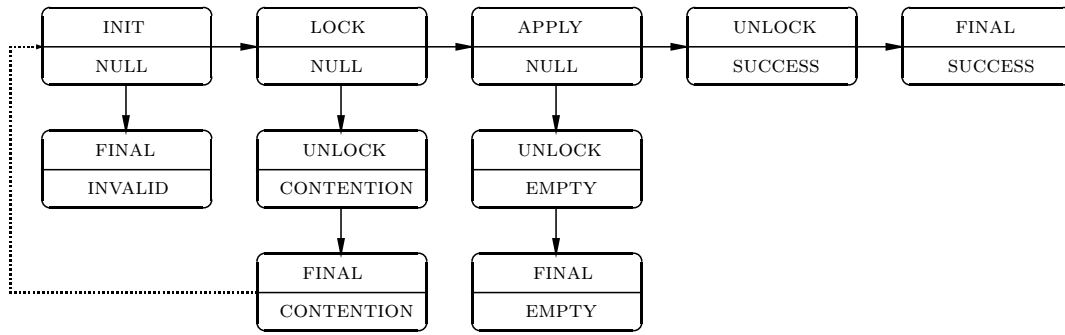


Figure 5: The state transitions of an operation: the lower part of the state is the *result* of the current invocation; the dashed line indicates re-invocation. The best-case scenario, encountering no contention, appear at the top. If the list is empty during a pop operation, then the operation completes without changing the linked list. If the source node is removed during the INIT phase of an operation, then the operation need not apply its changes. If there is contention during the LOCK phase, the operation releases its locks and it re-invoked.

largest color, implying that op_4 already locked all the nodes in its data set. Thus, when helping op_4 , op_3 only needs to apply the changes of op_4 , and not recursively help additional operations.

Dynamic Aspects: Another aspect of the algorithm is in handling the dynamic nature of the data structure, that is, after an operation reads the data set and verifies that the nodes are valid, another operation may modify the nodes and even the list structure, changing the data set of op . Our algorithm addresses this problem in a manner similar to [18]. A *data set memento*, holding a view of the data set when the operation is started, traces inconsistencies in the data set due to changes applied by concurrent operations. If, while locking the data set, an operation discovers that a node in its data set memento is invalid or inconsistent with its memento, it restarts. That is, the operation skips the APPLY phase and goes directly to the UNLOCK phase where it releases all the locks it already acquired and re-invokes the operation. Otherwise, the operation completes its LOCK phase, and the nodes it has locked are consistent with the data set memento, so the operation can continue with the APPLY phase as in a static locking scheme.

4.1.2 Detailed Pseudo-Code and Description

An operation may have several *invocations*, caused by inconsistencies between the items and its data set memento. The locking scheme is re-applied at each invocation until the operation *completes*. Figure 5 shows the state transition diagram of one invocation of an operation.

Figure 6 lists the main types and data structures used in the algorithm.

Operations are objects, whose structure and behavior are defined in the *Operation* class. An operation object is initialized with all the data required for its execution, specifically the source node on which the operation is applied, and the subject node to be inserted to the list (in insert or push operations).

Nodes are also objects. In addition to its *data* attribute, a node contains two pointers *left* and *right*, a *color* and a *lock*. A node memento is composed of a reference to the node itself, *node*, and a copy of the node’s meta data except for the lock, so the node is consistent with its memento even

```

define LEFT = 0, MIDDLE = 1, RIGHT = 2
define Phase = {INIT, LOCK, APPLY, UNLOCK, FINAL}
define Result = {NULL, SUCCESS, CONTENTION, INVALID, EMPTY}

structure State {int seq, Phase phase, Result result} // seq - for repeated invocations
structure NodePtr {Node node, int aba}
structure Color {Color color, int aba}
structure Lock {Operation op, int seq, int aba} // seq - operation's invocation number

structure Node {
    Data    data,
    NodePtr left,
    NodePtr right,
    Color   color,
    Lock    lock
}

structure NodeMemento {
    Node    node,
    NodePtr left,
    NodePtr right,
    Color   color
}

class Operation {
    State    state // initially ⟨0,INIT,NULL⟩
    Node     source // in push/pop operations the source is an anchor
    Node     subject // either the new node or the removed node
    NodeMemento[3] datasetMemento
    Color[3] colorSet
}

```

Figure 6: Types, structures and classes definitions.

if the lock is acquired and released arbitrarily, as long as the other attributed do not change.

Due to helping, several processes may execute the same operation op (by calling the `execute` method of op simultaneously); these are the *executing processes* of op . One of the executing processes—the one that first called the `execute` method of op —is the *initiator* of op .

Since an operation may have several invocations, its execution goes through alternating phases of acquiring and releasing locks. The *state* of an operation is a tuple $\langle seq, phase, result \rangle$: seq is the *invocation number*, an integer, initially 0, that is incremented when the initiator re-invokes the operation; $phase$ indicates the locking scheme phase within the invocation, set to `INIT` at the beginning of an invocation; $result$ holds the result of the current invocation execution, set to `NULL` at the beginning of an invocation.

The CAS primitive suffers from the ABA problem [21]; namely, a process p may read a value A from some memory location l , then other processes change l to B and then back to A, later p applies CAS on l and the comparison succeeds whereas it should have failed. We avoid this problem by associating each attribute with a monotonically increasing *ABA counter*. The attribute and the counter fit into a single memory location and are manipulated atomically; the counter is incremented whenever the attribute is updated. Assuming that the counter has enough bits, the CAS succeeds only if the counter has not changed since the process read the attribute.

Pseudocodes 1, 2 and 3 present the code for CAS-Chromo. The reserved word *self* in the

Pseudocode 1 CAS-Chromo

```
1: Result Operation::execute() {
2:   do
3:     clear data set memento and color set
4:     toInitState()
5:     if source is invalid then
6:       lock  $\leftarrow$  source.lock
7:       lock.op.help(lock.seq) // help lock.op
8:       toFinalInvalidState()
9:     if state.result = INVALID then return
10:    else // (re-)invoke the operation
11:      cloneDataset()
12:      toLockState()
13:      help(state.seq) // help myself
14:      toFinalState()
15:    while state.result = CONTENTION
16:    return state.result
17: }

18: Operation::help(int seq) {
19:   lockDataset(seq)
20:   if state.phase = APPLY then
21:     applyChanges()
22:     toUnlockState()
23:   unlockDataset(seq)
24: }

25: Operation::lockDataset(int seq) {
26:   if state  $\neq$   $\langle$ seq,LOCK,NULL $\rangle$  then return
27:   for each  $c$  in colorSet by increasing order do
28:     lockColor( $c$ ,seq)
29:   toApplyState(seq)
30: }

31: Operation::lockColor(Color  $c$ , int seq) {
32:    $\{nd_i\} \leftarrow c$ -colored nodes in  $seq$ -th memento
33:   while true do
34:      $\{lock_i\} \leftarrow$  get all locks from  $\{nd_i\}$ 
35:     checkNodes( $\{nd_i\}$ ,seq)
36:     if state  $\neq$   $\langle$ seq,LOCK,NULL $\rangle$  then return
37:     // lock all equally colored nodes
38:     for each  $nd_i$  from left to right do
39:       if  $lock_i = \langle \perp, \perp, t \rangle$  then
40:         CAS( $nd_i$ .lock,  $lock_i$ ,  $\langle$ self,seq,t+1 $\rangle$ )
41:         // check if locked all  $c$ -colored nodes or blocked
42:         lock  $\leftarrow$  get lock from  $\{nd_i\}$ 
43:         if lock.op = self then return // succeeded
44:         checkNodes( $\{nd_i\}$ ,seq)
45:         if state  $\neq$   $\langle$ seq,LOCK,NULL $\rangle$  then return
46:         if lock.op  $\neq \perp$  then
47:           // blocked by lock.op - help it
48:           lock.op.help(lock.seq)
49: }

46: Operation::checkNodes(Set(Node)  $\{nd_i\}$ , int seq) {
47:   if invalid node or inconsistent memento then
48:     for each  $i$  do // touch locks
49:        $l_i \leftarrow nd_i$ .lock
50:       // "touch" the ABA counter
51:       CAS( $nd_i$ .lock,  $l_i$ ,  $\langle$  $l_i$ .op,  $l_i$ .seq,  $l_i$ .aba+1 $\rangle$ )
52:   toUnlockContentionState(seq)
53: }

53: Operation::unlockDataset(int seq) {
54:   for each Node  $nd$  in  $seq$ -th memento do
55:     lock  $\leftarrow$  nd.lock
56:     if lock =  $\langle$ self,seq,t $\rangle$  and state.phase = UNLOCK then
57:       CAS(nd.lock, lock,  $\langle \perp, \perp, t+1 \rangle$ )
58: }
```

pseudocode denotes the operation object of the operation whose code is being executed.

