NON-BLOCKING DATA STRUCTURES HANDLING MULTIPLE CHANGES ATOMICALLY

NILOUFAR SHAFIEI

A DISSERTATION SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
JULY 2015

Abstract

Here, we propose a new approach to design non-blocking algorithms that can apply multiple changes to a shared data structure atomically using Compare&Swap (CAS) instructions. We applied our approach to two data structures, doubly-linked lists and Patricia tries. In our implementations, only update operations perform CAS instructions; operations other than updates perform only reads of shared memory.

Our doubly-linked list implements a novel specification that is designed to make it easy to use as a black box in a concurrent setting. In our doubly-linked list implementation, each process accesses the list via a *cursor*, which is an object in the process's local memory that is located at an item in the list. Our specification describes how updates affect cursors and how a process gets feedback about other processes' updates at the location of its cursor. We provide a detailed proof of correctness for our list implementation. We also give an amortized analysis for our list implementation, which is the first upper bound on amortized time complexity that has been proved for a concurrent doubly-linked list. In addition, we evaluate

our list algorithms on a multi-core system empirically to show that they are scalable in practice.

Our non-blocking Patricia trie implementation stores a set of keys, represented as bit strings, and allows processes to concurrently insert, delete and find keys. In addition, our implementation supports the REPLACE operation, which deletes a key from the trie and adds a new key to the trie simultaneously. Since the correctness proof of our trie is similar to the correctness proof of our list implementation, we only provide a sketch of a correctness proof of our trie implementation here. We empirically evaluate our trie and compare our trie to some existing set implementations. Our empirical results show that our Patricia trie implementation consistently performs well under different scenarios.

I dedicated this thesis to my son, Ryan, who brings endless joy to our lives.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Eric Ruppert, for his tremendous guidance, motivation and support over years and for understanding my complicated life. He taught me how to conduct research and approach a problem from different points of views and how to be precise in conducting research. I believe these skills are also very useful throughout life. I would also like to thank Frank Van Breugel for great comments that he provided during writing the thesis as a supervisory committee member. Besides, I thank Rachid Geurraoui, Patrick Dymond, Nantel Bergeron and Suprakash Datta for agreeing to be on the committee and giving great comments to improve the thesis. I thank Trevor Brown for providing lots of help and code for experiments in Section 9.4 and Michael L. Scott for giving us access to his multicore machines.

At last but not least, I thank my family. I thank my mother, Forough, for supporting me in any way she could, so I was able to spend more time on conducting research and writing the thesis. Besides, I thank my husband, Ali, my father,

Mansour and my sister, Nastartan, for their support and encouragement.

Table of Contents

List of Tables

List of Figures

1 Introduction

The first computers were developed with a single central processing unit. Nowadays, computers are increasingly turning to *multi-core* architectures, which implement multiprocessor systems within a single computer. Multi-core systems typically make use of *shared memory*; shared memory stores information that can be accessed concurrently by multiple processes. Therefore, it is crucial to design algorithms that can access shared data efficiently.

Since the steps of concurrent processes might interleave with one another, accessing shared memory is more complex than private memory. For example, suppose two processes, $p1$ and $p2$, would each like to increment a shared register x which is initially 0. If x is a private variable, a process would simply read it, add one to the value, and write it back. If, however, x is a shared variable, problems can occur if processes follow this simple procedure. First, suppose $p1$ and $p2$ read 0 from x. Next, $p1$ and $p2$ increment the local value they have for x. After that, $p1$ and then $p2$ both write 1 into x. The final value of x would be 1, but it should be 2,

since two processes were supposed to increment x. To avoid such problems, some synchronization must be used when concurrent processes access shared memory. In this thesis, we are concerned with implementing high-level shared data structures from the primitive shared objects that are typically provided in multi-core systems.

A lock is a traditional synchronization mechanism used to implement shared data structures. A lock is a shared object that can be acquired and then released by each process. An exclusive-access lock ensures that when a process p acquires the lock, no other process can acquire it until p releases it. (There are also locks with more complex semantics, like read-write locks.) Processes must acquire one or more locks to modify or access the data structure to ensure that operations on the data structure that might interfere with one another cannot run simultaneously. In some cases, synchronization using locks has the advantage of simplicity. However, this simplicity often comes at the cost of reduced concurrency. To increase concurrency, locks can be used to protect smaller parts of the data structure, so that processes can access or modify different parts of the data structure concurrently. However, having locks on smaller parts of the code may increase the overhead that is caused by acquiring and releasing locks.

Lock-based implementations have the following drawbacks. The link between locks and the data that it protects often exists only in the programmer's mind and it is typically not expressed in the code itself, which can make lock-based code difficult

to maintain. Locks are blocking: if several processes need to acquire the same lock, only one succeeds and the others must wait until the lock is released. Thus, any delay of a process holding a lock can cause performance problems. Possible sources of delay include process pre-emptions, page faults, remote memory accesses, and cache misses. Moreover, locks may cause priority inversion: A high-priority process might have to wait for a low-priority process to release a lock. Locks are not fault-tolerant: if the process holding a lock crashes, the whole system might stop making progress. Locks may cause *convoying*: suppose two or more processes with the same priority repeatedly attempt to obtain the same sequence of locks. While one process acquires a lock, others wait to acquire the same lock. Each time a process attempts to acquire the lock and fails, it has to do a context switch and does not use the rest of its scheduling quantum. The repeated context switches degrade the performance of the whole system.

There are other approaches for implementing shared data structures that do not use locks. In such implementations, one process may *help* to complete operations performed by other processes before performing its own operation. These types of implementations can be designed to satisfy various progress properties. *Non-blocking* [1] implementations guarantee some process completes its operation in a finite number of steps regardless of other processes' delays and failures. In a

[1]The non-blocking progress property is also sometimes called lock-freedom. In Chapter 4, we provide a more formal definition of this and other terms described informally in this chapter.

non-blocking implementation, the system as a whole is always making progress, but individual processes might *starve*; a process might not make progress as long as some other process is making progress. An implementation is *wait-free* if each process completes its operation in a finite number of steps regardless of other processes' delays and failures. Thus, no process starves in a wait-free implementation. However, non-blocking implementations usually make wait-free progress in practice [1].

All non-blocking and wait-free algorithms guarantee some progress property regardless of processes' delays and failures. They all provide fault-tolerance and robust performance even when faced with arbitrary process delays. However, they are often subtle and complex and it is not easy to design an efficient non-blocking or wait-free implementation and verify its correctness. In recent years, a lot of work has been done on designing non-blocking and wait-free algorithms.

Herlihy [25] showed that atomic read/write registers are not sufficient to construct non-blocking implementations of many simple shared data structures. He also showed that if atomic Compare&Swap (CAS) instructions are available in addition to read and write instructions, then *any* data structure can be implemented in a wait-free manner. A CAS(x, *old*, *new*) instruction returns false if the value of variable x is not equal to the expected value *old*. Otherwise, it changes the value of variable x from the expected value *old* to some new value *new* and returns true.

Most modern shared-memory systems support CAS instructions.

The correctness of algorithms using CAS often depends on the fact that, if the CAS succeeds, the value has not been changed since the preceding read. So, the *ABA problem* can sometimes arise when using CAS: Suppose the value of a variable x might be changed from the expected value A to another value B, and then set back to the expected value A just before the CAS occurs. Then, a CAS(x, A, C) can succeed when it is not supposed to. To avoid the ABA problem, different strategies can be used. (We mention some of them in Chapter 3.)

There are two general techniques for obtaining non-blocking or wait-free data structures: universal constructions and transactional memory. Universal constructions [25] transform sequential algorithms into non-blocking or wait-free algorithms. Transactional memory [42] guarantees that a sequence of reads and writes to shared memory is performed atomically. These two general schemes simplify designing concurrent implementations of shared data structures. However, such general techniques are usually less efficient than implementations designed for a specific data structure.

It is not feasible to test all possible executions of a non-blocking or wait-free implementation of a shared data structure to verify its correctness. Thus, to verify the correctness of these complex algorithms, correctness proofs are essential. A shared data structure implements an *abstract data type* (ADT). An ADT is usually

defined using a *sequential specification*, which describes how each operation changes the data structure and the result the operation should return if it is executed without concurrent interference from other operations. An ADT has a state that can be modified by *update* operations. Any operation other than an update is called a *query* and does not change the state of the ADT.

The most commonly used correctness condition for a shared data structure that implements an ADT is *linearizability* [26]. This correctness condition makes it easy for programmers to reason about and use the data structure. An implementation is linearizable if each operation takes effect instantaneously between the time when it is invoked and the time when it returns a result. (A more formal definition is given in Section 4.) Such an instant is called the *linearization point* of the operation. One way to show an implementation is correct is to assign a linearization point to each operation. After that, it must be shown that, during any execution, each operation returns the same result as it would return when all of the operations are applied to the ADT at their linearization points.

An *empirical evaluation* on multi-core machines can provide some information about the practicality and scalability of a shared data structure implementation. However, empirical evaluations only assess the performance of the implementation on some specific machines using some specific benchmarks. So, they usually do not provide a comprehensive assessment of the implementation. Since processes might

starve in non-blocking implementations, the implementation as a whole should be evaluated. One way to analyze the implementation as a whole is an *amortized analysis*. To analyze the amortized complexity of an implementation, all possible finite sequences of operations are considered and a worst-case bound on the total number of steps by all operations is computed. However, the amortized analysis of non-blocking implementations can be complex and only a few have been done.

1.1 Contributions

We give a new approach to design non-blocking, linearizable implementations of shared data structures whose operations can make multiple changes atomically. CAS instructions are employed in our approach. We apply our approach to two data structures, doubly-linked lists and Patricia tries. Both implementations are linearizable. In our implementations, if all pending updates are at disjoint parts of the data structure, they do not interfere with one another. In each implementation, we design one fairly simple routine that is called to perform the real work of all update operations. In contrast, different update operations in previous data structures such as [14] are handled by totally separate routines. This makes our proofs of correctness more modular than the proof of [14]. Our techniques and correctness proof can also be generalized to other data structures. Due to the modularity of our approach, the proof of correctness for the doubly-linked list and Patricia trie,

although complex, are quite similar. So, we provide a detailed proof of correctness only for the doubly-linked list in this thesis and sketch the proof for the Patricia trie. (A detailed proof for the Patricia trie is available in [40].)

In our implementations, one of the main challenges is to ensure the multiple changes required by an update appear to occur atomically. In our approach, we consider operations that make two changes to the data structure, but the operation is linearized at the first such change. Between these two changes, the data structure is temporarily inconsistent. We design a mechanism for detecting such inconsistencies and concurrent operations that detect the inconsistency behave as if the second change has already occurred. Using this mechanism, operations other than updates are performed without altering the shared memory: they do not help updates and they can be performed very efficiently. This is a desirable property since updates are less common than queries in many applications. For simplicity, we assume the existence of a garbage collector (such as the one provided in Java) that deallocates objects that are no longer reachable.

A list stores a sequence of items. The linked list is one of the most fundamental data structures and has many applications in distributed systems including processor scheduling [15], memory management [37] and sparse matrix computations [28]. It is also used as a building block for more complicated data structures such as deques, skip lists and Fibonacci heaps. In some applications such as scheduling, items

must be inserted in such a way that the items of the list are always in sorted order. In our doubly-linked list implementation, a process accesses the list via a *cursor*, which is an object in the process's local memory that locates an item in the list. Our doubly-linked list implementation supports two move operations: MoveRight and MoveLeft, and two update operations: InsertBefore and Delete. The MoveRight(c) and MoveLeft(c) operation move the cursor c to the adjacent item in either direction. The InsertBefore(c, x) operation inserts an item x into the list before the item at which the cursor c is located. The Delete(c) operation removes the item at which the cursor c is located. It is very straightforward to implement similar updates such as an InsertAfter or a Replace operation (that replaces an item of the list by another one).

We give a novel specification that describes how updates affect cursors and how a process gets feedback about other processes' updates at the location of its cursor. This interface makes the list easy to use as a black box. In particular, our list implementation can easily be used to maintain a list in sorted order.

We give an amortized analysis of our doubly-linked list implementation (excluding garbage collection). This is the first amortized analysis for a non-blocking doubly-linked list. Some parts of our analysis are similar to the amortized analysis of non-blocking trees in [12], which used a combination of an aggregate analysis and the accounting method. Here, we simplified the argument using the potential

method. Let $\dot{c}(op)$ be the maximum number of active cursors at any one time during the operation op. The amortized complexity of each operation op is $O(\dot{c}(op))$ for updates and $O(1)$ for moves. We also conduct a preliminary empirical evaluation that shows our doubly-linked list implementation is scalable in a multi-core system.

A Patricia trie [35] is a tree that stores a set of keys, which are represented as bit strings. The trie is structured so that the path from the root to a key is determined by the sequence of characters in the key. The simplicity of the data structure makes it a good candidate for concurrent implementations. Patricia tries are widely used in practice. For example, they have applications in routing systems [20], data mining [38], machine learning [17], bioinformatics [5]. Allowing concurrent access is essential in some applications and can boost efficiency in multi-core systems. Our trie implementation supports a wait-free Find operation and three non-blocking update operations: Insert, Delete and Replace. The Find(k) operation returns true if the trie includes the key k; otherwise, it returns false. The Insert(k) operation adds the key k to the trie and the Delete(k) operation removes the key k from the trie. The Replace(k, k') operation makes two changes to the trie atomically: it removes the key k from the trie and adds the key k' to the trie.

A Patricia trie can be used to store a set of points in $\mathbb{R}^d$. For example, a point in $\mathbb{R}^2$ whose coordinates are (x, y) can be represented by a key formed by interleaving the bits of x and y. (This yields a data structure very similar to a quadtree.) Then,

the Replace operation can be used to move a point from one location to another atomically. This operation has applications in Geographic Information Systems [18] that must keep track of the locations of moving objects. The Replace operation would also be useful if the Patricia trie were used to implement a priority queue, so that one could change the priority of an element in the queue.

Search trees are another class of data structures that are commonly used to represent sets. When keys are not uniformly distributed, balanced search trees generally outperform unbalanced ones. The reverse is often true when keys are uniformly distributed due to the simplicity of unbalanced search trees. Our empirical results show that the performance of our trie is consistently good in both scenarios. This is because our trie implementation is as simple as an unbalanced search tree but also keeps trees short since its height is at most the length of the key (and will often be shorter).