The initiator starts the execution with the `execute` method and as long as it suffers from contention and is unable to complete (line 15), the process repeatedly tries to clear the attributes (line 3) and re-invoke the operation. It generates the new data set memento (line 11), and then “helps” itself to follow the locking scheme (line 13): lock nodes in its data set (line 19), apply its changes (line 21), and release the data set (line 23). Different operations extending the *Operation* class, refine the protocols for cloning and manipulating the data set, according to their specifications. Pseudocode 3, shows how the data set memento (usually the source node and both

Pseudocode 2 CAS-Chromo: Detailed code for state transitions

```
59: Operation::toInitState() {
60:   s ← state
61:   CAS(state, s, ⟨s.seq+1,INIT,NULL⟩)
62: }

68: Operation::toLockState() {
69:   s ← state
70:   if s.phase != INIT then return
71:   CAS(state, s, ⟨s.seq,LOCK,NULL⟩)
72: }

78: Operation::toApplyState(int seq) {
79:   s ← state
80:   if s != ⟨seq,LOCK,NULL⟩ then return
81:   CAS(state, s, ⟨seq,APPLY,SUCCESS⟩)
82: }

88: Operation::toUnlockEmptyState() {
89:   s ← state
90:   if s != ⟨seq,APPLY,SUCCESS⟩ then return
91:   CAS(state, s, ⟨seq,UNLOCK,EMPTY⟩)
92: }

63: Operation::toFinalInvalidState() {
64:   s ← state
65:   if s.phase != INIT then return
66:   CAS(state, s, ⟨s.seq,FINAL,INVALID⟩) // LP1
67: }

73: Operation::toUnlockContentionState(int seq) {
74:   s ← state
75:   if s != ⟨seq,LOCK,NULL⟩ then return
76:   CAS(state, s, ⟨seq,UNLOCK,CONTENTION⟩)
77: }

83: Operation::toUnlockState() {
84:   s ← state
85:   if s.phase != APPLY then return
86:   CAS(state, s, ⟨s.seq,UNLOCK,s.result⟩) // LP2
87: }

93: Operation::toFinalState() {
94:   s ← state
95:   if s.phase != UNLOCK then return
96:   CAS(state, s, ⟨s.seq,FINAL,s.result⟩)
97: }
```

or one of its neighbors) is created. The `applyChanges` method changes the nodes according to the specification of the operation and maintains a legal coloring.

An operation op repeatedly tries to lock all nodes with a given color c in its current data set memento (`lockColor` method); after verifying the nodes are valid and consistent with their mementos (line 35), op tries to lock the nodes (line 38). If op discovers that none of these nodes is locked by another operation, when failing to lock them, it simply retries to acquire their locks. Otherwise, op finds that a node in its data set is locked by another, blocking operation op' (line 43), and helps op' to complete (line 44).

Before helping op' , the executing process of op verifies (again) that the nodes are consistent with their mementos (line 41). This is crucial for maintaining the locality properties of the algorithm. Lemma 5 proves that the length of helping chains is bounded by the number of colors, by showing that a process helps along a chain with monotonically increasing colors. The consistency check ensures that the color of the node did not decrease; without this check, a process may help along an unbounded chain with colors alternately increasing and decreasing.

The state attribute is used to synchronize the executing processes of an operation. It is possible that a delayed (slow) process, executing a previous invocation, tries to acquire locks on nodes that are associated with this previous invocation or releases locks that were re-acquired in the current invocation. Therefore, an executing process verifies, before acquiring a lock, that the invocation number of its execution matches the one in the state of the operation (line 36). Furthermore, if an executing process detects inconsistency between the node and the operation's memento (line 35 or line 41) then it violates the locks of these nodes by "touching" them² (line 50) before advancing the operation to the UNLOCK phase. This prevents other delayed processes from acquiring the

²This is similar to the way an operation is invalidated before releasing its nodes in [4].

Pseudocode 3 CAS-Chromo: Detailed code for clone and apply methods.

```
98: PushLeft::cloneDataset() {
    // subject is pushed, source is the left anchor
99:   cloneNode(source, LEFT)
100:  cloneNode(subject, MIDDLE)
101:  right ← datasetMemento[LEFT].right.node
102:  cloneNode(right, RIGHT)
103: }

111: Remove::cloneDataset() {
112:   cloneNode(source, MIDDLE)
113:   left ← datasetMemento[MIDDLE].left.node
114:   cloneNode(left, LEFT)
115:   right ← datasetMemento[MIDDLE].right.node
116:   cloneNode(right, RIGHT)
117: }

123: PushLeft::applyChanges() {
    // MIDDLE is the new node,
    // LEFT is the left anchor
124:   updateRight(MIDDLE,RIGHT)
125:   updateLeft(MIDDLE,LEFT)
    // the new node is now valid
126:   updateColor(MIDDLE)
127:   updateLeft(RIGHT,MIDDLE)
128:   updateRight(LEFT,MIDDLE)
129: }

141: Operation::updateRight(int i, int j) {
142:    $nm_i, nm_j \leftarrow$  get i-th, j-th node mementos
143:   nd ←  $nm_i$ .node
144:   newr ←  $nm_j$ .node
145:   rt ←  $nm_i$ .right
146:   CAS(nd.right, rt, ⟨newr,rt.aba+1⟩)
147: }

155: Operation::updateColor(int i) {
156:   nm ← get i-th node memento
157:   nd ← nm.node
158:   lftc ← nd.left.node.color
159:   rtc ← nd.right.node.color
160:   newc ← choose color not in {lftc,rtc}
161:   clr ← nm.color
162:   CAS(nd.color, clr, ⟨newc,clr.aba+1⟩)
163: }

104: PopLeft::cloneDataset() {
    // subject is popped, source is the left anchor
105:   cloneNode(source, LEFT)
106:   subject ← datasetMemento[LEFT].right.node
107:   cloneNode(subject, MIDDLE)
108:   righth ← datasetMemento[MIDDLE].right.node
109:   cloneNode(righth, RIGHT)
110: }

118: Operation::cloneNode(Node nd, int i) {
119:   if nd = ⊥ then return
    // Assume atomic assignment
120:   datasetMemento[i] ←
    ⟨nd,nd.left,nd.right,nd.data,nd.color⟩
121:   add nd's memento color to the color set
122: }

130: PopLeft::applyChanges() {
    // MIDDLE is popped, LEFT is the left anchor
131:   if empty list then
132:     toUnlockEmptyState()
133:   if state.result = EMPTY then return
134:   setTempColor(RIGHT)
135:   updateRight(LEFT,RIGHT)
136:   updateLeft(RIGHT,LEFT)
137:   updateRight(MIDDLE,⊥)
    // the popped node is now invalid
138:   updateLeft(MIDDLE,⊥)
139:   updateColor(RIGHT)
140: }

148: Operation::updateLeft(int i, int j) {
149:    $nm_i, nm_j \leftarrow$  get i-th, j-th node mementos
150:   nd ←  $nm_i$ .node
151:   newl ←  $nm_j$ .node
152:   lft ←  $nm_i$ .left
153:   CAS(nd.left, lft, ⟨newl,lft.aba+1⟩)
154: }

164: Operation::setTempColor(int i) {
165:   nm ← get i-th node memento
166:   nd ← nm.node
167:   clr ← nm.color
168:   CAS(nd.color, clr, ⟨ $c_0$ ,clr.aba+1⟩)
169: }
```

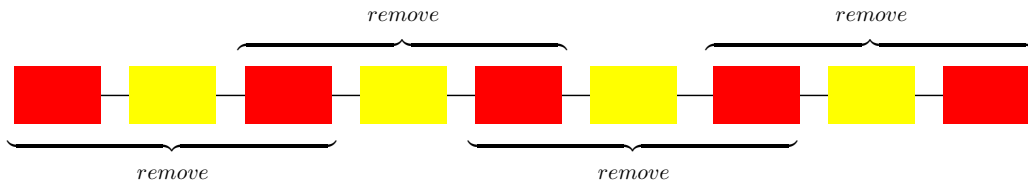


Figure 7: An example of a symmetric scenario; each operation removes a different yellow-colored node, and accesses the node and its two red-colored neighbors.

locks when the operation is not in the LOCK phase (see Lemma 2(5) in the appendix). To prevent a process from releasing locks acquired in a later invocation, the operation adds the invocation number to any lock it acquires (line 38); before a process releases a lock, it verifies (line 56) that the invocation numbers of the lock and its own parameter are equal (see Lemma 2(4)).

4.2 DCAS-Chromo: A Doubly-Linked List Algorithm

We now describe the extensions needed to obtain DCAS-Chromo, which allows removals from the middle of the list. This may create long helping chains, as demonstrated in Figure 7, which shows a long linked-list of nodes with alternating colors: red, yellow, red, yellow, \dots . Consider a set of concurrent operations, each of which is trying to remove a different yellow-colored node, by locking the node and its two red-colored neighbors. If the two red-colored nodes are locked one at a time, in the same order, e.g., first the left neighbor, it is possible that an operation holds a lock on its left red node, and needs to help all operations to its right.

One might suggest to extend the notion of a legal coloring and require that any triple of neighboring nodes is assigned different colors. This certainly allows to follow the color-based locking scheme, but how can we preserve this extended coloring property? In particular, when a node is removed, it is necessary to lock *four* nodes in order to legally re-color the remaining three nodes; this requires to further extend the coloring property to any four consecutive nodes, which in turn requires to lock *five* nodes and so on. This vicious circle is avoided by locking equally-colored nodes *atomically*. An operation accesses at most three consecutive nodes, which are legally colored, so at most two of them have the same color. We use DCAS to atomically lock these nodes, e.g., the two red-colored nodes in the scenario of Figure 7, and to break the symmetry.

The additional operation *Remove*, for removing a non-anchor valid node (the source node) from the list, is very similar to the pop operations (which remove nodes from the ends) except that it does not handle the case where the list is empty; therefore, we omit the pseudocode of `applyChanges` method for *Remove*. The most important modification, relative to CAS-Chromo, is in the `lockColor` method, which now uses DCAS when locking two nodes with the same color. That is, instead of locking the nodes one by one (Pseudocode 1, lines 37-38) DCAS locks both nodes atomically (Pseudocode 4, lines 37-38).

5 Safety Proof: Linearizability

In this section, we prove that CAS-Chromo and DCAS-Chromo are linearizable (Theorem 3). The proof does not assume that DCAS is used, and thus it holds for both algorithms. Using DCAS is critical only for proving that the coloring is legal and showing locality properties of the algorithm,

Pseudocode 4 Code changes for DCAS-Chromo.

```
31: Operation::lockColor(Color c, int seq) {
32:   {ndi} ← get all nodes with color c from the seq-th memento
33:   while true do
34:     {locki} ← get all locks from {ndi}
35:     checkNodes({ndi},seq)
36:     if state != ⟨seq,LOCK,NULL⟩ then return
      // atomically lock two equally colored nodes
37:     if for each ndi, locki = ⟨⊥,⊥,⊥⟩ then
38:       DCAS(nd1.lock, nd2.lock, lock1, lock2, ⟨self,seq,lock1.aba+1⟩, ⟨self,seq,lock2.aba+1⟩)
39:     lock ← get lock from {ndi} // check if succeeded or blocked
40:     if lock.op = self then return // locked all c-colored nodes
41:     checkNodes({ndi},seq)
42:     if state != ⟨seq,LOCK,NULL⟩ then return
43:     if lock.op != ⊥ then // blocked by lock.op operation
44:       lock.op.help(lock.seq) // help blocking operation
45: }
```

and is considered only in Section 6.1.

The linearizability of both algorithms hinges on showing that the implementation follows the locking scheme. Namely, the executing processes preserve the correct phase transitions of the operation—locking, changing and releasing nodes—and take steps in accordance with the operations’ phase. Most importantly, nodes in the data set are changed only while all of them are locked. As mentioned before, this is somewhat more complicated than in previous work [1,4,7,25,28], since the data set is dynamic.

It can be inferred directly from the methods of Pseudocode 2, which are the only way to make state transitions, that an operation follows the state transition diagram in Figure 5. Moreover, the invocation number is increased before every invocation. Hence, no operation makes a transition to the same state tuple more than once.

The following terminology is used in the proofs. The LOCK phase of the r -th invocation is called the r -th LOCK *phase*, and similarly for the other phases. The code implies that an operation is in APPLY phase at most once (since all transitions from it are to a final state), in which case the operation completes and will not be re-invoked again; this is called the *last invocation*. Only the initiator generates the data set memento, once per invocation (line 11); the data set memento written in the r -th invocation is called the r -th *data set memento*; the data set memento of the last invocation is called the *last data set memento*.

Several executing processes can make the transitions to the APPLY and UNLOCK phases concurrently, but other transitions are only made by the initiator. A transition to the APPLY phase only occurs once, in the last invocation; the method implementing a state transition to the UNLOCK phase in case of contention takes as argument the invocation number, to ensure the transition occurs only if it is executed for the correct invocation.

The next two lemmas state the main invariants of the algorithm, indicating that the locking scheme is followed. Their proofs are deferred to Appendix A.

Lemma 1 *An operation op successfully applies changes only when it is in APPLY phase, and only to nodes in op ’s last data set memento.*

Lemma 2 *The following claims all hold for every operation op :*

1. *If op locks a node t in the r -th invocation, then the r -th memento of t in op 's data set is valid and t , excluding t 's lock, has not changed after it was cloned by op in the r -th invocation.*
2. *op advances from the r -th LOCK phase to the r -th APPLY phase only if all the nodes in its data set are consistent with the r -th data set memento, and are locked by op .*
3. *op only applies changes to nodes that are locked by it.*
4. *op releases locks only when it is in UNLOCK phase, and during its r -th UNLOCK phase it only releases locks from nodes it has locked in the r -th invocation.*

Linearizability follows directly from these properties. The next theorem applies for both CAS-Chromo and DCAS-Chromo, since they follow the same locking scheme.

Theorem 3 (Linearizability) *CAS-Chromo and DCAS-Chromo are linearizable.*

Proof: We prove the theorem by identifying, for every operation, a *linearization point* inside its interval, so that the operation appears to occur atomically at this point. The linearization point of an operation op_i is either at the transition to state $\langle \text{last}, \text{FINAL}, \text{INVALID} \rangle$ (line 8), or at the transition to state $\langle \text{last}, \text{UNLOCK}, \text{SUCCESS} \rangle$ (line 22); that is, when the CAS of the transition is applied successfully (line 66, marked LP1, or line 86, marked LP2, respectively). Only one of these occurs in an execution of an operation, and the linearization point is well defined.

In the first case, op_i discovers that the *source* node is invalid, since another operation op_j removes it. Before its transition to the FINAL phase, op_i helps op_j (line 7). By Lemma 1, the transition to the APPLY phase of op_j already occurred and op_i helps it to complete in case it has not completed yet. Thus, op_i need not apply its changes, and it is linearized after op_j .

In the second case, Lemma 2 (1) and (2) imply that when the transition to the APPLY phase occurs in configuration C , all the nodes in the data set memento of op_i are valid, have not changed since they were cloned (except for their locks) and they are locked by op_i . By Lemma 2(4), these nodes remain locked while op_i is in the APPLY phase, which means, by Lemma 2(3), that no other operation changes these nodes and they are modified only by op_i during the execution of the APPLY phase. Finally, the `applyChanges` methods clearly preserve the specification of the corresponding doubly-linked list operations. ■

6 CAS-Chromo and DCAS-Chromo are Local Nonblocking

The legality of the coloring is now used to show that the algorithms are local nonblocking. Formally, a node is *legally colored* if its color is not equal to the colors of its neighbors; the left anchor is legally colored if its color is different from its right neighbor, and analogously for the right anchor.

6.1 DCAS-Chromo

It is simple to see that all operations access three consecutive nodes in the linked list, and that each operation only changes the color of a single node: insert and push operations change the color of the new node, and a pop and remove operations change the color of the right node in their data set. No operation changes the color of the left node in its data set. Since an operation only changes a node while holding its lock, this ensures that the colors of two adjacent nodes is not changed at the same time, even if concurrent operations access them.

Recall that a node is valid if it is an anchor or both its left and right links are not null. The coloring property of the algorithm is stated in the following lemma.