To summarize our contributions:

- We provide a general technique to design a non-blocking linearizable implementation of shared data structure that can apply multiple changes to the data structure atomically using single-word CAS.

- We apply our approach to doubly-linked lists and Patricia tries.

- We provide implementations and proofs that are modular and can be adapted

for other data structures.

- In both implementations, one routine is employed to implement the real work of all update operations.
- In our list implementation, cursors are updated and moved by only reading the shared memory.
- The cursors provided by our list implementation are robust: they can be used to traverse and update the list, even as concurrent operations modify the list.
- Our list implementation can easily be used to maintain a sorted list.
- In our list implementation, the amortized number of steps performed by each update op is $O(\dot{c}(op))$ and each move is $O(1)$.
- We evaluate our list implementation on a multi-core machine.
- Searches in our Patricia trie are wait-free.
- We compare our Patricia trie implementation empirically to other existing concurrent data structures and our results show that our trie performs consistently well when the keys are uniformly or non-uniformly distributed.
- Our trie implementation supports the Replace operation that can be used in Geographical Information Systems to implement moving objects.

Our Patricia trie implementation has been published in [41].

2 Simple Examples of Non-blocking Implementations

Non-blocking implementations are often subtle and complex. Here, we present two simple examples to introduce some ideas that are used in non-blocking implementations of data structures from CAS instructions.

2.1 Non-blocking Stack Implementation

Here, we describe a non-blocking stack implementation. The stack is represented as a singly-linked list with a *Top* pointer. (See Figure 2.1.) The direction of the links is from the top node to the bottom node in the stack and the *Top* pointer always points to the top node in the stack. The implementation supports the POP operation, which removes the top node from the stack, and the PUSH operation, which adds a node to the top of the stack. The goal is to maintain the invariant that the elements on the stack are the elements of the singly-linked list, starting from the node that *Top* points to, in order.

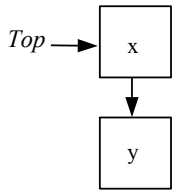


Figure 2.1: The stack containing x and y

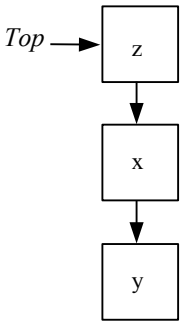


Figure 2.2: The stack after the PUSH operation

If atomic read and write instructions are used to change the *Top* pointer, concurrent operations might change the *Top* pointer incorrectly. For instance, suppose the stack initially contains two nodes whose values are x and y as in Figure 2.1. Suppose a PUSH operation tries to push a new node whose value is z onto the stack and a POP operation concurrently tries to pop the top node from the stack. Both operations could read x from *Top* and then the PUSH operation would link the new node whose value is z to the top node and write a pointer to z into *Top*. (See Figure 2.2.) After that, the POP operation writes a pointer to y into *Top*. (See Figure 2.3.) Thus, after performing the PUSH and POP operation, the stack incorrectly contains only one node and this violates the stack invariant described above.

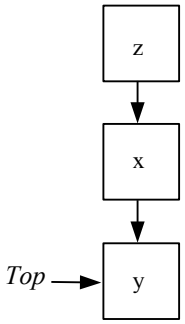


Figure 2.3: The stack after the POP operation

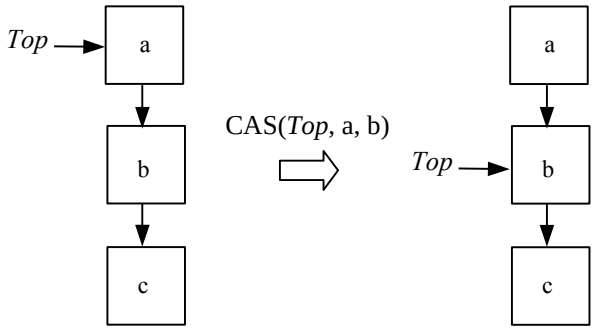


Figure 2.4: The POP operation

In fact, it turns out to be impossible to implement a stack using only reads and writes [25]. Thus, non-blocking stack implementations must use stronger primitives like CAS.

Next, we describe the shared stack implementation of Treiber [46]. A CAS instruction is used to modify the value of the *Top* pointer atomically. If the *Top* pointer is not null, the POP operation uses a CAS to swing the *Top* pointer to the second node from the top of the stack. (See Figure 2.4.) If the CAS succeeds, the POP operation is linearized at this CAS step. Linearizing at this point maintains the

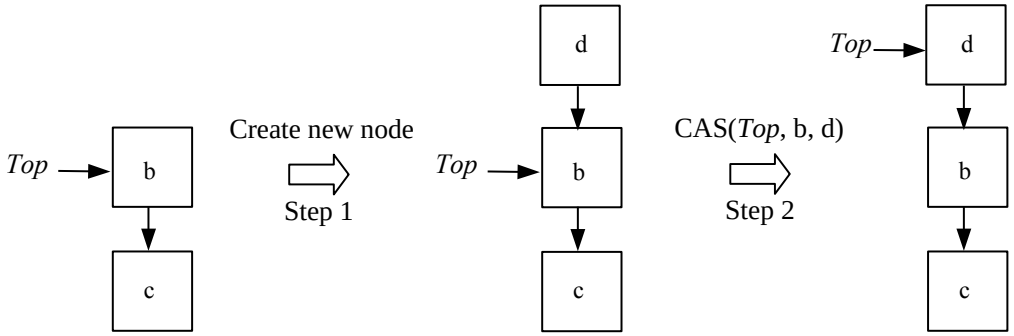


Figure 2.5: The PUSH operation

stack invariant described above. If the CAS fails, the operation retries. If the Top pointer is null, the POP operation returns empty. In this case, the POP operation is linearized at the step that reads the Top pointer (since the stack is empty at that step). The PUSH operation creates a new node that points to the top node in the stack and then uses a CAS to swing the Top pointer to the new node. (See Figure 2.5.) The PUSH operation is linearized at the CAS step that successfully changes the Top pointer. Linearizing at this point maintains the stack invariant described above. Again, if the CAS fails, the operation retries. Pseudo-code of the shared stack implementation is presented in Figure 2.6.

Next, we analyze the amortized complexity of the stack implementation. The goal is to bound the total number of steps taken in a finite execution α. Suppose op is a POP or PUSH operation. Let $\dot{c}(op)$ be the maximum number of operations running at any one time during op. We show that the amortized number of steps per operation op is $O(\dot{c}(op))$. More precisely, we show that the total number of

```
1.  type Node
2.      Value value
3.      Node nxt

4.  Initialization
5.  Top ← null

6.  Pop(): {Value, empty}
7.      do
8.          t ← Top
9.          if (t = null) then                    ▷ the linearization point if returns empty
10.             return empty
11.         x ← t.nxt
12.     while (CAS(Top, t, x) = false)            ▷ the linearization point if CAS succeeds
13.     return t.value

14. Push(val: Value): {true}
15.     x ← new Node(val, null)
16.     do
17.         t ← Top
18.         x.nxt ← t
19.     while (CAS(Top, t, x) = false)            ▷ the linearization point if CAS succeeds
20.     return true
```

Figure 2.6: The pseudo-code of the non-blocking stack implementation

steps taken in α is bounded by $O(\sum_{op \text{ in } \alpha} \dot{c}(op))$. Since the number of steps that op takes inside the loop iteration 7–12 or 16–19 is constant, we count the number of iterations of the loop that op takes. If the CAS step of op on line 12 or 19 fails, it means some other concurrent operation op' successfully changed the Top pointer during that loop iteration. So, that successful operation op' pays for all of the iterations that fail as a result of the successful CAS step of op'. There are at most $\dot{c}(op')$ such iterations, since all of them are running at the time of the successful CAS step of op'. Since only during the last iteration, the CAS on line 12 or 19

might succeed, each operation op that successfully changes Top pays $\dot{c}(op)$. So, the amortized complexity of op is $O(\dot{c}(op))$.

2.2 Non-blocking Queue Implementation

Here, we present the non-blocking queue implementation of Michael and Scott from CAS instructions [34] as another example. Their queue is implemented as a singly-linked list with a shared $Head$ and a $Tail$ pointer. The direction of the links is from the $Head$ to the last node in the list. The goal is to maintain the invariant that the elements in the queue are the elements of the singly-linked list, starting with the node that $Head$ points to and ending with the node that $Tail$ points to, in order. Their implementation supports the DEQUEUE operation which removes a node from the beginning of the list and the ENQUEUE operation which adds a node to the end of the list. The DEQUEUE operation uses a CAS to swing the $Head$ pointer to the next node in the list. (See Figure 2.7.) The DEQUEUE operation is linearized at the step that successfully changes the $Head$ pointer. Linearizing at this point maintains the queue invariant described above. If two operations are trying to dequeue simultaneously, the use of CAS instructions ensures that only one of them succeeds in swinging the $Head$ pointer; the other one has to restart. The ENQUEUE operation uses a CAS to link a new node to the end of the list and then another CAS to swing the $Tail$ pointer to the new node. (See Figure 2.8.)

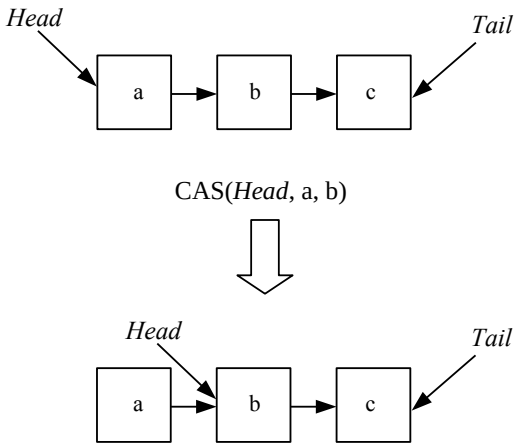


Figure 2.7: The DEQUEUE operation

If the first CAS step of an operation fails, the operation restarts. The ENQUEUE operation is linearized at the step that successfully links the new node to the end of the list. Linearizing at this point maintains the queue invariant described above.

The ENQUEUE operation takes two key steps: linking a node onto the end of the list and updating the $Tail$ pointer. Since the process performing the ENQUEUE could crash or slow down in between performing the two steps, the implementation uses *helping* to ensure that the data structure does not remain in this inconsistent state. If an operation *op* reaches the new node between the two CAS steps of the ENQUEUE, *op* helps the incomplete ENQUEUE by swinging the $Tail$ pointer to the new node using a CAS, and then restarts its own operation. For instance, suppose there are two concurrent ENQUEUE operations and the first ENQUEUE operation

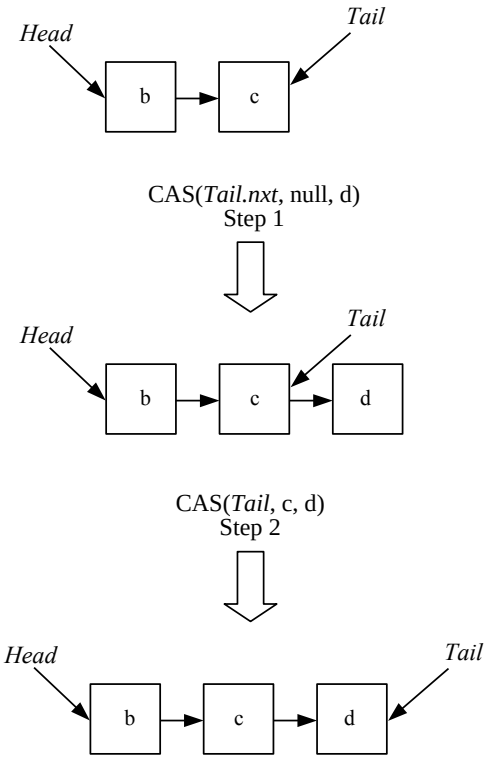


Figure 2.8: The ENQUEUE operation

links a new node whose value is d to the end of the list, but it crashes before swinging the $Tail$ pointer to the new node. (See step 1 of Figure 2.8.) Then, the second ENQUEUE operation helps the first one and swings the $Tail$ pointer to the node whose value is d before performing its own operation.

3 Related Work

Here, we focus on implementations of shared data structures that do not use locks. There are two general techniques for obtaining non-blocking data structures: universal constructions (see [13] for a survey) and transactional memory (see [21] for a survey). Such general techniques are usually not as efficient as algorithms that are designed for specific data structures. (For example, our empirical results also show that a doubly-linked list implementation using transactional memory does not scale.)

Turek, Shasha and Prakash [48] and Barnes [4] introduced a technique in which processes cooperate to complete operations to ensure non-blocking progress. Each update operation creates a descriptor object that contains information that other processes can use to help complete the update. This *cooperative technique* has been used for various data structures, including several described in this chapter.

We are interested in implementing data structures from atomic CAS, read, and write instructions. In addition, the pair *Load Linked* (LL) and *Store Conditional*

(SC) are a useful building block that is used in some of the implementations described in this chapter. LL(x) reads the value of the variable x and might later be used in combination with SC. A SC(x, a) by a process p writes the value a into the variable x and returns true if no other process has performed a successful SC on x since the last LL(x) executed by p, and otherwise returns false without changing the value of x. Unlike CAS, LL/SC are not subject to the ABA problem. However, many modern systems do not support LL/SC instructions.

3.1 Multi-word Compare&Swap

If stronger instructions than CAS are available, some data structures can be implemented more easily. One of such instruction is a *multi-word Compare&Swap* (k-word CAS) instruction. The k-word CAS($x[1..k]$, $a[1..k]$, $b[1..k]$) atomically sets the variable $x[i]$ to the value $b[i]$ for all i if, for all i, the value of $x[i]$ is $a[i]$. The *multi-word Compare and Single Swap* (k-word CASS) instruction is a more restricted version of k-word CAS. The k-word CASS($x[1..k]$, $a[1..k]$, b) sets the variable $x[1]$ to the value b if, for all i, the value of $x[i]$ is $a[i]$. The k-word CAS and k-word CASS primitives are usually not available in hardware for $k > 1$. However, there are some non-blocking implementations of them from widely available primitives.