Lemma 4 *All valid nodes are legally colored.*

By Theorem 3, we can assume that the changes are applied in isolation from other operations. Therefore, the proof of this lemma, which appears in the appendix, is merely a step-by-step sequential analysis of the `applyChanges` methods of the various operations.

An operation locks at most three consecutive nodes, and by the lemma, at most two of them have the same color, hence, DCAS suffices to lock equally colored nodes in the operations' data set.

The proof that the algorithm is local nonblocking starts by showing that a process only helps operations within constant distance of the operation it is executing. In the proof, we consider the number of `help` methods the process started executing but have not yet completed.

Lemma 5 *Consider process p that called $h > 0$ `help` methods and completed $h' < h$ of them. If the last call is the `help(r_i)` method of an operation op_i , then the r_i -th invocation of op_i locks a node with color greater than or equal to $c_{h-h'-1}$.*

Proof: The proof is by induction on $k = h - h' - 1$; the base case is when $k = 0$. A node with color c_0 is either a new node that is locked by the operation during its initialization, or the right node in a remove operation that is colored c_0 during the `APPLY` phase after all the data set is locked by the operation.

In the induction step, $k > 0$, and since $h - h' \geq 2$, p called `help` method for at least one operation other than op_i and did not complete. Assume the penultimate `help` method called and not completed by p is for operation op_j . When the `help` method of op_j was called, op_j already locked color $\geq c_{k-1}$, by the induction assumption, and it tries to lock node t with color $c > c_{k-1}$. Process p reads t 's lock (line 39), and discovers op_j failed to lock t (line 40) and that t is consistent with its memento (lines 41-42), i.e., its color did not change. Then, p discovers that op_j is blocked by operation op_i in its r_i -th invocation (line 43) and calls the last `help` method (line 44). Since equally colored nodes are locked atomically, op_i locked color $c \geq c_k$, and the lemma follows. ■

Note that $h - h' - 1$ bounds from above the distance to operations that a process helps. Thus, Lemma 5 implies:

Corollary 6 *If op_i helps op_j , at distance d , then op_j already acquired a lock on a node with color greater or equal to c_d . In particular, op_j is in the 3-neighborhood of op_i , and if $d = 3$ then op_j completed the `LOCK` phase.*

Afek et al. [1] define the next notion, capturing the locality of implementations in terms of memory contention:

Definition 3 *An algorithm has d -local contention if two processes, p_1 and p_2 , access the same memory location in operations op_1 and op_2 , respectively, only if op_1 and op_2 are within distance d .*

Theorem 7 (Local contention) *DCAS-Chromo has 7-local contention.*

Proof: Two processes p_i and p_j access the same memory location if they help execute operations, op_k and op_l respectively, within distance one. By Corollary 6, op_i is in the 3-neighborhood of op_k and op_j is in the 3-neighborhood of op_l . Thus, the distance between op_i and op_j is at most 7. ■

Once an operation is in its APPLY phase, it is straightforward that it completes after one of its executing processes takes a constant number of steps. It remains to prove that if the operation is blocked outside the APPLY phase, then some “nearby” operation (in a sense made precise by Lemma 8) completes. We do so by considering executions of the loop of `lockColor(c, r)` method (lines 33-44), called a *c-locking iteration*. Lemma 8 below shows that in every locking iteration of an executing process, whether successful or not, some “nearby” operation makes progress.

Fix an arbitrary operation op_b initiated by process p_b ; progress in the neighborhood of op_b is tracked by three counters:

- The *completed operations counter*, denoted co , initially 0, is increased whenever an operation in the 4-neighborhood of op_b completes.
- The *color counter*, denoted cl , holds the color of the last locking iteration that the initiator of op_b executed.
- The *changes counter*, denoted ch , initially 0, is increased whenever an operation in the 5-neighborhood of op_b changes an item in its data set.

The values of the counters in a configuration C are denoted $co(C)$, $cl(C)$, $ch(C)$. Note that co and ch are nondecreasing, while cl is not necessarily monotone.

Assume that process p_b takes an infinite number of steps, executing infinitely many locking iterations, without completing op_b . Let the configurations at the start of the locking iterations of p_b be denoted $C_0, C_1, \dots, C_t, \dots$, in the order they occur. The next lemma argues that each configuration C_t is a milestone in the progress of the operations in the neighborhood of op_b :

Lemma 8 *For every $t > 0$, the value of at least one of the counters co , cl , or ch at configuration C_t is strictly larger than the value of the corresponding counter in configuration C_{t-1} .*

Proof: Assume that process p_b starts a c -locking iteration at C_{t-1} , during a `lockColor(c, r)` method of op , the j -th operation of p_i , and a c' -locking iteration at C_t , during a `lockColor(c', r')` method of op' , the j' -th operation of $p_{i'}$. By Corollary 6, op and op' are in the 3-neighborhood of op_b .

Consider first the case that $i \neq i'$, i.e., the operations are issued by different processes.

If op completes before the second iteration, then $co(C_{t-1}) < co(C_t)$. Otherwise, the first locking iteration of op failed. If the failure is due to contention, then some operation op_l at distance one from op applied a change to its data set; since op_l is in the 4-neighborhood of op_b , $ch(C_{t-1}) < ch(C_t)$. Otherwise, the first locking iteration of op failed since op' blocked it. That is, op fails to lock a node t with color c since it is already locked by op' , and then p_b helps op' and executes the second c' -locking iteration. Since op' atomically locks all the nodes with color c , $c < c'$ and hence, $cl(C_{t-1}) < cl(C_t)$.

Now, consider the case that $i = i'$, i.e., both operations are issued by the same process, and therefore, $j \leq j'$. If $j < j'$, p_i completed its j -th operation and $co(C_{t-1}) < co(C_t)$.

Otherwise, $j = j'$, i.e., both locking iterations are of the same operation. The invocation number of the locking iterations is monotonically increasing, thus, $r \leq r'$. If $r < r'$, then op is re-invoked before the second locking iteration, due to contention in the first iteration. Thus, in

the first iteration some operation op_l at distance one from op applied a change to its data set that failed op . The operation op_l is in the 4-neighborhood of op_b and $ch(C_{t-1}) < ch(C_t)$.

Otherwise, $r = r'$, i.e., both locking iterations are of the same invocation. The colors p_b is locking in the same invocation of the same operation are nondecreasing, thus, $c \leq c'$. If $c < c'$ then $cl(C_{t-1}) < cl(C_t)$.

Finally, we are left with the case that $i = i'$, $j = j'$, $r = r'$, and $c = c'$, that is, two consecutive c -locking iterations in the same invocation of the same operation. The process executing the locking iterations, p_b , fails to lock some node t with color c in the first iteration. Then, without helping any other operation (since the lock is already released), p_b retries the locking iteration. The process p_b fails to lock t in the first iteration since another operation op_l , in the 1-neighborhood of op , holds the lock on t . The operation op_l is in the 4-neighborhood of op_b . If op_l releases the lock on t after it completes, then $co(C_{t-1}) < co(C_t)$. Otherwise, the lock is released since op_l discovers that a node it is trying to lock, t' , is inconsistent with its memento. Thus, another operation op_k in the 1-neighborhood of op_l changed t' after op_l generated the memento of t' . Since op_k is in the 5-neighborhood of op_b , $ch(C_{t-1}) < ch(C_t)$. ■

Theorem 9 *DCAS-Chromo is a 4-local nonblocking implementation of a doubly-linked list.*

Proof: Consider the initiator p_b of an operation op_b . By Lemma 8, at least one counter increases with each locking iteration of p_b . Since the color counter is at most 3, after at most three consecutive locking iterations, some counter other than the color counter must increase. If the completed operations counter increases, then some pending operation in the 4-neighborhood of op_b completes. Otherwise, the changes counter increases. Once it is in the APPLY phase, an operation completes within a constant number of changes. Thus, after p_b executes a number of locking iterations that is linear in the number of operations in the 5-neighborhood of op_b , some pending operation in the 4-neighborhood of op_b completes. Since there is a finite number of processes, it follows that after p_b takes a finite number of steps, some operation in the 4-neighborhood of op_b completes. ■

6.2 CAS-Chromo

CAS-Chromo does not support removals from the middle of the linked list, and hence only *pop* operations acquire locks on three nodes, which may include two nodes with the same color. Since operations lock equally colored nodes by their order in the list from left to right, whenever a process calls a new *help* method it helps a new operation. Thus, the number of *pop* operations a process started helping and did not complete is at most two, and helping cycles are avoided.

In order to prove the locality properties of the algorithm we revise Lemma 5 and Corollary 6.

Lemma 5' *Consider process p that called $h > 0$ help methods and completed $h' < h$ of them, such that from the $h - h'$ uncompleted help methods, ℓ are of pop operations. If the last call is the $help(r_i)$ method of an operation op_i , then the r_i -th invocation of op_i locks a node with color greater than or equal to $c_{h-h'-1-\ell}$.*

Proof: The proof is by induction on $k = h - h' - 1$. The base case, $k = 0$, follows by arguments similar to those applied in the base case of the proof of Lemma 5.

In the induction step, $k > 0$ and since in this case, $h - h' \geq 2$, p called the *help* method of least one operation other than op_i did not complete. Assume the penultimate *help* method called and not completed by p is the *help* method of operation op_j .

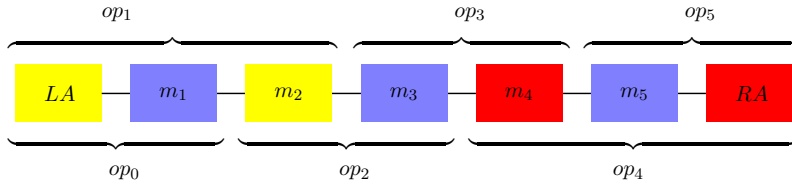


Figure 8: CAS-Chromo worst case scenario for overlapping deque operations.

We first assume that op_i is a pop operation. By the induction assumption, when the `help` method of op_j is called, op_j locked a color greater or equal to $c_{k-1-(\ell-1)} = c_{k-\ell}$. Process p helps op_j to lock a node t with color $c > c_{k-\ell}$. We review p 's steps while helping op_j to show that op_i locked color c : p reads t 's lock (line 39), and discovers op_j failed to lock t (otherwise it returns in line 40) and that t is consistent with its memento (line 41), i.e., its color is still c . Then p discovers that op_j is blocked by operation op_i in its r_i -th invocation (line 43) and calls the last `help` method (line 44). The operation op_i locked t with color c in its r_i -th invocation, and the lemma holds.

If op_i is not a pop operation, the the induction assumption implies that when the `help` method of op_j is called, op_j already locked color greater or equal to $c_{k-1-\ell}$. Process p helps op_j to lock some node t with color $c \geq c_{k-\ell}$ and takes the same steps as in the previous case, to find that op_j is blocked by op_i in its r_i -th invocation, then it calls the last `help` method. The operation op_i , which is not a pop operation, has only one node t in its data set with color $c \geq c_{k-\ell}$, which it locked in its r_i -th invocation, and the lemma holds. ■

Corollary 6' *If op_i helps op_j , at distance d , then op_j already acquired a lock on a node with color greater or equal to c_{d-2} . In particular, op_j is in the 5-neighborhood of op_i , and if $d = 5$ then op_j completed the LOCK phase.*

The revised lemma and corollary imply that CAS-Chromo has 11-local contention, in a manner similar to the proof for DCAS-Chromo. We next argue that the worst-case scenario, in terms of delay and helping chains, is the one presented in Figure 8, to get a better bound of 5-local contention. Assume op_0 is a *PushLeft* operation, op_1 is a *PopLeft* operation, op_2 and op_3 are *InsertRight* operations, op_4 is a *PopRight* operation and op_5 is a *PushRight* operation. Consider an execution in which op_1 locks the left anchor, op_2 locks m_2 , op_3 locks m_3 , op_4 locks m_4 , and op_5 locks m_5 and the right anchor. The operation op_5 is in the 5-neighborhood of op_0 ; these operation contend while accessing the right anchor: op_0 tries to lock the left anchor and thus it helps op_1 ; op_1 tries to lock m_2 and thus it helps op_2 ; op_2 tries to lock m_3 and thus it helps op_3 ; op_3 tries to lock m_4 and thus it helps op_4 ; op_4 tries to lock the right anchor and thus it helps op_5 . This implies:

Theorem 10 *CAS-Chromo is an implementation of a doubly-linked list, allowing removals only at the ends, that has 5-local contention and is 5-local nonblocking.*

An implementation of the deque data structure provides operations only at the ends. Therefore, the worst case scenario has a shorter linked list consisting only the nodes m_1 , m_2 and m_5 (and the anchors), and the execution only of operations op_0 , op_1 , op_4 , and op_5 in Figure 8, where op_1 and op_4 access a common node, m_2 . The analysis can be further tightened to show:

Theorem 11 *CAS-Chromo is an implementation of a deque that has 3-local contention and is 3-local nonblocking.*

7 Related Work

This section discusses two research threads related to our results: generic implementation techniques based on locking, and specific algorithms for linked-list data structures.

7.1 Generic Locking Scheme

The original *locking scheme* [7, 25, 28] systematically derives implementations of concurrent data structures from arbitrary lock-based algorithms, by going through the LOCK, APPLY, and UNLOCK phases. During the LOCK phase, an operation may hold locks on one or more items while waiting for another operation to release the lock on another item. The latter operation might also be waiting for a third operation to release a lock, leading to a *hold-and-wait chain* of operations.

The operations may help *recursively*, namely, a process helps another process to help a third process and so on, possibly causing long *helping chains*. For example, assume the nodes in Figure 3 are locked from left to right. Consider an execution α in which op_2 , op_3 and op_4 concurrently lock their left-most nodes successfully, and then op_1 tries to lock its nodes while the other operations are delayed. Since m_2 is locked by op_2 , op_1 has to help op_2 ; since m_4 is locked by op_3 , op_1 has to help op_3 ; and since m_5 is locked by op_4 , op_1 has to help op_4 . Thus op_1 is delayed by a chain of conflicting operations. In general, op_1 can be delayed by any operation within finite distance from it, implying that the implementation is not local.

In some implementations ([25]), an operation helps only an immediate neighbor. Nevertheless, the number of steps a process performs depends on the length of the longest path connected to this operation in the conflict graph. Consider again an execution that starts with op_2 , op_3 and op_4 locking their low-address nodes successfully, then op_1 fails to lock m_2 , op_2 fails to lock m_4 , and op_3 fails to lock m_5 ; each operation then helps its (immediate) neighbor. Prior to helping, op_2 and op_3 relinquish their locks and stop taking steps, thus op_1 and op_2 discover their help is unnecessary. Assume that op_4 completes, and again op_1 , op_2 and op_3 try to lock their data sets. It is possible that op_2 and op_3 lock their low-address nodes, and op_1 tries, in vain, to help op_2 , which releases its locks due to op_3 , etc. As the length of the path of overlapping operations increases, the number of times op_1 futilely helps op_2 increases as well.

The *color-based locking* scheme for binary operations [4] bounds the length of helping chains by coloring the data items with ordered colors. An operation starts by coloring the nodes it is going to access with a constant number of colors, so that neighboring nodes have different colors, and then acquires locks on data items in an increasing order of colors. In this scheme, op helps op' only if op' already locked a higher color. It can be shown that the length of helping chains is bounded by the number of colors, and an operation helps only operations at constant distance.

Afek et al. [1] extend this scheme to arbitrary k -ary operations. In addition to local contention (Definition 3), Afek et al. also define an implementation to have *d -local step complexity*, if the step complexity of an operation depends only on the number of operations within distance d of it.

These schemes [1, 4] apply with arbitrary data sets, and must therefore color the nodes at the beginning of each operation. This is done by obtaining information about operations (and their data sets) at non-constant distance, leading to $O(\log^* n)$ -local step complexity and contention, and complicating the implementations. By keeping the coloring legal, our approach renders this initial coloring obsolete, thereby avoiding its cost. In particular, our progress proofs imply that CAS-Chromo has 3-local step complexity, and DCAS-Chromo has 5-local step complexity.

algorithm	insertions	removals	primitive	interference	comments
Harris [17]	anywhere	anywhere	CAS	any pair	singly-linked list
Greenwald [14]	anywhere	anywhere	DCAS	any pair	
Michael [23]	anywhere	anywhere	CAS	any pair	singly-linked list
Sundell and Tsigas [26]	anywhere	anywhere	CAS	any pair	
DCAS-Chromo	anywhere	anywhere	DCAS	distance ≤ 7	
Greenwald [15]	ends	ends	DCAS	opposite ends	
Agesen et al. [2]	ends	ends	DCAS	distance = 1	
Michael [22]	ends	ends	CAS	opposite ends	
Herlihy et al. [19]	ends	ends	CAS	distance = 1	obstruction free, array-based
Sundell and Tsigas [27]	ends	ends	CAS	distance ≤ 2	
CAS-Chromo	ends	ends	CAS	distance ≤ 3	
CAS-Chromo	anywhere	ends	CAS	distance ≤ 5	

Table 1: Linked list algorithms; *interference* indicates which operations may delay other operations.