Israeli and Rappoport [29] gave a non-blocking implementation of k-word CAS

using LL/SC instructions and show how LL/SC can, in turn, be implemented from CAS instructions in a non-blocking manner. (Each LL and each SC performs one CAS instruction when there is no contention.) They used the cooperative technique in their k-word CAS implementation. Information about each process's latest k-word CAS is stored in a descriptor together with an indicator of the operation's status (in-progress, failed, succeeded or terminated). The k-word CAS operation first uses LL/SC to write its process's id into each of the k locations of $x[1..k]$ if it holds the specified expected value. If two operations concurrently write their processes' ids into the same set of locations in an opposite order, they might never make progress. This is called deadlock. To avoid deadlock, each operation writes its id into locations in order of their memory addresses. If the process reads another process's id from a location, the process attempts to help complete the other process's operation using the information in that operation's descriptor and then continues its own operation. If the operation does not write its id into all k locations successfully, the operation removes its id from all locations and then sets its status to failed using LL/SC. Otherwise, the operation uses LL/SC to set its status to succeeded and then the new values specified by the k-word CAS are written into the k locations using LL/SC steps. Before the operation terminates, it uses LL/SC to set its status to terminated. If there is no contention, an operation uses LL/SC to write its id and then the specified new values into the k locations. It

also uses LL/SC to change its status to succeeded and then terminated. Since each pair of LL/SC takes 2 CAS instructions, when there is no contention, this k-word CAS implementation performs $4k + 4$ CAS instructions. If a read operation reads a process id instead of a value from a location, the read operation first attempts to help complete that process's k-word CAS operation and then returns the value that is written in that location.

Harris, Fraser and Pratt [23] presented a non-blocking k-word CAS using 2-word CASS instructions, also using the cooperative technique. The 2-word CASS primitive is not ordinarily available in hardware. They use CAS instructions to give a non-blocking implementation of 2-word CASS which performs two CAS instructions if there is no contention. In addition to the information required to help complete an operation, the descriptor of the operation also includes a status field. They use 2-word CASS to set each of the k locations to the operation's descriptor object if it holds the specified expected value and the status of the operation is in-progress. If the operation encounters another operation's descriptor object, it attempts to help complete the other operation using the information in the descriptor and then retries. If a location does not hold the specified expected value, the status of the operation is set to failed using a CAS. If the operation successfully sets all k locations to pointers to its descriptor object, it uses a CAS to set its status to succeeded and then replaces the pointers in k locations with the specified new values using

CAS steps. When there is no contention, this implementation performs $3k+1$ CAS steps. If a read operation reads an operation descriptor from a location, it helps that operation to complete using the information in the descriptor and then retries.

Attiya and Hillel [2] used the cooperative technique to give a non-blocking implementation of k-word CAS using CAS and 2-word CAS instructions. The 2-word CAS primitive is not ordinarily available in hardware. To avoid the ABA problem, a single word must store both a pointer and a tag and a CAS step atomically increases the tag when the pointer changes. In their implementation, an operation can *acquire* an object by using a CAS to set a pointer in the object to the operation's descriptor and the operation can *release* an object by using a CAS to set a pointer in the object to null. Each k-word CAS operation first acquires its own descriptor and then each of the k locations. The operation's descriptor also contains a counter that shows the number of locations that have been acquired by the operation and the counter is increased using a CAS after each location is acquired. If the operation successfully acquires all k locations, it uses CAS instructions to make the necessary changes to the k locations. After that, the operation releases each of the k locations. If the operation op encounters a location that is acquired by another operation op', op decides whether to help complete op' or reset op' by comparing how many locations they acquired. If op and op' have acquired the same number of locations, op uses a 2-word CAS to acquire both its own descriptor and the de-

scriptor of op'. If op succeeds, it resets op'. Otherwise, op helps complete op'. This conflict resolving technique avoids deadlock without having to acquire locations in the order of their addresses. The *conflict graph* is an undirected graph, in which vertices represent locations and edges represents operations. If an operation op accesses the locations a and b, the graph includes an edge, labeled op, between the vertices represented a and b . Their implementation ensures an operation op is only delayed due to other operations that are close to op in the conflict graph. When there is no contention, this implementation of a k-word CAS operation performs $4k + 2$ CAS steps.

Sundell [43] presented a wait-free implementation of k-word CAS using CAS steps, also using the cooperative technique. Operation descriptors include a status field whose value is in-progress, give, succeeded or failed. The status field is changed using a CAS step. In his implementation, an operation can acquire a location using a CAS to change its value to a pointer to its descriptor object. The operation first acquires each of k locations that holds the specified expected value. Operations do not need to acquire locations in order of their addresses. If a location does not have the specified expected value, the status of the descriptor is set to failed and then each of k locations that is earlier set to the operation descriptor is set back to the specified expected value using a CAS step. When the operation op cannot acquire any more locations because they are acquired by others, the ids of processes that

initiated the operations are used to resolve the conflict. Suppose a location that op wishes to acquire is already acquired by op'. If the process that initiated op has a lower id than the process that initiated op', op uses a CAS to change the status of op' to give and then acquires any location that it wishes from op'. Otherwise, op tries to help op'. Sundell suggests this resolving scheme can be made wait-free by either cycling processes' ids before an operation is initiated or changing resolution ordering dynamically at run-time. When the operation successfully acquires all k locations, the operation sets its status to succeeded and then uses CAS steps to set the values of k locations to the specified new values. When there is no contention, his implementation performs $2k+1$ CAS steps. If a read operation reads an operation's descriptor from a location, the read operation returns the operation's specified expected value or new value, depending on the status of the descriptor. Table 3.1 shows the number of CAS steps that the k-word CAS implementations described in this chapter take in absence of contention.

A weaker progress condition than the non-blocking property is *obstruction-freedom*. An implementation is obstruction-free if each process completes its operation in a finite number of steps if it takes sufficiently many steps without any other process taking steps. Luchangco, Moir and Shavit [31] presented an obstruction-free implementation of a k-word CASS using LL/SC instructions. They use CAS instructions to give an obstruction free implementation of LL/SC, which adds

k-word CAS implementation	number of CAS steps in absence of contention
Israeli and Rappoport [29]	$4k + 4$
Harris, Fraser and Pratt [23]	$3k + 1$
Attiya and Hillel [2]	$4k + 2$
Sundell [43]	$2k + 1$

Table 3.1: The number of CAS steps that a k-word CAS implementation performs in absence of contention

some overhead. When there is no contention, their implementation performs 2 CAS steps.

3.2 Lists

Valois [49] presented the first non-blocking implementation of a singly-linked list using CAS steps. In his implementation, auxiliary nodes are inserted between adjacent nodes, so the list might be longer than it is supposed to be. This implementation uses a cursor that points to three consecutive nodes in the list. If the part of the list that the cursor is associated with is changed, the cursor becomes invalidated. To restore the validity of its own cursor, a process may have to perform CAS steps to help complete other processes' updates.

Harris [22] presented a non-blocking singly-linked list implementation using CAS

steps. He provided a sketch of a correctness proof. In his implementation, each node includes a key and the next pointer. Before deleting a node, the node's next pointer is marked using a CAS step. There is no support for cursors. At the beginning of each operation, the operation searches from the beginning of the list for a node at which to perform the operation. Michael [32] used Harris's design to present a non-blocking singly-linked list implementation. However, unlike Harris's implementation, his list is compatible with efficient memory management techniques.

Fomitchev and Ruppert [16] also gave a non-blocking singly-linked list implementation using CAS steps. Three CAS steps are performed to remove a node from the list. Similar to [22], each operation searches the list for a node on which to operate. Each node also has a back-link pointer. When a node is deleted, a back-link pointer is set to its last predecessor. If an operation's CAS fails, it uses back-link pointers to resume its operation. However, this back-link pointers cannot be used to move a cursor to left since the back-link pointers of only removed nodes are set. They provided a correctness proof and an amortized analysis for their implementation. The amortized analysis uses a complicated billing scheme.

Timnat et al. [45] used the cooperative technique to present a wait-free singly-linked list using CAS steps. They extended Harris's implementation by using a wait-free helping mechanism: At the beginning, an operation gets a time-stamp

which is used to avoid starvation. Then, the operation is announced in a shared array. After that, the operation goes through the array, helping all operations with a time-stamp lower than or equal to its own.

Non-blocking *doubly*-linked lists have received much less attention. Doubly-linked lists can be implemented using k-word CAS primitives. However, this is not as straightforward as it might seem. Suppose each item is represented by a node with nxt and prv fields that point to the adjacent nodes. Consider a list that has four consecutive nodes, A, B, C and D, as in Figure 3.1. A deletion of C must change $B.nxt$ from C to D and $D.prv$ from C to B, as in Figure 3.2. It is *not* sufficient for the deletion to update these two pointers with a 2-word CAS. If two concurrent deletions remove B and C in this way, $A.nxt$ would be set to C by the deletion of B. (See Figure 3.3.) So, C would still be accessible through A after the two deletions. This problem can be avoided by using 4-CAS to simultaneously update the two pointers and check whether the two pointers of C still point to B and D. Then, the 4-CAS of one of the two concurrent deletions would fail. The 4-CAS works for updating pointers, but it is not obvious how to detect if the item that one process's cursor is located at has been removed by another process. To do this, the multiword CAS may have to operate on even more words. The most efficient k-word CAS implementation [43] uses $2k+1$ CAS steps to change k words when there is no contention. (See Table 3.1.) Thus, at least 9 CAS steps are

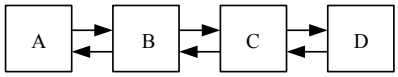


Figure 3.1: A doubly-linked list containing four nodes

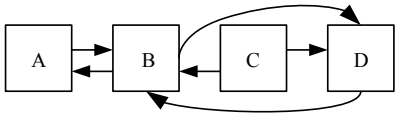


Figure 3.2: After deletion of C

required for 4-CAS. Our implementation uses only 5 CAS steps for contention-free updates.

Only a few non-blocking doubly-linked-list implementations exist. There are two implementations of doubly-linked lists that use 2-word CAS; however, most modern systems do not support 2-word CAS. Greenwald [19] presented one such implementation. In his approach, all processes cooperate to execute a single piece of sequential code. An operation executes a step of the sequential code and increments

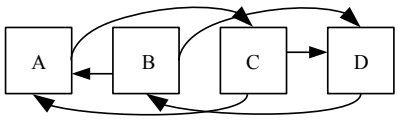


Figure 3.3: Incorrect result of deleting B and C concurrently

a shared counter that is equal to the line number of the running code simultaneously using 2-word CAS. When one operation is completed, each process can try to update the programme counter to the beginning of the operation it wants to perform next. This implementation does not support any concurrency: only one operation can make progress at a time.

Attiya and Hillel [3] also proposed a doubly-linked list implementation using 2-word CAS, but it only supports update operations. Three ordered colours are assigned to nodes in the list such that any two adjacent nodes have distinct colours. In their implementation, each update operation needs to acquire three consecutive nodes by using CAS steps to write its id into the nodes before performing its update. If two of the nodes have the same colour, a 2-word CAS is used to acquire those two nodes. If a process does not acquire a node because it is acquired by another operation, the process first helps the other operation and then continues its own operation. The operation acquires nodes in the order of their colours. So, an operation may help the other operations that acquire the nodes with "higher" colours. The implementation has the nice property that concurrent operations can interfere with one another only if they are changing nodes close to each other. If there is no interference, an operation performs 13 to 15 CAS steps (and one 2-word CAS). To avoid the ABA problem, a single word must store both a pointer and a counter. If an operation is called with a cursor located at a deleted node, the operation termi-

Implementation	supports cursors?	operations to move cursor?	recover a cursor's location?	primitive used	# of CAS with no contention
Greenwald [19]	No	-	-	2-CAS	depends on size of list
Attiya and Hillel [3]	Yes	No	No	2-CAS	13-15
Sundell and Tsigas [44]	Yes	Yes (CAS used)	Yes (CAS used)	CAS	2-4
List presented here	Yes	Yes (No CAS)	Yes (No CAS)	CAS	5

Table 3.2: Implementations of doubly-linked lists

nates without updating the cursor. So, deletions might make other processes lose their place in the list. They also give a restricted implementation using single-word CAS, in which deletions can be performed only at the ends of the list.

If an update removes the item where a cursor c is located, an operation called with c first needs to *recover c's location* in the list. In our implementation, cursors are moved and cursors' locations are recovered without performing any CAS steps. None of the implementations that use k-CAS handle cursors with the same functionality as ours. (See Table 3.2.) Besides, they all perform many CAS steps when there is no contention. Our implementation performs only 5 CAS steps for a contention-free update.

Sundell and Tsigas [44] gave the first non-blocking doubly-linked list using single-word CAS (although a word must store both a pointer and a bit). As in our implementation, cursors are used to access the list but their specification of cursors is somewhat different. Linearizable data structures are notoriously difficult to design, so detailed correctness proofs are essential. In [44], a proof of the non-blocking property is provided, but to justify the claim of linearizability, the linearization points of operations are defined without providing a proof that they are correct. In fact, their implementation appears to have some minor errors: using the Java PathFinder model checker [50], we discovered an execution that incorrectly dereferences a null pointer. Their implementation is ingenious but quite complicated. In particular, their helping mechanism is very complex, partly because update operations can terminate before completing the necessary changes to the list, so operations may have to help non-concurrent updates. Similar to our implementation, theirs supports operations to move cursors. In our doubly-linked list implementation, an update helps only updates that are concurrent with itself, and operations that move a cursor do not help at all. In the best case, their updates perform 2 to 4 CAS steps. However, moves perform CAS steps to help complete updates. In fact, a series of deletions can construct a long chain of deleted nodes whose pointers to adjacent nodes do not get updated by the deletions. Then, a move operation may have to traverse this chain, performing CAS steps at every node.

3.3 Trees

Here, we review some tree-based data structures that do not use locks. Tsay and Li [47] gave a general wait-free construction for tree-based data structures. To access a node, a process makes a local copy of the path from the root to the node, performs computations on the local copy, and then atomically replaces the entire path by the local copy by swinging a pointer to the root of the tree. Since this approach copies many nodes and causes high contention at the root, their approach is not very efficient.

Ellen et al. [14] presented a non-blocking binary search tree data structure from CAS steps. Their approach has some similarity to the cooperative technique of [48] and [4]. Each update operation creates a descriptor object that contains information that other processes can use to help complete the update. Then, for each node that would be removed from the tree or whose children would be changed, the update sets a pointer in the node to the descriptor object. After that, the operation applies its changes to the tree without interference with other operations. If any other operation sees that a node points to an update descriptor object, the operation uses the information in the descriptor object to help to complete the update. In [14], modifications were only made at the leaves of the search tree. Our new Patricia trie implementation also copes with modifications that can occur anywhere in the

trie. This requires proving that changes in the middle of the trie do not cause concurrent search operations passing through the modified nodes to go down the wrong branch.

Brown and Helga [10] generalized the binary search trees of [14] to non-blocking k-ary search trees. Brown et al. [8, 9] also extended the approach used in [14] to propose a new set of primitive operations to implement non-blocking data structures. Their approach simplifies designing non-blocking data structures but can only handle one change to the data structure atomically. Our technique is the first one that extends the approach used in [14] to handle multiple changes to a data structure atomically.

Howley and Jones [27] presented a non-blocking search tree from CAS operations using a cooperative technique similar to [14]. Their tree stores keys in both leaves and internal nodes. However, search operations sometimes perform CAS steps to help update operations. Their implementation introduced changes in the middle of a search tree but only to keys stored in internal nodes, not the structure of the tree itself. Braginsky and Petrank [6] proposed a non-blocking balanced B+tree from CAS operations. To rebalance the tree, updates may split, join or copy leaf nodes. They extend the techniques in [22] that marks pointers before deletion and [14] that sets the state of nodes before deletion.

Prokopec et al. [39] described a non-blocking hash trie that uses CAS steps. In

their implementation, nodes have up to 2^k children (where k is a parameter) and extra intermediate nodes are inserted between the actual nodes of the trie. Unlike our trie implementation, their search operation may perform CAS steps. Oshman and Shavit [36] sketched a non-blocking trie implementation that also maintains a doubly-linked-list and skip list at the bottom of the trie.

4 Formal Definitions

An abstract data type (ADT) is usually defined using a *sequential specification* that consists of the following parts.

- a set Q of possible states,
- the initial state, $q_0 \in Q$,
- a set OP of operations that the ADT supports,
- a set RES of possible responses that operations can return, and
- a transition relation $\delta \subseteq Q \times OP \times RES \times Q$.

If $(q,\ op,\ res,\ q') \in \delta$, operation op can change the state of the ADT from state q to state q' and return the response res. An operation $op \in OP$ is called a *query* operation if, for each $(q,\ op,\ res,\ q') \in \delta$, $q = q'$. Otherwise, op is an *update* operation.

We describe the queue ADT as an example. A state q in Q is a finite sequence $\langle v_1, v_2, ..., v_k \rangle$. The state q_0 is $\langle\rangle$. The set OP includes ENQUEUE(v), and DEQUEUE

```
1. CAS(x: Variable, old: Value, new: Value): Boolean
2.     if x = old then
3.         x ← new
4.         return true
5.     else return false
```

Figure 4.1: Steps of CAS

and the set RES includes ack, $empty$ and a value v. The transition relation δ consists of all tuples of the form

$(\langle v_1, v_2, ..., v_k \rangle$, ENQUEUE$(v)$, ack, $\langle v_1, v_2, ..., v_k, v \rangle)$,

$(\langle v_1, v_2, ..., v_k \rangle$, DEQUEUE, v_1, $\langle v_2, ..., v_k \rangle)$, and

$(\langle\rangle$, DEQUEUE, $empty$, $\langle\rangle)$.

In this thesis, we consider implementations of shared data structures. An implementation of an ADT is a piece of code that each process can execute for each of the ADT's operation. An implementation also specifies initial values of all shared and local variables. The code for each operation can access local variables and shared variables using atomic read, write and CAS instructions. Figure 4.1 shows the steps that a CAS takes in a single atomic action.

A *configuration* of an implementation at some time consists of the values of all parts of the shared memory and of the local variables of all processes at that time. The local variables include the program counter of each operation that is active.

A step of an implementation is some part of the code of the implementation that is performed by one process and includes at most one access to the shared memory. An *execution of an implementation* is a sequence of configurations, $C_0, C_1, ..., C_t$ such that, for every $0 \leq i < t$, C_{i+1} follows from C_i by a step of the implementation, and all variables have the initial values specified by the implementation in C_0. Thus, the model of computation is completely *asynchronous* since we consider all possible executions formed by interleaving individual steps of processes.

The *invocation* of an operation is the step when a process calls an operation and the *response* of an operation is the step when an operation terminates. A *history* is a sequence of invocations and responses of operations. A history H is *well-formed* if the subsequence of H corresponding to each process starts with an invocation and alternates between invocations and matching responses. We assume that all histories are well-formed. A history is *sequential* if it starts with an invocation, and each invocation (except possibly the last one) is immediately followed by the matching response, and each response (except possibly the last one) is immediately followed by an invocation. A sequential history S is *legal* if, for each data structure ds, the subsequence op_1, res_1, op_2, res_2, ... that is performed on ds satisfies the sequential specification of ds. More precisely, there exists q_1, q_2,... in Q such that $(q_{i-1}, op_i, res_i, q_i) \in \delta$ for all i, and q_1 is the initial state given in the sequential specification.

An operation is *complete* in a history H if H includes both the invocation and response of the operation. A history H is *linearizable* if there is a sequential history S that includes all complete operations in H and some subset of the incomplete ones whose invocations are in H satisfying the following three conditions.

- Each operation that terminates in H returns the same response in S.
- S is legal.
- If the response of some operation op in H is before the invocation of another operation op' in H and both appears in S, then op precedes op' in S.

An implementation of an ADT is *linearizable* if, for each execution of the implementation, the subsequence of operations' invocations and responses in the execution is a linearizable history. One way to show an implementation is linearizable is to assign a *linearization point* to each operation. A linearization point of an operation is an instant between its invocation and its response. (We say the operation is *linearized at* that instant.) Then, it must be proved that operations would return the same response if each operation is performed atomically at its linearization point.

We present two histories of a queue data structure as examples. Both consist of three operations: enqueue(v), enqueue(v') and dequeue, denoted $op1$, $op2$ and $op3$ respectively. Let inv_{op} be the invocation of operation op and let res_{op} be the

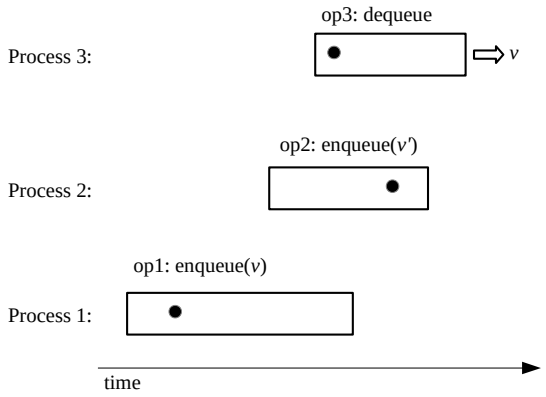


Figure 4.2: An execution that corresponds to the history H

(Each rectangle represents an operation and each dot inside a rectangle shows the linearization point of the operation.)

response of op. Let H be the sequence, inv_{op1}, inv_{op2}, inv_{op3}, res_{op1}, res_{op2} and res_{op3} and $op3$ returns v. Figure 4.2 shows an execution that corresponds to H. In Figure 4.2, each operation represented by a rectangle and each dot inside a rectangle shows the linearization point of the operation. Then, H is linearizable (since each operation returns a response as all operations are performed atomically at their linearization points). Let H' be the sequence, inv_{op1}, inv_{op2}, res_{op1}, res_{op2}, inv_{op3} and res_{op3} and $op3$ returns $empty$. Figure 4.3 shows an execution that corresponds to H'. H' is not linearizable (since $op3$ is invoked after $op1$ and $op2$ are terminated and v and v' are enqueued and then $op3$ returns $empty$).

An implementation is *non-blocking* if there is no infinite execution of the implementation such that the execution has a configuration after which some processes

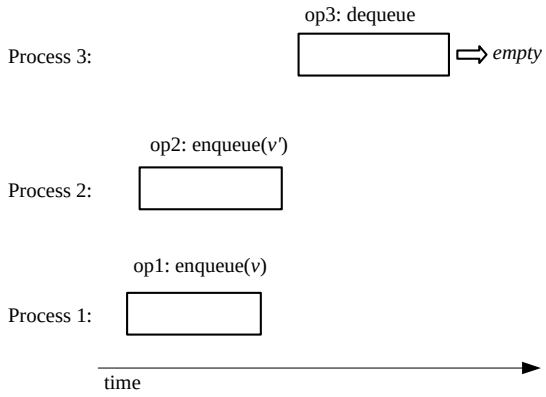


Figure 4.3: An execution that corresponds to the history H'

take infinitely many steps and no operation terminates.

5 General Approach

In this chapter, we describe our general approach to implement a non-blocking linearizable data structure. Our approach can handle multiple changes to the data structure atomically. We assume that the data structure can be partitioned into smaller units of data such as the nodes in a linked list or in a tree. We refer to these units as nodes in the following.

Nodes are represented by Node objects. Each Node object contains the following parts: the data that the data structure puts in the node, including pointers to other nodes, and a field *info* that stores a pointer to an operation descriptor object.

First, we describe the structure of update operations. When an update operation *op* is called, *op* first searches the data structure to determine which nodes would be *affected* by the update. We say a node would be affected by an update if at least one of its fields would be changed or the node would be deleted from the data structure by the update. Before applying the update to the data structure, *op* must *flag* affected nodes by setting their *info* pointers to an operation descriptor

object that describes the update. This is done as follows.

After finding affected nodes, *op* checks whether any of them are already flagged by another concurrent update. If so, *op* tries to help the concurrent update to complete and restarts from scratch. In our approach, only update operations help each other to complete their updates and query operations do not need to help any other operation.

If none of *op*'s affected nodes are flagged by others, *op* creates an operation descriptor object that describes all the changes required to perform the update. After creating of the descriptor object, *op* flags its affected nodes by setting their *info* fields to the descriptor object using CAS steps. Like locking, flagging a node is used to give an operation exclusive permission to change the fields of that node. Unlike locks, the descriptor object stores enough information, so that if a process performing an operation crashes while nodes are flagged for it, other processes can attempt to complete the operation and remove the flags.

If all nodes are successfully flagged, *op* changes the fields of some nodes using CAS steps to perform the update. The ABA problem must be avoided when our technique is applied to implement a data structure. There are various ways to do this. One way to do this for fields that store pointers is to ensure that whenever a pointer is updated from an old value to a new value, the old value is removed from the data structure and never used again. In our approach, to change a field v of a

node whose value is not a pointer, to a new value, one could replace the node with a new copy of the node whose field v is set to the new value.

After making all changes to the data structure, *op unflags* the flagged nodes that are still in the data structure. Nodes that have been removed from the data structure by one operation can never be flagged for another operation. Nodes can be unflagged by setting their pointers to empty descriptor objects using CAS steps. Another way to unflag nodes is setting a bit in the descriptor object. There are trade-offs between these two approaches. Unflagging using the second approach is less expensive since it sets a bit instead of using a CAS step on each node. However, to check whether a node is flagged, a bit in the descriptor that the node points to must be read. Besides, the second approach might make chains of pointers from the nodes in the data structure to the nodes that are removed from the data structure; this is because even after the operation terminates, all of its affected nodes might still point to the operation's descriptor, which points to nodes that have been removed from the data structure. This might cause memory management problems since the memory of removed nodes cannot easily be freed for reuse. For example, suppose an update flags two nodes x and y with a descriptor object I and then removes y from the data structure and after that, unflags x and y by setting a bit in I. Since *x.info* might be still point to I after the update terminates and a field of I points to y, there might be a chain of pointers from x to y after the update

removes y from the data structure. So, y might not be garbage collected after it is removed. One solution to this problem is to set all pointers in the descriptor object to null after the update is completed. Then, to avoid dereferencing null pointers, processes read a field of an operation descriptor only if the operation is not yet completed. The descriptor object also contains the status of the operation that it describes and this information can be used to determine whether the operation is completed or not.

If *op* does not flag all its affected nodes successfully, *op* unflags the nodes that *op* already flagged and restarts from scratch.

Each update operation might make more than one change to the data structure. We say an update operation is *committed* when the operation successfully makes its first change to the data structure. Any query operation only reads the shared memory and does not help any update operation or change the state of the data structure. If an operation op' reaches a node that is already removed from the data structure, op' can determine this by checking whether the update that was trying to remove the node is committed or not. If the update is already committed, op' behaves like the node has been removed from the data structure. Suppose an update operation is committed when a field of x is changed from v_{old} to a new value. One way to determine if the update has been committed or not is to check whether the field of x is still equal to v_{old} or not using the information in the descriptor

object of the update. Since our design avoids the ABA problem, if the field of x is v_{old}, the update has not been committed yet. So, in our approach, each committed update operation behaves like an atomic operation that is performed at the step that commits the update. Thus, the linearization is obtained by ordering operations according to the times they were committed.

6 Doubly-linked List Implementation

We employ our approach to implement a non-blocking linearizable doubly-linked list. In this chapter, we describe our implementation.

Doubly-linked lists have many applications in distributed systems. For instance, when the Java garbage collector detects that the memory allocated to represent an object can be freed, it adds the object to a doubly-linked list that keeps track of all objects to be deallocated. Some scheduling algorithms also use an ordered doubly-linked list of tasks to be scheduled.

A list consists of a sequence of items. A process can access the list via a cursor that locates an item in the list. In a system with multiple processes, there may be multiple cursors for a list. Our doubly-linked list supports several operations to move a cursor along the list and insert or delete an item at a cursor's location. We provide a novel sequential specification for the doubly-linked list to make it easy to use as a black box in different applications.

Multiple cursors may point to the same item, so the item that one cursor is

located at might be removed from the list by a deletion using a different cursor. To ensure the cursor remains located at an item in the list, the cursor advances to the next item in the list. In our implementation, the last item in the list is always a special end-of-list marker (EOL) that no operation can remove from the list. This ensures that the list has at least one item at any time. So, in the case of deletion, the next item always exists. Thus, cursors always point to an item in the list so that one process's deletion cannot cause another process to lose its place in the list.

Suppose a process prc creates a cursor c and sets the cursor c to an item x in the list. Then, another process prc' deletes x from the list, causing c to move to the next item y in the list. Since x is removed by prc', prc does not yet know that c is no longer at x. Then, prc calls a DELETE operation with c to attempt to remove x. This DELETE should not remove y from the list. Instead, in our specification, the deletion of x causes c to become temporarily *invalid*. So, that process's deletion returns invalidCursor as a warning that the cursor has been moved by another operation. More generally, when a cursor becomes invalid, the next operation that is called using it returns invalidCursor to indicate to the owner of the cursor that the cursor has been moved. When an operation returns invalidCursor, the cursor becomes valid again.