7.2 Previous Linked-List Algorithms

Several papers proposed implementations of dynamic linked list data structures (see Table 1).

Harris [17] used CAS to implement a singly-linked list, with insertions and removals anywhere; however, in this algorithm, a process can access a node previously removed from the linked list, possibly yielding an unbounded chain of uncollected garbage nodes. Michael [23] fixed these memory management issues. Elsewhere [22], Michael proposed an implementation of a deque; in this algorithm, a single word (called *anchor*) holds the head and tail pointers, causing all operations to interfere with each other, and making the implementation inherently sequential. Sundell and Tsigas [27] avoid the use of a single anchor, allowing operations on the two ends to proceed concurrently. They extend the algorithm to allow insertions and removals in the middle of the list [26]; in the latter algorithm, a long path of overlapping removals may cause interference among distant operations; moreover, during intermediate states, there can be a consecutive sequence of inconsistent backward links, causing part of the list to behave as singly-linked. An *obstruction-free* deque, providing a weaker progress property, was proposed by Herlihy et al. [19]; besides blocking when there is even a little contention, this array-based implementation bounds the deque’s size.

Greenwald [14, 15] suggests to use DCAS to simplify the design of many data structures. His implementations of deques, singly-linked and doubly-linked lists synchronize via a single designated memory location, resulting in a strictly sequential execution. Agesen et al. [2] present the first DCAS-based, dynamically-sized deque implementation supporting concurrent access to both ends of the deque, and has 1-local step complexity; this algorithm does not allow operations in the middle of the linked list. SNARK [8] is an attempt for further improvement that uses only a single DCAS primitive per operation in the best case, instead of two. Unfortunately, SNARK is incorrect and the corrected version allows removed nodes to be accessed from within the deque, thus preventing the garbage collector from reclaiming long chains of unused nodes [12]. The authors argue that primitives more powerful than DCAS, e.g., 3CAS, are needed in order to obtain simple and efficient implementations of data structures guaranteeing that some operation makes progress at any time.

We show that DCAS suffices for these purposes. Moreover, in our algorithms, an operation completes within $O(1)$ steps in an execution suffix if it is running solo, i.e., it has constant *obstruction-*

free step complexity [13]. Our algorithms do not leave accessible chains of stale “garbage” nodes.

8 Discussion

This paper presents a new built-in coloring approach for designing high-throughput implementations of linked list data structures. We show a DCAS-based implementation of insertions and removals in a doubly-linked list; for a deque and priority queues, where nodes are removed only at the ends, the implementation uses CAS. These implementations are intended as a proof-of-concept and require further optimizations to make them more practical; it is also necessary to implement a *search* operation in order to support the full functionality of priority queues and lists. Finally, it is interesting to explore other applications of our scheme, e.g., for tree-based data structures.

One of our algorithms, DCAS-Chromo, uses DCAS, which is seldom provided in hardware, but DCAS is an ideal candidate to be supported by *hardware transactional memory* [9–11], being a short transaction with small static data set. Alternatively, DCAS can be simulated in software from CAS [4, 13], or by applying a simple randomized algorithm [16]. In particular, using the highly-concurrent implementation of Attiya and Dagan [4], which is $O(\log^* n)$ -local nonblocking, yields an implementation that is $O(\log^* n)$ -local nonblocking, using only CAS.

Acknowledgments: We thank David Hay, Danny Hendler and Gadi Taubenfeld for helpful comments.

References

- [1] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations. In *PODC 1997*, pages 111–120.
- [2] O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, M. Moir, N. Shavit, and G. L. S. Jr. DCAS-based concurrent deques. *Theory Comput. Syst.*, 35(3):349–386, 2002.
- [3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.
- [4] H. Attiya and E. Dagan. Improved implementations of binary universal operations. *J. ACM*, 48(5):1013–1037, 2001.
- [5] H. Attiya and E. Hillel. Highly-concurrent multi-word synchronization. In *ICDCN 2008*, pages 112–123.
- [6] H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations and Advanced Topics*. John Wiley& Sons, second edition, 2004.
- [7] G. Barnes. A method for implementing lock-free shared-data structures. In *SPAA 1993*, pages 261–270.
- [8] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. S. Jr. Even better DCAS-based concurrent deques. In *DISC 2000*, pages 59–73.
- [9] D. Dice, Y. Lev, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware TM. In *SPAA 2010*, pages 325–334.
- [10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS 2009*, pages 157–168.
- [11] S. Diestelhorst and M. Hohmuth. Hardware acceleration for lock-free data structures and software-transactional memory. In *EPHAM 2008*.

- [12] S. Doherty, D. Detlefs, L. Grove, C. H. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. L. S. Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA 2004*, pages 216–224.
- [13] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free step complexity: Lock-free dcas as an example (brief announcement). In *DISC 2005*, pages 493–494.
- [14] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *PODC 2002*, pages 260–269.
- [15] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
- [16] P. H. Ha, P. Tsigas, M. Wattenhofer, and R. Wattenhofer. Efficient multi-word locking using randomization. In *PODC 2005*, pages 249–257.
- [17] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC 2001*, pages 300–314.
- [18] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*, pages 388–402.
- [19] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS 2003*, pages 522–529.
- [20] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [21] IBM. *IBM System/370 Extended Architecture, Principle of Operation*, 1983. IBM Publication No. SA22-7085.
- [22] M. M. Michael. CAS-based lock-free algorithm for shared dequeues. In *Euro-Par 2003*, pages 651–660.
- [23] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA 2002*, pages 73–82.
- [24] J. Schneider and R. Wattenhofer. Bounds On Contention Management Algorithms. In *ISAAC 2009*.
- [25] N. Shavit and D. Touitou. Software transactional memory. *Dist. Comp.*, 10(2):99–116, 1997.
- [26] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology, 2004.
- [27] H. Sundell and P. Tsigas. Lock-free and practical doubly linked list-based dequeues using single-word compare-and-swap. In *OPODIS 2004*, pages 240–255.
- [28] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *PODS 1992*, pages 212–222.

A Safety Proofs: Lemmas 1 and 2

Lemma 1 An operation op successfully applies changes only when it is in APPLY phase, and only to nodes in op 's last data set memento.

Proof: An executing process of op that calls `applyChanges` first verifies that the operation is in the APPLY phase (line 20), implying that this is the last invocation of the operation, and it holds the last data set memento. Since an operation changes only nodes in its data set memento, executing processes apply the same changes on the same nodes (from the last data set memento).

Let p_j be the process that advances op from APPLY phase to UNLOCK phase (line 22). Before changing the state, p_j executes `applyChanges` (line 21), thus CAS is applied at least once to each of

the attributes based on values in the node mementos (see the code in Figure 3), prior to the state transition. Once a CAS is applied successfully the attribute is inconsistent with its memento since at least the ABA-prevention counter has changed. If after the transition another process p_i applies CAS to some attribute while applying op 's changes, p_i 's CAS fails. ■

In order to prove Lemma 2(1-4), it is helpful to inductively carry two additional items (5 and 6).

Lemma 2' (extended) *The following claims all hold for every operation op :*

1. *If op locks a node t in the r -th invocation, then the r -th memento of t in op 's data set is valid and t , excluding t 's lock, has not changed after it was cloned by op in the r -th invocation.*
2. *op advances from the r -th LOCK phase to the r -th APPLY phase only if all the nodes in its data set are consistent with the r -th data set memento, and are locked by op .*
3. *op only applies changes to nodes that are locked by it.*
4. *op releases locks only when it is in UNLOCK phase, and during its r -th UNLOCK phase it only releases locks from nodes it has locked in the r -th invocation.*
5. *op locks nodes only when it is in LOCK phase, and during its r -th LOCK phase it only locks nodes that are in its r -th data set memento.*
6. *When an executing process of op returns from $\text{lockColor}(c,r)$, either all nodes with color c in the r -th data set memento are locked by op , or op is not in the r -th LOCK phase.*

Proof: The claims are proved simultaneously by induction on the execution length. In the base case, the empty execution, all claims vacuously hold. In the induction step, we consider each claim:

1. If the r -th memento of a node t in op 's data set is invalid no executing process tries to lock t (line 35, and line 50).