Insertions can also cause a cursor to become invalid. For example, suppose a process prc is traversing the list using a cursor c and c is associated with an item

with value 5 in the list. Then, *prc* advances c to the next item in the list, whose value is 8. After that, another process prc' inserts an item with value 7 before the item with value 8. Then, *prc* calls InsertBefore(c, 6) because it wishes to insert an item with value 6 between the items whose values are 5 and 8. If InsertBefore succeeded, it would actually insert the item with value 6 between the items whose values are 7 and 8 and the order of items would not be preserved. In our sequential specification, instead, the InsertBefore operation returns invalidCursor. This invalidation ensures that an item can be inserted between two specific items in our list. So, if an item is inserted before the cursor, the cursor is invalid only if the next operation called with the cursor is InsertBefore. This makes it easy to use a list to store items in sorted order. If an InsertBefore invalidates a cursor and the next operation that is called with the cursor is also InsertBefore, the second operation returns invalidCursor to indicate to the owner of the cursor that the previous item in the list has changed, in case the process wants to preserve an ordered list. When an operation returns invalidCursor, the cursor becomes valid again.

6.1 Sequential Specification

We now provide a formal sequential specification of a list's behaviour. A *list* is a pair (L, S) where L is a finite sequence of distinct items ending with a special

end-of-list marker (EOL), and S is a set of cursors. The state of the list is initially $(\langle \text{EOL} \rangle, \emptyset)$. Each item in the list has a value. These values need not be distinct. For example, if a list NL contains natural numbers, then multiple items in NL may have the value 7, but each occurrence of 7 in NL is treated as a distinct item in the list. Processes can access the sequence L only via cursors. Each cursor is an indicator in a process's local memory that is located at an item in the sequence.

A list supports eight types of operations: InitializeCursor, DestroyCursor, ResetCursor, InsertBefore, Delete, Get, MoveRight and MoveLeft. Each operation is called with a cursor. InitializeCursor adds a new cursor to S, DestroyCursor removes the cursor from S and ResetCursor moves the cursor to the first item in L. InsertBefore adds a new item into L at the cursor's location and Delete removes the item at the cursor's location. Get returns the value of the item that a cursor is located at. MoveRight and MoveLeft move the cursor to the adjacent item in either direction. For a cursor c, if InitializeCursor(c) is called but not DestroyCursor(c), c is *active*. Each operation except InitializeCursor is called with an active cursor.

Each cursor is a tuple $(name, item, invDel, invIns)$ that includes the unique name of the cursor, the item that at which the cursor is located and two boolean values that indicate whether the cursor is invalid for different operations. The $invDel$ and $invIns$ fields of the cursor keep track of whether the cursor is invalid

as a result of a deletion or insertion performed using another cursor. We refer to the fields of a cursor c using the notation $c.name$, $c.item$, etc. We assume cursors belong to individual processes. Thus, a process prc can call an operation with a cursor c only if prc itself created c and c has not been destroyed.

Below, we describe the state transitions and responses for each type of operation on a list in state (L, S). Let $firstItem$ be the first item in L. If c is a cursor in S, let $c.nxtItem$ be the next item after $c.item$ in L (if $c.item$ is not EOL) and $c.prvItem$ be the item preceding $c.item$ in L (if $c.item$ is not $firstItem$). If an operation is called with a cursor c, the operation sets both $c.invDel$ and $c.invIns$ to false before the operation terminates. Tables 6.1–6.3 summarize the effects of each operation on the list.

- InitializeCursor(c) adds a new tuple ($name$, $firstItem$, false, false) to the set S and returns ack. (The $name$ is a new value that is chosen arbitrarily and it is unique for each cursor. It plays the role of a unique identifier.)

- DestroyCursor(c) removes c from S and returns ack.

- ResetCursor(c) sets $c.item$ to $firstItem$ and returns ack.

If Delete(c), Get(c), MoveRight(c) or MoveLeft(c) is called and $c.invDel$ is true, the operation returns invalidCursor. If InsertBefore(c) is called and either $c.invDel$ or $c.invIns$ is true, the operation returns invalidCursor.

Otherwise, the operation induces the following state transition and response.

Operation	Effect on L	Effect on S	Return value
InitializeCursor(c)	-	add a new cursor c $c.invDel \leftarrow$ false $c.invIns \leftarrow$ false $c.item \leftarrow firstItem$	ack
DestroyCursor(c)	-	remove cursor c	ack
ResetCursor(c)	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false $c.item \leftarrow firstItem$	ack
Get(c) where $c.invDel =$ false	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	$c.item$'s value
Get(c) where $c.invDel =$ true	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	invalidCursor
InsertBefore(c, v) where $c.invIns$ and $c.invDel$ are false	add new item with value v just before $c.item$ into L	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false $c'.invIns \leftarrow$ true	true
InsertBefore(c, v) where $c.invIns$ or $c.invDel$ is true	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	invalidCursor

Table 6.1: Effects of InitializeCursor, DestroyCursor, ResetCursor, Get and InsertBefore operations (c' is a cursor such that $c \neq c'$ and $c.item = c'.item$)

Operation	Effect on L	Effect on S	Return value
DELETE(c) where $c.invDel$ = false and $c.item \neq$ EOL	remove $c.item$ from L	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false $c.item \leftarrow c.nxtItem$ $c'.invDel \leftarrow$ true $c'.item \leftarrow c.nxtItem$	true
DELETE(c) where $c.invDel$ = true	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	invalidCursor
DELETE(c) where $c.invDel$ = false and $c.item$ = EOL	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	false
MOVERIGHT(c) where $c.invDel$ = false and $c.item \neq$ EOL	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false $c.item \leftarrow c.nxtItem$	true
MOVERIGHT(c) where $c.invDel$ = true	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	invalidCursor
MOVERIGHT(c) where $c.invDel$ = false and $c.item$ = EOL	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	false

Table 6.2: Effects of DELETE and MOVERIGHT operations (c' is a cursor such that $c \neq c'$ and $c.item = c'.item$)

Operation	Effect on L	Effect on S	Return value
MoveLeft(c) where $c.invDel$ = false and $c.item \neq firstItem$	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false $c.item \leftarrow c.prvItem$	true
MoveLeft(c) where $c.invDel$ = true	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	invalidCursor
MoveLeft(c) where $c.invDel$ = false and $c.item = firstItem$	-	$c.invDel \leftarrow$ false $c.invIns \leftarrow$ false	false

Table 6.3: Effects of MoveLeft operations

- InsertBefore(c, v) adds a new item with value v just before $c.item$ in L and returns true. For all cursors $c' \neq c$ such that $c'.item = c.item$, it sets $c'.invIns$ to true.

- If $c.item \neq$ EOL, Delete(c) removes $c.item$ from L. For all cursors $c' \neq c$ such that $c'.item = c.item$, it sets $c'.item$ to $c.nxtItem$ and $c'.invDel$ to true. It also sets $c.item$ to $c.nxtItem$ and returns true.

If $c.item =$ EOL, Delete(c) returns false.

- Get(c) does not change (L, S) and it returns the value of $c.item$.

- MoveRight(c) does not change L. If $c.item \neq$ EOL, it sets $c.item$ to

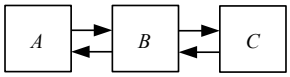


Figure 6.1: The list containing three Nodes

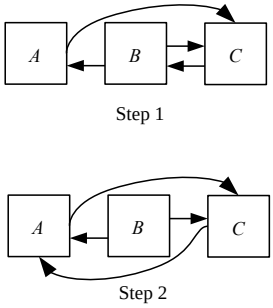


Figure 6.2: Removing the Node B from the list

$c.nxtItem$ and returns true; otherwise, it does not change (L, S) and returns false.

- MoveLeft(c) does not change L. If $c.item \neq firstItem$, it sets $c.item$ to $c.prvItem$ and returns true; otherwise, it does not change (L, S) and returns false.

6.2 Overview of How Updates Are Performed

List items are represented by Node objects, which have pointers nxt and prv to adjacent Nodes. A Cursor is represented in a process's local memory by a single pointer to a Node.

Consider a list with three consecutive Nodes, A, B and C. (See Figure 6.1.) To

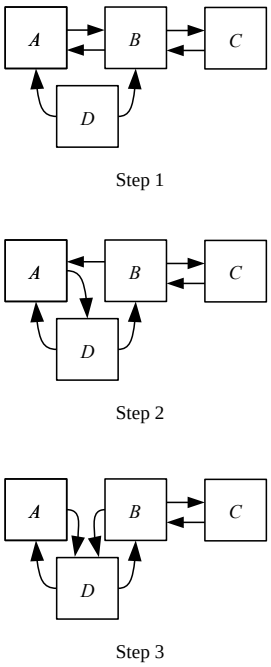


Figure 6.3: Inserting the new Node D between A and B (standard sequential approach)

remove B from the list, $A.nxt$ is set to C and $C.prv$ is set to A. (See Figure 6.2.) So, to apply our methodology, we must flag three Nodes A, B and C before changing the pointers.

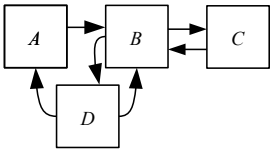


Figure 6.4: The Insertion swings $B.prv$ from A to D incorrectly.

To insert a new Node D between A and B, the standard sequential implemen-

tation of a doubly-linked list simply sets $A.nxt$ and $B.prv$ to D. (See Figure 6.3.) However, doing this in our concurrent setting would cause an ABA problem. Suppose a process prc attempting to insert D between A and B successfully sets $A.nxt$ to D (step 2 of Figure 6.3) and then goes to sleep just when it is about to swing the pointer $B.prv$ from A to D. After that, another process prc' attempts to remove D from the list. So, prc' first helps prc to complete its insertion by setting $B.prv$ to D (step 3 of Figure 6.3) and then removes D from the list. At this point, the list looks like Figure 6.1 again. If prc then wakes up, it attempts to swing $B.prv$ from A to D and succeeds, since $B.prv$ is again equal to A. (See Figure 6.4.) This is incorrect, since D has been removed from the list.

To avoid the ABA problem, we could replace A and B by new copies, but it turns out that replacing one of them is sufficient. When a Node is inserted between A and B, we replace the Node B with a new copy. Thus, if the list consists of three Nodes, A, B and C (Figure 6.1), to insert a new Node D between A and B, first the new Node D and a new copy of B, denoted B', are created. Then, $A.nxt$ is set to D and $C.prv$ is set to B'. (See Figure 6.5.) By doing the insertion in this way, we ensure that the node that the two modified pointers used to point to is removed from the data structure, as required by our methodology. To apply our methodology, we must flag three Nodes A, B and C before changing these pointers.

Suppose a Cursor is located at the Node B and then B is removed from the list.

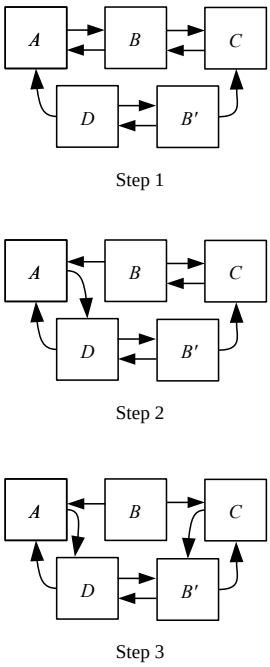


Figure 6.5: Inserting the new Node D between A and B (our approach, which creates a new copy B' of B)

If B is removed by a DELETE operation as in Figure 6.2, the Cursor must advance to the next Node in the list, which is C. If B is removed by an INSERTBEFORE operation as in Figure 6.5, B is replaced with a new copy B'. Then, the Cursor must move to the new copy B'. To distinguish these two cases, each Node also has a *state* field. Between flagging Nodes and changing the pointers to remove B, the operation sets the *state* field of B to marked if B has been removed by a deletion or copied if B has been replaced by a new copy as a result of an insertion. To find the new copy of a Node efficiently, each Node also has a *copy* field that is set to the

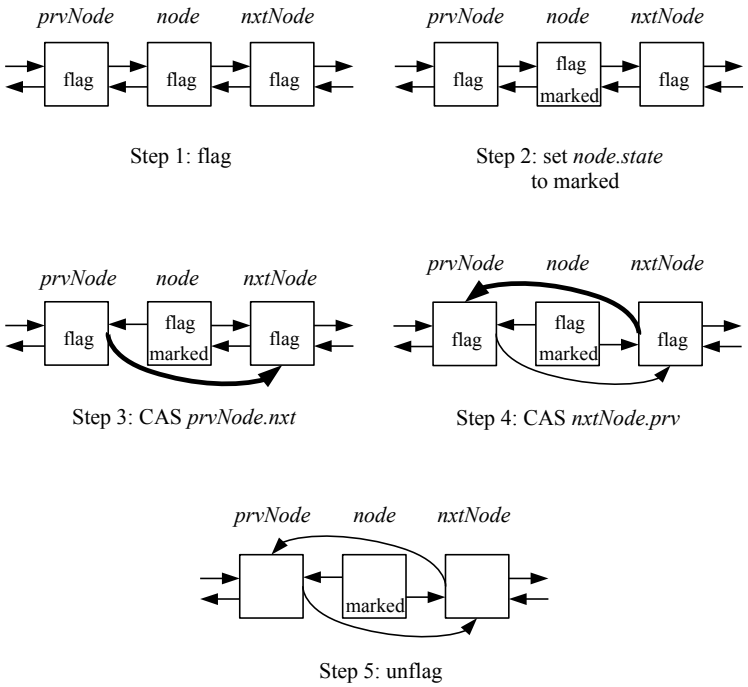


Figure 6.6: Steps of DELETE operation

new copy before replacing the Node. So, the Cursor that is located at the removed Node B can decide to move to the next Node in the list or to the new copy of the Node based on the *state* field of the Node B.

Applying our methodology to the doubly-linked list, update operations are done in several steps as shown in Figure 6.6 and 6.7. Between the successful forward and the successful backward CAS step of an update (step 3 and 4 in Figure 6.6 and step 4 and 5 in Figure 6.7), the pointers of the list are temporarily inconsistent. The update operation will be linearized at its successful forward CAS step. We say a Node v is *reachable at configuration* C_i if there is a path of the *nxt* pointers from

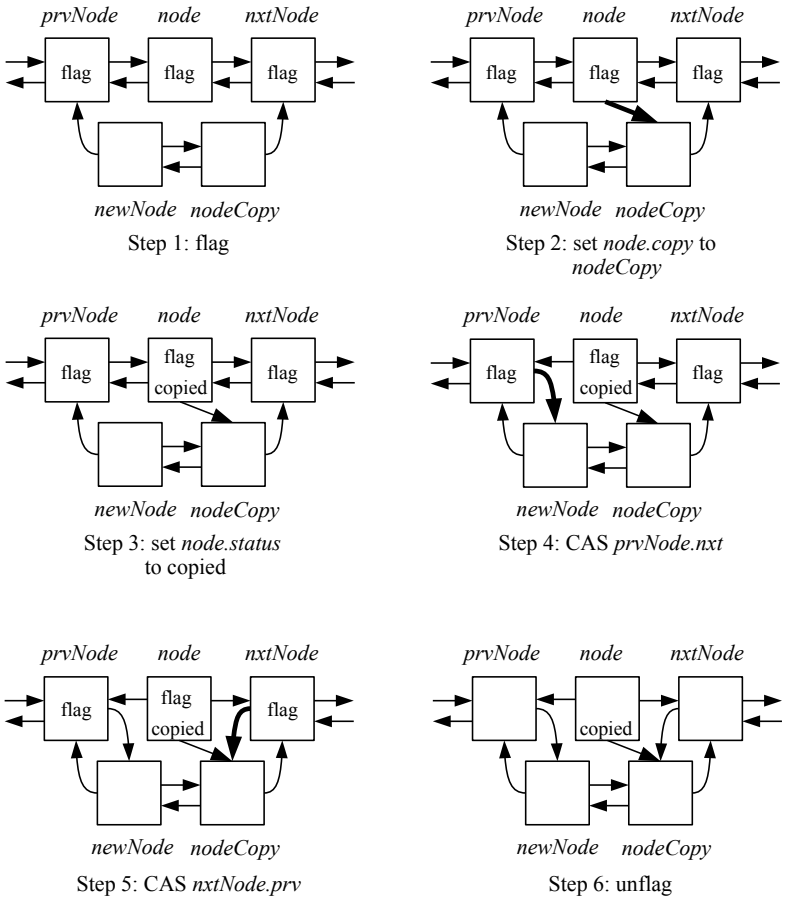


Figure 6.7: Steps of InsertBefore operation

$Head$ to v in C_i. In particular, the $Head$ Node is always *reachable*. The goal is to ensure reachable Nodes always represent the state of the list (i.e., all the items in their proper order).

6.3 Representation of the List in Memory

Figure 6.8 presents the object types used to implement our doubly-linked list.

```
1.  type Cursor
2.      Node node                                   ▷ location of Cursor

3.  type Node                                       ▷ represent list item
4.      Value value
5.      Node nxt                                    ▷ next Node
6.      Node prv                                    ▷ previous Node
7.      Node copy                                   ▷ new copy of Node (if replaced)
8.      Info info                                   ▷ descriptor of update
9.      {copied, marked, ordinary} state
                    ▷ indicates if Node is replaced by a copy or marked for deletion

10. type Info                                       ▷ descriptor of an update operation
11.     Node[3] nodes                               ▷ Nodes to be flagged
12.     Info[3] oldInfo                             ▷ expected values of CASs that flag
13.     Node newNxt                                 ▷ set nodes[0].nxt to this
14.     Node newPrv                                 ▷ set nodes[2].prv to this
15.     Boolean rmv                                 ▷ is I.nodes[1] being deleted?
16.     {inProgress, committed, aborted} status
```

Figure 6.8: Object types used to implement doubly-linked lists

The elements of a list are represented by Node objects and cursors that are located at a list are represented by Cursor objects. Info objects are used as our update operations' descriptors to facilitate helping.

A Node has the following fields. The *value* field contains the item's value, *nxt* and *prv* point to the next and previous Nodes in the list (if these exist), *copy* points to a new copy of the Node (if any), *info* points to an Info object that is the descriptor of the update that last flagged the Node, and *state* is initially ordinary and is set to copied (before the Node is replaced by a new copy) or marked (before the Node is deleted). The *info* field of each Node is initially set to a dummy Info

▷ Initialization

17. $dum \leftarrow$ new Info(null, null, null, null, false, aborted) ▷ dummy Info object
18. $Head \leftarrow$ new Node(null, null, null, null, dum, ordinary) ▷ sentinel Node
19. $EOLnode \leftarrow$ new Node(EOL, null, $Head$, null, dum, ordinary) ▷ end-of-list marker
20. $Tail \leftarrow$ new Node(null, null, $EOLnode$, null, dum, ordinary) ▷ sentinel Node
21. $Head.nxt \leftarrow EOLnode$
22. $EOLnode.nxt \leftarrow Tail$

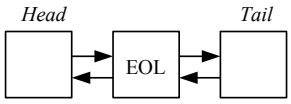


Figure 6.9: Initialization of the doubly-linked list

object, *dum*.

To avoid special cases, we add sentinel Nodes $Head$ and $Tail$, which do not represent list items, at the ends of the list. This ensures that each Node that represents an item in the list has non-null *nxt* and *prv* pointers. The $Head$ and $Tail$ Nodes are never changed and Cursors never move to $Head$ or $Tail$. The last Node before $Tail$ always contains the value EOL. Figure 6.9 shows the initialization of the list.

The *info*, *nxt* and *prv* fields of a Node are changed using CAS steps. A CAS step on the *info* field of a Node is called a *flag CAS*. A CAS step on the *nxt* field of a Node is called a *forward CAS*. A CAS step on the *prv* field of a Node is called a *backward CAS*.

An Info object I has the following fields. $I.nodes[0..2]$ stores the three Nodes $prvNode$, $node$ and $nxtNode$ to be flagged before changing the list. $I.oldInfo[0..2]$

stores the expected values to be used by the flag CAS steps on $prvNode$, $node$ and $nxtNode$. $I.newNxt$ and $I.newPrv$ store the new values for the forward and backward CAS steps on $prvNode.nxt$ and $nxtNode.prv$. $I.rmv$ indicates whether $node$ should be deleted from the list or replaced by a new copy. $I.status$, indicates whether the update is inProgress (the initial value), committed (because the update has been completed) or aborted (because a Node was not flagged successfully). One exception is the dummy Info object *dum*, whose *status* is initially aborted. The *status* field of I is the only field of I whose value might be changed after I is created.

A Node is *flagged for* I if its *info* field is I and $I.status$ = inProgress. We say a Node is flagged if it is flagged for some Info object I. Thus, setting $I.status$ to committed or aborted also has the effect of removing I's flags.

6.4 Descriptions of Algorithms

Pseudo-code for our doubly-linked list implementation is given on page 67 to 69.

Both the InsertBefore(c, v) and Delete(c) operations have the same structure. They first call UpdateCursor(c) to bring the Cursor c up to date (line 25 or 39), and return invalidCursor if this routine indicates c has been invalidated (line 26 or 40). Then, they call CheckInfo to see if there is interference by other updates (line 29 or 43). If not, they create an Info object I (line 33 or 45) and call

```
23. InsertBefore(c: Cursor, val: Value): {true, invalidCursor}
24.   while (true)
25.     ⟨node, nodeInfo, nxtNode, prvNode, invDel, invIns⟩ ← UpdateCursor(c)
                                                          ▷ recover c's location
26.     if invDel or invIns then return invalidCursor      ▷ c is invalid for insertion
27.     nodes ← [prvNode, node, nxtNode]
28.     oldInfo ← [prvNode.info, nodeInfo, nxtNode.info]
29.     if CheckInfo(nodes, oldInfo) then      ▷ if no interference with other updates
30.       newNode ← new Node(val, null, prvNode, null, dum, ordinary)
31.       nodeCopy ← new Node(node.value, nxtNode, newNode, null, dum, ordinary)
32.       newNode.nxt ← nodeCopy
33.       I ← new Info(nodes, oldInfo, newNode, nodeCopy, false, inProgress)
                                              ▷ create the insertion's descriptor
34.       if Help(I) then                        ▷ if the insertion is completed
35.         c.node ← nodeCopy                      ▷ move c to the new copy
36.         return true                          ▷ the insertion is completed

37. Delete(c: Cursor): {true, false, invalidCursor}
38.   while (true)
39.     ⟨node, nodeInfo, nxtNode, prvNode, invDel, -⟩ ← UpdateCursor(c)
                                                          ▷ recover c's location
40.     if invDel then return invalidCursor                          ▷ c is invalid
41.     nodes ← [prvNode, node, nxtNode]
42.     oldInfo ← [prvNode.info, nodeInfo, nxtNode.info]
43.     if CheckInfo(nodes, oldInfo) then      ▷ if no interference with other updates
44.       if node.value = EOL then return false      ▷ node is the last item in the list
45.       I ← new Info(nodes, oldInfo, nxtNode, prvNode, true, inProgress)
                                              ▷ create the deletion's descriptor
46.       if Help(I) then                        ▷ if the deletion is completed
47.         c.node ← nxtNode                       ▷ move c to the next item
48.         return true                          ▷ the deletion is completed

49. MoveLeft(c: Cursor): {true, false, invalidCursor}
50.   ⟨node, -, -, prvNode, invDel, -⟩ ← UpdateCursor(c)          ▷ recover c's location
51.   if invDel then return invalidCursor                          ▷ c is invalid
52.   if prvNode = Head then return false                ▷ node is the first item in the list
53.   if prvNode.prv.nxt ≠ prvNode and prvNode.nxt = node then ▷ prvNode not in the list
54.     if prvNode.state = copied then c.node ← prvNode.copy  ▷ move c to the new copy
55.     else                                        ▷ prvNode is deleted from the list
56.       prvPrvNode ← prvNode.prv
57.       if prvPrvNode = Head then return false
                                          ▷ prvNode was the first item in the list
58.       c.node ← prvPrvNode                ▷ move c to the item before prvNode
59.   else c.node ← prvNode                      ▷ move c to the item before node
60.   return true
```

61. MoveRight(c: Cursor): {true, false, invalidCursor}
62. $\langle node$, -, $nxtNode$, -, $invDel$, -$\rangle \leftarrow$ UpdateCursor(c) ▷ recover c's location
63. if $invDel$ then return invalidCursor ▷ c is invalid
64. if $node.value$ = EOL then return false ▷ $node$ is the last item in the list
65. $c.node \leftarrow nxtNode$ ▷ move c to the item after $node$
66. return true

67. InitializeCursor(c: Cursor): ack
68. $c.node \leftarrow Head.nxt$ ▷ initialize c to the first item in the list
69. return ack

70. DestroyCursor(c: Cursor): ack
71. return ack

72. ResetCursor(c: Cursor): ack
73. $c.node \leftarrow Head.nxt$ ▷ move c to the first item in the list
74. return ack

75. Get(c: Cursor): Value
76. $\langle node$, -, -, -, $invDel$, -$\rangle \leftarrow$ UpdateCursor(c) ▷ recover c's location
77. if $invDel$ then return invalidCursor ▷ c is invalid
78. return $node.value$

79. UpdateCursor(c: Cursor): ⟨Node, Info, Node, Node, Boolean, Boolean⟩
    Post-conditions:
    if UpdateCursor returns $\langle$ $node$, $info$, $prvNode$, $nxtNode$, -, -$\rangle$, then at some earlier configurations during UpdateCursor,
    - $node$ was reachable,
    - $node.info$ was $info$,
    - $prvNode.nxt$ was $node$, and
    - $node.nxt$ was $nxtNode$, and
    - if x is the value of $c.node$ at the beginning of UpdateCursor, between x and $node$, there is a chain of $nxt/copy$ pointers (depending on the value of the $state$ of Nodes between x and $node$).
80. $invDel \leftarrow$ false
81. $invIns \leftarrow$ false
82. while($c.node.prv.nxt \neq c.node$) ▷ $c.node$ is not in the list
83. if $c.node.state$ = copied then ▷ $c.node$ is replaced
84. $invIns \leftarrow$ true ▷ make c invalid for insertion
85. $c.node \leftarrow c.node.copy$ ▷ move c to the new copy
86. if $c.node.state$ = marked then ▷ $c.node$ is deleted
87. $invDel \leftarrow$ true ▷ make c invalid
88. $c.node \leftarrow c.node.nxt$ ▷ move c to the next item
89. $info \leftarrow c.node.info$ ▷ the $info$ field is read before reading the prv and nxt field
90. return $\langle c.node, info, c.node.nxt, c.node.prv, invDel, invIns\rangle$

```
91.  CheckInfo(nodes: Node[3], oldInfo: Info[3]): Boolean
     Pre-condition: for each 0 ≤ i ≤ 2, nodes[i].info was earlier equal to oldInfo[i].
92.    for i ← 0 to 2,                                  ▷ detect other updates in progress
93.        if oldInfo[i].status = inProgress then
94.            Help(oldInfo[i])                          ▷ help other update
95.            return false                              ▷ retry my update
96.    for i ← 0 to 2,                                  ▷ detect removed Nodes
97.        if nodes[i].state ≠ ordinary then return false ▷ retry my update
98.    for i ← 1 to 2,                                  ▷ check if flagging will fail
99.        if nodes[i].info ≠ oldInfo[i] then
100.           return false                              ▷ retry my update
101.   return true                                       ▷ no interference detected

102. Help(I: Info): Boolean
     Post-conditions:
       if Help returns true at configuration C5, then there is a sequence of configurations
           C1 < C2 < C3 < C3 < C4 < C5 such that
           • at C1, all Nodes in I.nodes are successfully flagged by I,
           • at C2, if I.rmv is false, I.nodes[1].copy is set to I.newPrv,
           • at C3, I.nodes[1].state is set to marked or copied,
           • at C4, the first forward and first backward CAS of I succeed.
103.   doPtrCAS ← true
104.   i ← 0
105.   while (i < 3 and doPtrCAS)
106.       CAS(I.nodes[i].info, I.oldInfo[i], I)                    ▷ flag CAS
107.       doPtrCAS ← (I.nodes[i].info = I)
108.       i ← i + 1
109.   if doPtrCAS then                  ▷ all three Nodes are successfully flagged for I
110.       if I.rmv then I.nodes[1].state ← marked                  ▷ in case of deletion
111.       else                                                     ▷ in case of insertion
112.           I.nodes[1].copy ← I.newPrv          ▷ set the copy field to the new copy
113.           I.nodes[1].state ← copied
114.       CAS(I.nodes[0].nxt, I.nodes[1], I.newNxt)                ▷ forward CAS
115.       CAS(I.nodes[2].prv, I.nodes[1], I.newPrv)                ▷ backward CAS
116.       I.status ← committed                                     ▷ the update is completed
117.   else if I.status = inProgress then
118.       I.status ← aborted                                       ▷ the update failed
119.   return (I.status = committed)
```

HELP(I) to complete the update (line 34 or 46). If unsuccessful, they retry. Next, we describe each of the three routines used by updates in more detail.