Now, assume t has changed after op cloned it in the r -th invocation, and that p_i , an executing process of op calls $\text{lockColor}(c,r)$, where c is the color of t . If the change occurs before p_i verifies t is consistent with its memento (line 35), then p_i does not try to lock t . Otherwise the change occurs after verifying the consistency, and in particular after p_i reads the content of t 's lock (line 34). By Lemma 1, the change is applied by an operation op' in APPLY phase. By (3), t is locked by op' when applying the change. If op' locked t before p_i reads the content of t 's lock, then by (4) it is not released until after op' applied the changes, i.e., until after p_i reads the lock, then p_i does not try to lock t . Otherwise, op' locked t after p_i reads t 's lock, in this case, p_i fails acquiring t 's lock, since the lock's ABA-prevention counter has changed.

2. A transition of op from the r -th LOCK phase to the r -th APPLY phase by an executing process occurs only after the executing process calls $\text{lockColor}(c,r)$ with all colors from the r -th color set while op is in the r -th LOCK phase. By (6), when returning from each such invocation, all relevant nodes are locked by op , and by (4), they were not released since then. As the set of colors includes all nodes in the r -th data set memento, they are all locked by op while the transition occurs. By (1), no node in the r -th data set memento is changed before op advances to the APPLY phase. So, when the transition occurs all nodes in the r -th data set memento are locked by op , and they are consistent with their mementos.
3. By Lemma 1, op only applies changes while it is in APPLY phase and only to nodes that are in the last data set memento. By (2), when the transition to APPLY phase occurs all nodes in the r -th (last) data set memento are locked by op , and by (4), it does not release the nodes while it is in the APPLY phase, implying that op changes a node only while holding its lock.

4. The transition of op from the r -th UNLOCK phase to the r -th FINAL phase (line 14) by the initiator of op , p , occurs after p calls the `unlockDataset` method (line 23), to unlock all nodes in the r -th data set memento (line 54). All processes executing the r -th invocation try to release the same nodes from the r -th data set memento, since by (5), op only locks nodes from the r -th data set memento in the r -th invocation. When a process p_i unlocks op 's data set while executing the r -th invocation of op , it reads the lock on a node t (line 55), and tries to release t (line 57) after verifying that op is in the UNLOCK phase and t is locked by op in its r -th invocation (line 56). It can be easily verified from the code that after p completes the `unlockDataset` method all the nodes that were locked by op are released. Thus, if p_i tries to release the locks after the transition occurs, the CAS fails since the ABA-prevention counter of the lock has changed.
5. Consider an executing process p_i executing the r -th invocation of op . First p_i reads t from op 's r -th data set memento (line 32); then p_i reads t 's lock (line 34); verifies that t 's memento is valid (line 35); and that op is in the r -th LOCK phase (line 36). Let p_j be the executing process that advances op from the r -th LOCK phase either to the r -th APPLY phase or to the r -th UNLOCK phase. Assume the transition occurs after p_i verifies the phase, and specifically after it reads t 's lock, but before p_i tries to lock t (line 38). If p_j advances op to the r -th APPLY phase, by (2) all nodes in the r -th data set memento, including t , are locked by op when p_j makes the transition. Thus either p_i discovers that t is locked by op or it fails acquiring t 's lock, since at least the ABA-prevention counter has changed.

Otherwise, p_j advances to the r -th UNLOCK phase since it discovers that some node t' in the r -th data set memento is inconsistent with its memento (line 35 or line 41). There are three cases depending on the order between the colors of the nodes:

- (i) t' and t have the same color. Before the transition occurs, p_j violates the locks of both nodes by "touching" their ABA-prevention counter (line 50). By (1), since t changed while op is in the r -th LOCK phase, p_i cannot successfully lock it in this invocation.
 - (ii) t' has lower color than t . Thus, p_i executes `lockColor(c', r)`, where c' is the color of t' , before trying to lock t . By (6), t' was locked by op and by (1), it has not changed while op is in the r -th LOCK phase, which contradicts the assumption that p_j discovers it is inconsistent with its memento.
 - (iii) t' has higher color than t . Thus, p_j executes `lockColor(c, r)`, where c is the color of t , before discovering the change in t' . By (6), t was locked by op . Thus either p_i discovers that t is locked by op or it fails acquiring t 's lock, since at least the ABA-prevention counter has changed.
6. An executing process of op that executes `lockColor(c, r)` first verifies that op is in the r -th LOCK phase (line 26). It returns from the method in one of three cases: Two cases are when it recognizes a change in the state (line 36 or line 42), and it is evident by the state diagram that when returning from the method, op is no longer in the r -th LOCK phase. The third case is after verifying all the nodes with color c in the r -th data set memento are locked by op (line 40). Now, if when returning from the method some of the nodes are not locked by op , then, by (4), they are released by the operation that locked them, while it is in UNLOCK phase, so this case also satisfies the condition. ■

B Proof of Lemma 4

The proof is by induction on the execution order. In the base case, the linked list is empty: the left anchor is colored c_1 and the right anchor is colored c_3 , and hence, they are legally colored.

Induction step: a node can become illegally colored only when some operation applies its changes to the node or one of its neighbors. By Lemma 2(3), an operation changes a node only if it holds a lock on it. This implies that no node is inserted or removed immediately to the left or to the right of an operation data set while the operation applies its changes. Moreover, by the above observation a pop and remove operations only change the color of the right node in the data set and an insert or push operations only change the color of the new, i.e., middle, node in the data set. Thus, we can derive the next claim:

Claim 12 *An operation changes a node's color only if it holds locks on the node and its left neighbor.*

We analyze every step in the APPLY phase of an operation, and we show that after each such step the node that was changed is still legally colored. Consider first the *PushLeft* operation presented in Figure 9. The data set of the operation, op_1 , is the new node (m), the left anchor (LA), and its right neighbor (m_1). While op_1 is applying its changes, other operations neither remove nor insert nodes to the right of the right neighbor node, and also do not change the colors of the nodes in the data set. Claim 12 imply that other operations do not change the color of an additional right neighbor (m_2). For *PushRight* or insert operations, the induction assumption also implies that a left neighbor is legally colored even if its color is changed by another operation. It remains to show, by inspecting the code, that the changes applied by the operation keep the nodes legally colored.

- Line 124, update right link of the new node: the new node is not yet valid (Figure 9(b));
- Line 125, update left link of the new node: the new node is valid and it is legally colored (Figure 9(c));
- Line 126, the new node is assigned with a non-temporary color different than its neighbors and the new node is legally colored (Figure 9(d));
- Line 127, update left link of the right neighbor (m_1): the right neighbor has color different than the colors of the new node and the right neighbor (m_2), and thus it is legally colored (Figure 9(e)).
- Line 128, update right link of the left anchor (or the source node in the case of an *insertRight* operation): the left anchor has color different than the color of the new node (in the case of an *insertRight* operation the source node also has color different than the color of the left neighbor), and thus it is legally colored (Figure 9(f));

We next analyze the *PopLeft* operation (see Figure 10). The data set of the operation, op_2 , is the left anchor (LA) and its right neighbors (m_1 and m_2). While op_2 is applying its changes, other operations neither remove nor insert nodes to the right of the right neighbor, and also do not change the colors of the nodes in the data set. Claim 12 implies that other operations do not change the color of an additional right neighbor (m_3). For a *PopRight* operation, the induction assumption implies that the left neighbor is legally colored even if its color is changed by another operation. We show again that the operation's changes keep the nodes legally colored:

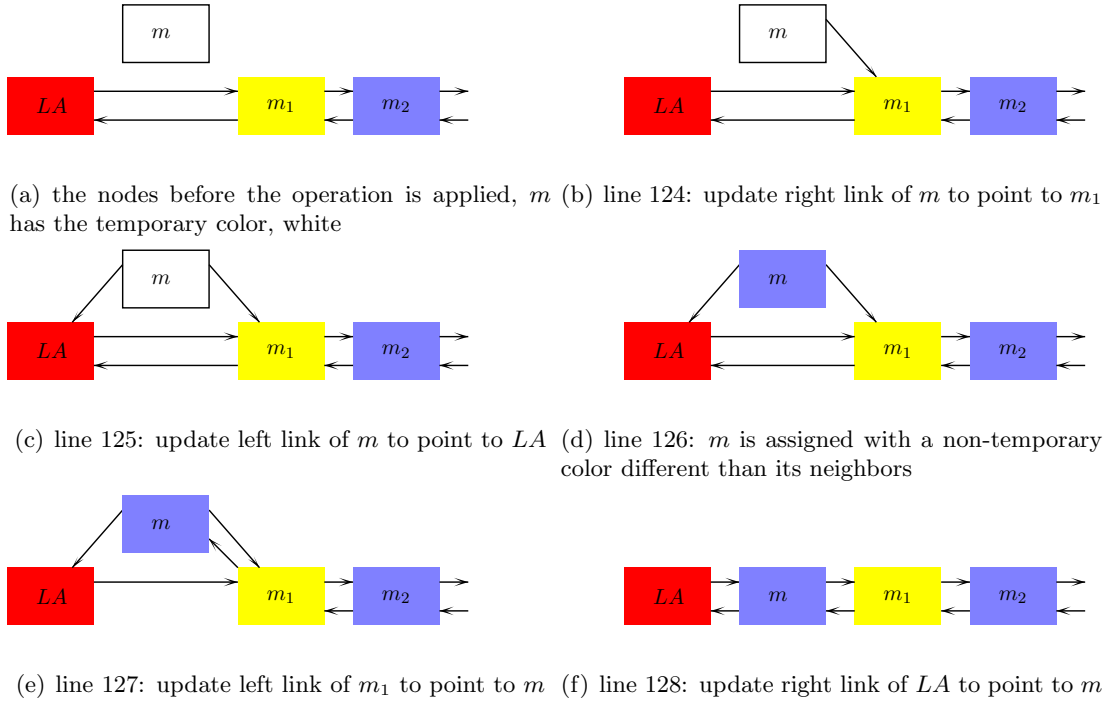


Figure 9: An example of an execution of the `applyChanges` method of *PushLeft* operation (see Pseudocode 3). Given the nodes LA, m_1, m_2 from Figure 3, op_1 pushes a new node, m , into the left side of the list.

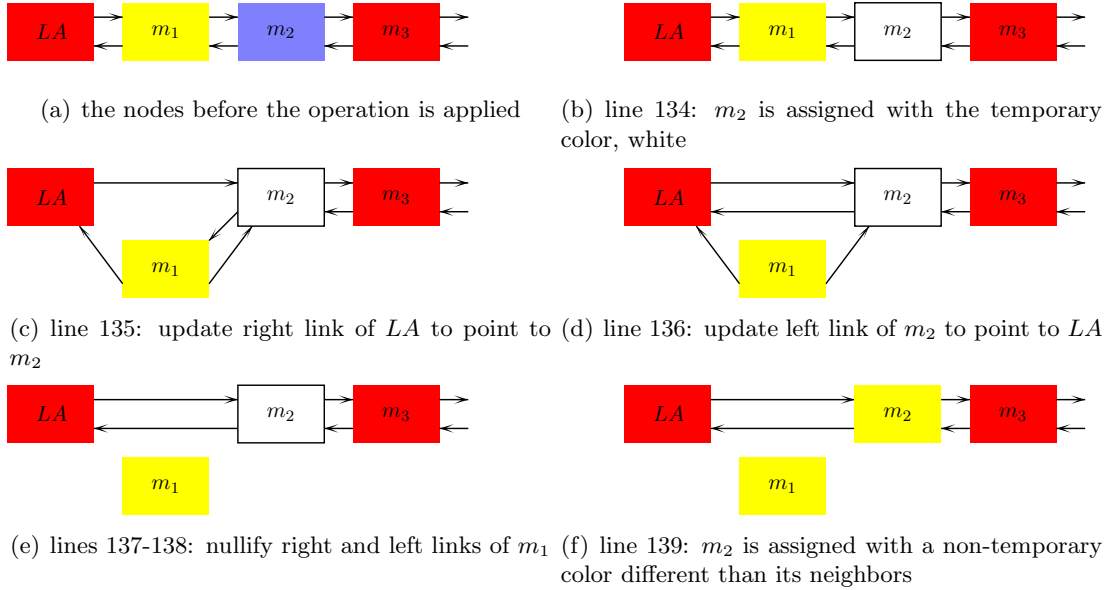


Figure 10: An example of an execution of the `applyChanges` method of *PopLeft* operation (see Pseudocode 3). Given the nodes LA, m_1, m_2, m_3 from Figure 3, op_2 pops the node m_1 .

Line 134, the right neighbor (m_2) is assigned with the temporary color: the right neighbor is legally colored, since the subject node (m_1) and the right neighbor (m_3) have colors different than the temporary color (Figure 10(b));

Line 135, update right link of the left anchor (m_2): the left anchor has a non-temporary color, and thus it is legally colored (Figure 10(c));

Line 136, update left link of the right neighbor: the right neighbor is legally colored, since the left anchor and the adjacent node to the right have colors different than the temporary color (Figure 10(d));

Line 137, set right link of the subject node to null: the source node is now invalid (Figure 10(e));

Line 138, set left link of the subject node to null: the subject node is invalid (Figure 10(e));

Line 139, the right neighbor is assigned with a non-temporary color different than its neighbors, and thus it is legally colored (Figure 10(f)).

It remains to show that the *remove* operation keeps the coloring legal (see Figure 11). The remove operation manipulates its data set in a manner similar to the pop operation. We analyze the *Remove* operation, op_4 , presented earlier in Figure 3. Figure 11(a) presents the nodes m_2, m_3, m_4, m_5, m_6 from Figure 3, op_4 removes the source node m_4 . The data set of the operation is the source node and its neighbors (m_3 and m_5). During the LOCK phase, op_4 first locks m_4 , and then atomically acquires the locks on m_3 and m_5 , which are equally colored. While op_4 is applying its changes, other operations neither remove nor insert nodes to the left of the left neighbor and to the right of the right neighbor, and also do not change the colors of the nodes in the data set. Claim 12 implies that other operations do not change the color of an additional right neighbor (m_6). Moreover, the left neighbor (m_2) is legally colored even if its color is changed by another operation, while op_4 is applying its changes. We omit the detailed description of this operation as it follows the lines of the description of the *PopLeft* operation.

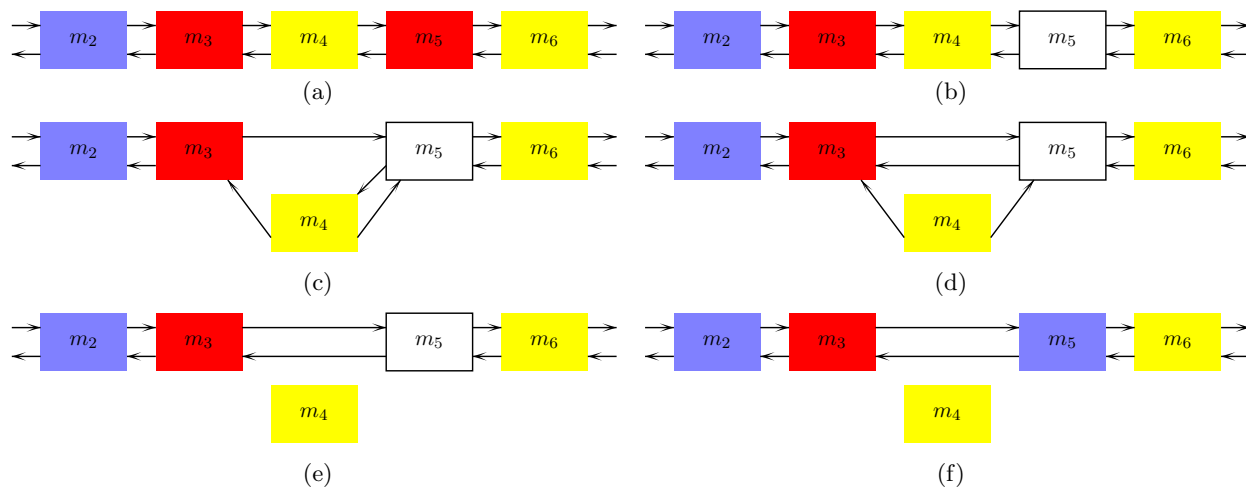


Figure 11: An example of an execution of a *Remove* operation— op_4 from Figure 3.