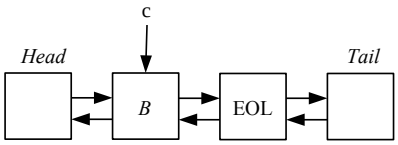
Since a Cursor c is a pointer in a process's local memory, it becomes out of date if the Node to which it points is deleted or replaced by an update using another Cursor. Thus, at the beginning of every update, move or GET operation called with c, UPDATECURSOR(c) is called to bring $c.node$ up to date and determine whether it is invalid. If $c.node$ has been replaced with a new copy by an INSERTBEFORE, UPDATECURSOR sets $invIns$ to true (line 84) and follows the copy pointer (line 85). Similarly, if $c.node$ has been deleted, UPDATECURSOR sets $invDel$ to true (line 87) and follows the nxt pointer (line 88), which is the next Node at the time of deletion. UPDATECURSOR repeats the loop at line 82–88 until the test on line 82 indicates that $c.node$ is in the list. (We show how this test works later.)

The following example shows how UPDATECURSOR works. Suppose a Cursor c created by a process prc is located at Node B in the list shown in Figure 6.10a. Then, another process prc' adds a new Node A before Node B as shown in Figure 6.10b. As a result of the insertion, Node B is also replaced with a new copy B'. After the replacement, c should be located at B', but since it is in prc's local memory, prc' cannot update it. After that, prc' deletes Node B' as shown in Figure 6.10c. When this deletion occurs, c should be advanced from B' to the next Node EOL, according to our sequential specification. Next, prc calls an operation

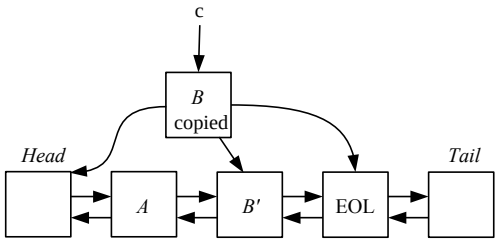
with Cursor c. That operation begins by calling UpdateCursor(c) to bring c up to date. At the beginning of UpdateCursor, $c.node$ is still Node B. Since B is no longer in the list, the test on line 82 evaluates to true. Because $B.state$ is set to copied and $B.copy$ is B', the first iteration of the loop in UpdateCursor sets $invIns$ to true on line 84 and $c.node$ to B' on line 85. Since B' is not in the list either, the test on line 82 evaluates to true again. Since $B.state$ is set to marked, the second iteration of the loop sets $invDel$ to true on line 87 and $c.node$ to the Node EOL (read from $B'.nxt$) on line 88. Because EOL is in the list, the test on line 82 fails and UpdateCursor exits the loop. Thus, c has been updated to the correct Node, EOL, and it has detected that both an insertion and a deletion have invalidated the Cursor since it was last used.

After calling UpdateCursor, each update op calls CheckInfo to see if some Node that op wants to flag is already flagged with an Info object I' of another update (line 93). If so, it calls Help(I') (line 94) to try completing the other update, and returns false to indicate op should retry. Similarly, if CheckInfo sees that one of the Nodes is already removed from the list (line 97), it returns false, causing op to retry. If CheckInfo sees that the $info$ fields of $node$ or $nxtNode$ has already been changed by another process (line 99), to avoid flagging $prvNode$, it returns false, causing op to retry. (This test is crucial for our amortized analysis.)

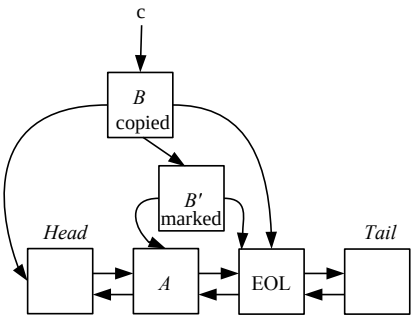
The Help(I) routine performs the real work of an update as in Figures 6.6



(a) The list containing two Nodes B and EOL



(b) Node A is inserted before Node B, so Node B is replaced by a new copy B'



(c) Node B' is removed from the list

Figure 6.10: An example of a call to UpdateCursor

and 6.7. First, it uses flag CAS steps to store I in the *info* fields of the Nodes to be flagged (line 106). If HELP(I) sees a Node v is not flagged successfully (line 107), HELP(I) checks if $I.status$ is inProgress (line 117). If so, we will prove that no helper of I succeeded in flagging all three nodes. So, v was flagged by another update before HELP(I)'s flag CAS. Thus, $I.status$ is set to aborted (line 118) and HELP(I) returns false (line 119), causing *op* to retry.

If the Nodes $prvNode$, $node$ and $nxtNode$ in $I.nodes$ are all flagged successfully with I, $node.state$ is set to marked (line 110) for a deletion, or copied (line 113) for an insertion. In the latter case, $node.copy$ is first set to the new copy (line 112). Then, a forward CAS (line 114) changes $prvNode.nxt$ and a backward CAS (line 115) changes $nxtNode.prv$. Finally, HELP(I) sets $I.status$ to committed (line 116) and returns true (line 119).

Next, we describe the move operations. Since the nxt pointers always reflect the "true" state of the list, MOVERIGHT is fairly simple. A MOVERIGHT(c) calls UPDATECURSOR(c) (line 62), which sets $c.node$ to a Node $node$ and also returns a Node $nxtNode$ read from $node.nxt$. MOVERIGHT returns invalidCursor if this routine indicates c has been invalidated (line 63). Otherwise, we show there is a time during the MOVERIGHT when $node$ is reachable and $node.nxt = nxtNode$. If $node.value$ = EOL, the operation cannot move c and returns false (line 64). Otherwise, it sets $c.node$ to $nxtNode$ (line 65) and returns true.

A MoveLeft(c) is more complex than MoveRight because prv pointers are updated *after* an update's linearization point, so they are sometimes inconsistent with the true state of the list. A MoveLeft first calls UpdateCursor(c) (line 50), which updates $c.node$ to some Node $node$ and also returns a Node $prvNode$ read from $node.prv$. MoveLeft returns invalidCursor if this routine indicates c has been invalidated (line 51). If $prvNode$ is $Head$, the operation cannot move c to $Head$ and returns false (line 52). If the test on line 53 indicates $prvNode$ was reachable, $c.node$ is set to $prvNode$ (line 59). This is also done if $prvNode.nxt \neq node$; in this case, we can show that $node$ became unreachable during the MoveLeft operation, but $prvNode.nxt$ pointed to $node$ just before it became unreachable. Otherwise, $prvNode$ has become unreachable and the test $prvNode.nxt = node$ on line 53 ensures that $prvNode$ was the element before $node$ when it became unreachable. If $prvNode$ was replaced by an insertion, $c.node$ is set to that replacement node (line 54). If $prvNode$ was removed by a deletion, we set $c.node$ to $prvNode.prv$ (line 58), unless that node is $Head$. We prove later that whenever MoveLeft updates $c.node$ to some value v, there is a time during the operation when v is reachable and $v.nxt = node$.

7 Correctness Proof of Doubly-linked List

In this chapter, we present a detailed correctness proof of our doubly-linked list. Chapter 7 includes four sections. Section 7.1 shows some basic invariants for our list implementation. This includes showing that a Cursor is never located at $Head$ or $Tail$ and any field that is read in the pseudo-code is non-null. After that, we associate a real number, called an abstract value and denoted $absVal$, with each Node and we show that, for Nodes x and y, if $x.nxt = y$ or $y.prv = x$, then $x.absVal < y.absVal$. This ensures, for example that the pointer structure does not contain loops.

For an Info object I, a *CAS of* I refers to a CAS step executed inside Help(I). Section 7.2 shows that flagging works as expected: We show that there is no ABA problem on the *info* field of a Node object and only the first flag CAS of an Info object I on a Node can succeed. Then, we prove the *info* field of a Node is not changed from I to another value while $I.status$ is inProgress. We also show that, for each of the lines 110–116, when the first execution of that line among all calls

to help(I) occurs, all Nodes in $I.nodes$ are flagged for I.

Section 7.3 shows that the list is changed by updates as expected: Suppose I is an Info object. We show that there is no ABA problem on the nxt and prv field of a Node. We prove that the first forward CAS of I and the first backward CAS of I succeed and no other forward or backward CAS of I succeeds. We show that, at the configuration before the first forward CAS of I, $I.nodes[0]$, $I.nodes[1]$ and $I.nodes[2]$ are three consecutive Nodes in the list. Besides, for each Node $x \neq Head$, we show that if the test ($x.prv.nxt \neq x$) evaluates to false, x is reachable at the configuration before the test; otherwise x is unreachable at the configuration after the test.

Section 7.4 shows that our list implementation is linearizable: We define a linearization point for each operation that terminates. Then, we define an auxiliary variable $(\mathfrak{L}, \mathfrak{S})$ of type list. When an operation is linearized, the same operation is atomically applied to $(\mathfrak{L}, \mathfrak{S})$ according to the sequential specification. To prove our linearization is correct, we show how the auxiliary variable $(\mathfrak{L}, \mathfrak{S})$ is accurately reflected in the state of the actual list, implying each operation returns the same response as the corresponding operation on $(\mathfrak{L}, \mathfrak{S})$. We use abstract values to construct a one-to-one correspondence between Nodes in the list and items in $\mathfrak{L}$.

Throughout this chapter, we consider an execution C_0, C_1, C_2, ... of our implementation of the doubly-linked list, where C_0 is the initial configuration, *after*

the initialization code of Figure 6.9 is executed.

7.1 Basic Invariants

First, we have the following observations from the pseudo-code.

Observation 7.1. *The value field of a Node object is never changed.*

No field of an Info object is changed except the status field.

When a new Info object is created at line 33 or 45, its *status* field is initially inProgress. Since the *status* field of an Info object is only set to committed or aborted (at line 116 or 118), we have the following observation.

Observation 7.2. *The status field of an Info object is never set to inProgress after initialization.*

The *state* field of a Node is set only at line 110 or 113, which set the value to marked or copied, respectively.

Observation 7.3. *Let x be a Node object. Then, $x.state$ is initially ordinary and $x.state$ is only set to copied or marked.*

A new Node is created at line 18, 19, 20, 30 or 31. Let *dum* be the Info object that is created at line 17. When a Node is created, its *info* field is initially *dum*. Since *dum.status* is initially aborted, we have the following corollary of Observation 7.2.

Corollary 7.4. *The field dum.status is never inProgress.*

HELP(I) is called to perform the update described by I. Since the Info object *dum* does not describe any update operation, we require the following lemma.

Lemma 7.5. *If* HELP*(I) is called, I is not the Info object dum that is created at line 17.*

Proof. The HELP routine is called at line 34, 46 or 94. Before HELP(I) is called at line 34 or 46, I is created at line 33 or 45. Before HELP(I) is called at line 94, $I.status$ is inProgress at line 93. Since $dum.status$ is never inProgress by Corollary 7.4, $I \neq dum$. □

The call to UPDATECURSOR on line 25 or 39 that precedes the creation of an Info object I on line 33 or 45 is called *the* UPDATECURSOR *that belongs to* I. The call to CHECKINFO on line 29 or 43 that precedes the creation of an Info object I on line 33 or 45 is called *the* CHECKINFO *that belongs to* I.

A forward or backward CAS of an Info object I is also called a *pointer CAS of* I. *An execution of line* x *belonging to* I refers to an execution of line x inside any invocation of HELP(I). The values of $Head$ and $Tail$ are initialized on line 18 and 20 and never changed thereafter. The following invariant shows when a pointer can or cannot be $Tail$ or $Head$. Since $Head$ and $Tail$ are sentinel Nodes, we wish to show that no Cursor is ever located at $Head$ or $Tail$.

Invariant 7.6. *For each Node x, Cursor c and Info object I, the following statements are true.*

1. *If $x.nxt = Tail$, then $x.value = EOL$.*
2. *$I.newNxt$ is neither $Head$ nor $Tail$.*
3. *$I.newPrv \neq Tail$.*
4. *If $x.value = EOL$, then $x.state \neq marked$.*
5. *If x is $Head$ or $Tail$, then $x.state = ordinary$.*
6. *If I was created at line 45, then $I.nodes[1].value \neq EOL$.*
7. *$x.nxt \neq Head$.*
8. *$x.prv \neq Tail$.*
9. *$x.copy$ is neither $Head$ nor $Tail$.*
10. *$c.node$ is neither $Head$ nor $Tail$.*
11. *If $\langle y,$ -, $z, x,$ -, -$\rangle$ is the result of some call to* UpdateCursor*, then $z \neq Head$, $x \neq Tail$, and if $z = Tail$, then $y.value = EOL$.*

Proof. It is easy to check that the invariant is true at C_0. We assume the invariant is true up to C_{i-1}, then we prove the step s from C_{i-1} to C_i preserves the invariant.

Statement 1 We need only consider steps s that set the nxt field of a Node x, since a Node's $value$ never changes (by Observation 7.1). If s is a successful forward CAS of some Info object I, then s sets $x.nxt$ to $I.newNxt$, which is not $Tail$ by induction hypothesis 2. If s sets $x.nxt$ to v at line 32, then v is created at line 31, so $v \neq Tail$. Suppose s sets $x.nxt$ to z at line 31. Then, the preceding call to UpdateCursor in line 25 returned $\langle y, \text{-}, z, \text{-}, \text{-}, \text{-}\rangle$. By induction hypothesis 11, if $z = Tail$, $x.value =$ EOL.

Statement 2 Let I be an Info object. $I.newNxt$ is only set if step s is an execution of line 33 or 45. If I is created at line 33, $I.newNxt$ is created at line 30, so it is neither $Head$ nor $Tail$. Suppose I is created at line 45. Then, the call to UpdateCursor that belongs to I returned $\langle node, \text{-}, I.newNxt, \text{-}, \text{-}, \text{-}\rangle$ for some $node$. By induction hypothesis 11, $I.newNxt \neq Head$. Since the condition at line 44 preceding the creation of I is not true, $node.value \neq$ EOL. By induction hypothesis 11, $I.newNxt \neq Tail$.

Statement 3 Let I be an Info object. $I.newPrv$ is only set if step s is an execution of line 33 or 45. If I is created at line 33, $I.newPrv$ is created at line 31, so it is not $Tail$. Suppose I is created at line 45. Then, the call to UpdateCursor that belongs to I returned $\langle \text{-}, \text{-}, \text{-}, I.newPrv, \text{-}, \text{-}\rangle$. By induction hypothesis 11, $I.newPrv \neq Tail$.

Statement 4 We need only consider steps s that set the *state* field of a Node to marked, since a Node's *value* never changes (by Observation 7.1). Suppose s sets $x.state$ to marked at line 110 inside HELP(I) for some Info object I. We show $x.value \neq$ EOL. Since s sets $x.state$ at line 110, $I.rmv$ is true at line 110. So, I is created at line 45. Since $x = I.nodes[1]$, by induction hypothesis 6, $x.value \neq$ EOL.

Statement 5 We need only consider steps s that set the *state* field of a Node to marked or copied. We show s does not set the *state* of $Head$ or $Tail$. Suppose s sets $x.state$ to marked or copied at line 110 or 113 inside HELP(I) for some Info object I. Then, $I.nodes[1] = x$. Let $\langle node$, -, -, -, -, -$\rangle$ be the result of the call to UPDATECURSOR that belongs to I. Since the *node* field of a Cursor is set to *node* when line 85 or 88 is executed for the last time inside that call to UPDATECURSOR, $node = I.nodes[1]$ is neither $Head$ nor $Tail$ by induction hypothesis 10.

Statement 6 We need only consider a step s that creates an Info object I at line 45. Then, the call to UPDATECURSOR that belongs to I returned $\langle I.nodes[1]$, -, -, -, -, -$\rangle$. Then, $I.nodes[1].value \neq$ EOL at line 44 preceding the creation of I. Since a Node's *value* never changes (by Observation 7.1), $I.nodes[1].value$ is never EOL.

Statement 7 We need only consider steps s that set the nxt field of a Node x. If s is a successful forward CAS of some Info object I, then s sets $x.nxt$ to $I.newNxt$, which is not $Head$ by induction hypothesis 2. Line 30 sets the nxt field of a Node to null. If s sets $x.nxt$ to z at line 32, then z is created at line 31, so $z \neq Head$. Suppose s sets $x.nxt$ to z at line 31. Then, the preceding call to UpdateCursor in line 25 returned $\langle$-, -, z, -, -, -$\rangle$. By induction hypothesis 11, $z \neq Head$.

Statement 8 We need only consider steps s that set the prv field of a Node x. If s is a successful backward CAS of some Info object I, then s sets $x.prv$ to $I.newPrv$, which is not $Tail$ by induction hypothesis 3. If s sets $x.prv$ to y at line 31, then y is created at line 30, so $y \neq Tail$. Suppose s sets $x.prv$ to y at line 30. Then, the preceding call to UpdateCursor at line 25 returned $\langle$ -, -, -, y, -, -$\rangle$. By induction hypothesis 11, $y \neq Tail$.

Statement 9 The *copy* field of a Node x is only set at line 112. Suppose s sets $x.copy$ to y at line 112 inside Help(I) for some Info object I. Then, $I.rmv$ is false at line 110. So, I is created at line 33 and y is created at line 31 preceding the creation of I. Thus, y is neither $Head$ nor $Tail$.

Statement 10 We need only consider steps s that modify the *node* field of a Cursor c. If s sets $c.node$ to x at line 35, x is created at line 31, so x is neither

$Head$ nor $Tail$. Suppose s sets $c.node$ to x at line 47 or 65. Then, the preceding call to UpdateCursor in line 39 or 62 returned $\langle node, \text{-}, x, \text{-}, \text{-}, \text{-}\rangle$. By induction hypothesis 11, $x \neq Head$. Since the operation does not return false at line 44 or 64, $node.value \neq$ EOL. So, $x \neq Tail$ by induction hypothesis 11.

If s sets $c.node$ to x at line 54 or 85, x is read from the *copy* field of a Node. So, x is neither $Head$ nor $Tail$ by induction hypothesis 9.

If s sets $c.node$ to x at line 58 or 59, x is read from the *prv* field of a Node earlier at line 56 or at line 90 during the call to UpdateCursor at line 50. So, x is not $Tail$ by induction hypothesis 8. Since the operation does not return false at line 52 or 57, $x \neq Head$.

If s sets $c.node$ to x at line 68 or 73, $Head.nxt = x$ before C_i. By induction hypothesis 1, since $Head.value \neq$ EOL, $x \neq Tail$. By induction hypothesis 7, $x \neq Head$.

Suppose s sets $c.node$ to x on line 88. Then, x is read from the *nxt* field of a Node y. By induction hypothesis 7, $x \neq Head$. Since $y.state$ is marked at line 86, by induction hypothesis 4, $y.value \neq$ EOL. So, $x \neq Tail$ by induction hypothesis 1.

Statement 11 Suppose s is the step in which some call to UpdateCursor returns $\langle y, \text{-}, z, x, \text{-}, \text{-}\rangle$. Since z is read from the *nxt* field of y on line 90 inside that call to UpdateCursor, $z \neq Head$ (by induction hypothesis 7) and if $z = Tail$,

then $y.value$ = EOL (by induction hypothesis 1). Since x is read from the prv field of y on line 90 inside that call to UpdateCursor, $x \neq Tail$ (by induction hypothesis 8). □

The following invariant ensures that whenever the code accesses a field of an object, that object is not null.

Invariant 7.7. *The following are not null:*

1. *the nxt field of each Node (except Tail and any Node to which the newNode pointer of some process points between line 30 and 32),*
2. *the prv field of each Node (except Head),*
3. *the info field of each Node,*
4. *every field of each Info object (except the Info object created at line 17),*
5. *the copy field of each Node whose status is copied,*
6. *the node field of each Cursor, and*
7. *any local variable of type Node or Info that has been assigned a value.*

Proof. The invariant is true at C_0. We assume the invariant is true up to C_{i-1}, and we prove the step s from C_{i-1} to C_i preserves the invariant.

When a Node is created at line 30, it is written only into the local variable *newNode* and cannot be written anywhere else before line 32. So, no process can read its *nxt* field while it contains null.

Statement 1 and 2 We consider steps s that set the *nxt* or *prv* field of a Node to x. If s sets the *prv* field of a Node to x at line 30 or the *nxt* field of a Node to x at line 31, x is an initialized local variable, which is non-null by induction hypothesis 7. If s sets the *prv* field of a Node to x at line 31, x is created earlier at line 30. If s sets the *nxt* field of a Node to x at line 32, x is created earlier at line 31.

If s is a forward or backward CAS of an Info object I, s sets the *nxt* or *prv* field of a Node to $I.newNxt$ or $I.newPrv$, which are not null by induction hypothesis 4 and Lemma 7.5.

Statement 3 We consider steps s that set the *info* field of a Node x. If s is an execution of line 30 or 31, s sets $x.info$ to the Info object created at line 17. Suppose s is a flag CAS of some Info object I at line 106 inside an invocation of Help(I). If Help(I) is called at line 34 or 46, I is created at line 33 or 45. Otherwise, Help(I) is called at line 94 inside an invocation of CheckInfo(-, *oldInfo*) where *oldInfo* is an array of Info objects. Since I is equal to an element of *oldInfo*, I is non-null by induction hypothesis 4 (and Lemma 7.5).

Statement 4 The only steps s that set a field of an Info object I are the steps that create I at line 33 or 45. At line 33 or 45, $I.nodes[i]$ and $I.oldInfo[i]$ are set to local variables, which are non-null by induction hypothesis 7, for $0 \leq i < 3$.

If I is created at line 33, $I.newNxt$ and $I.newPrv$ are set to Node objects that are created at line 30 and 31 respectively. If I is created at line 45, $I.newNxt$ and $I.newPrv$ are set to local variables which are non-null by induction hypothesis 7.

Statement 5 After a Node is created, the only line that can set the Node's *copy* field is line 112, which sets it to a non-null value, by induction hypothesis 4 (and Lemma 7.5). When a Node's *state* field is set to copied (at line 113), its *copy* field is non-null because line 112 has been executed.

Statement 6 We consider steps s that set the *node* field of a Cursor c to x. If s sets $c.node$ to x at line 35, 47, 58, 59 or 65, since x is an initialized local variable, x is non-null by induction hypothesis 7.

If s sets $c.node$ to x at line 54, x was read from the *copy* field of some Node y before C_i. Then, $y.state$ is copied earlier when the test on line 54 is evaluated. So, x was non-null by induction hypothesis 5.

If s sets $c.node$ at line 68 or 73, $x = Head.nxt$ earlier. Since $Head \neq Tail$, x is non-null by induction hypothesis 1.

If s sets $c.node$ to x at line 85, x is read from the *copy* field of some Node z.

Then, $z.state$ is copied earlier at line 83. By induction hypothesis 5, $z.copy$ at line 85 is non-null.

If s sets $c.node$ to x at line 88, x is read from the nxt field of a Node z. Since $z.state$ is marked on line 86, $z \neq Tail$ by Invariant 7.6.5. Since $z.nxt = x$ before C_i and $z \neq Tail$, x is non-null by induction hypothesis 1.

Statement 7 We consider steps s that set local variables of type Node or Info. Suppose s sets a local variable y to the value x.

If s sets y to x by the invocation of the UpdateCursor routine at line 25, 39, 50, 62 or 76, x is a local variable which is non-null by induction hypothesis 7.

Suppose s sets y to x at line 25, 39, 50, 62 or 76. Let $\langle node, nodeInfo, nxtNode, prvNode, \text{-}, \text{-}\rangle$ be the result of the call to UpdateCursor at that line. If $y = node$, since the $node$ field of a Cursor is set to $node$ when line 85 or 88 is executed before C_i, $node$ is non-null by induction hypothesis 6. If $y = nodeInfo$, since $nodeInfo$ was read from the $info$ field of a Node when line 89 is executed before C_i, $nodeInfo$ is non-null by induction hypothesis 3. If y is $nxtNode$ or $prvNode$, y was read from the nxt or prv field of $node$ when line 90 is executed before C_i. Since the $node$ field of a Cursor is set to $node$ when line 85 or 88 is executed, $node$ is neither $Head$ nor $Tail$ by Invariant 7.6.10. So, y is non-null by induction hypothesis 1 and 2.

If s sets y to x at line 27 or 41, x is a local variable which is non-null by induction

hypothesis 7. If s sets y to x at line 28 or 42, x is either a local variable or read from the *info* field of a Node earlier, so x is non-null by induction hypothesis 3.

If s sets y to x by the invocation of the CheckInfo routine at line 29 or 43, x is a local variable which is non-null by induction hypothesis 7.

If s sets y at line 30, 31, 33 or 45, x is an object that has just been created.

If s sets y to x by the invocation of the Help routine at line 34, 46 or 94, x is a local variable which is non-null by induction hypothesis 7.

If s sets y to x at line 56, x is read from the *prv* field of some Node z earlier. Since the operation does not return at line 52, $z \neq Head$. By induction hypothesis 2, x is non-null.

If s sets y to x at line 89, x is read from the *info* field of a Node, so x is non-null by induction hypothesis 3. □

We assign a real number to each Node that is called its *abstract value*. The abstract value of a Node never changes. The abstract value is only used in the correctness proof. We prove an invariant that Nodes in the list are sorted by abstract values.

The abstract value of a Node x is denoted $x.absVal$. The $Head$, $EOLnode$ and $Tail$ are created at line 18, 19 and 20 respectively and $Head.absVal = 0$, $EOLnode.absVal = 1$ and $Tail.absVal = 2$. In addition, Nodes can be created at line 30 and 31. Let $\langle node$, -, -, $prvNode$, -, -$\rangle$ be the result returned by the

preceding call to UpdateCursor at line 25. The Node created at line 30 has the abstract value $\frac{prvNode.absVal+node.absVal}{2}$ and the Node created at line 31 has the same abstract value as *node*.

Invariant 7.8. *Let x and y be Node objects. If $x.nxt = y$ or $y.prv = x$, then $x.absVal < y.absVal$.*

Proof. By definition, $Head.absVal < EOLnode.absVal < Tail.absVal$. $EOLnode.prv$ is set to $Head$ at line 19, and $Tail.prv$ is set to $EOLnode$ at line 20. $Head.nxt$ is set to $EOLnode$ at line 21 and $EOLnode.nxt$ is set to $Tail$ at line 22. So, the invariant is true in configuration C_0.

We now show that each step s preserves the invariant. The only steps that can set *nxt* or *prv* fields of Nodes are line 30, 31 and 32 in the InsertBefore operation and line 114 and 115 in the Help routine.

Let s be an execution of line 30, 31 or 32. Let *newNode* and *nodeCopy* be the Nodes that are created at line 30 and 31. Let $\langle node$, -, $nxtNode$, $prvNode$, -, -$\rangle$ be the result returned by the preceding call to UpdateCursor at line 25. Since $node.prv = prvNode$ and $node.nxt = nxtNode$ at line 90 inside UpdateCursor and the invariant is true at all configurations before s, $prvNode.absVal < node.absVal < nxtNode.absVal$. By definition, $newNode.absVal = \frac{prvNode.absVal+node.absVal}{2}$, and $nodeCopy.absVal = node.absVal$. So, $prvNode.absVal < newNode.absVal < node.absVal = nodeCopy.absVal <$

$nxtNode.absVal$. Line 30 sets $newNode.prv$ to $prvNode$. Line 31 sets $nodeCopy.nxt$ and $nodeCopy.prv$ to $nxtNode$ and $newNode$, respectively. Line 32 sets $newNode.nxt$ to $nodeCopy$. Thus, all of these lines preserve the invariant.

We now consider an execution of line 114 or 115. Let I be an Info object. If s is a successful forward CAS at line 114 inside Help(I), s sets $I.nodes[0].nxt$ to $I.newNxt$. If s is a successful backward CAS at line 115 inside Help(I), s sets $I.nodes[2].prv$ to $I.newPrv$. We show that $I.nodes[0].absVal < I.newNxt.absVal$ and $I.newPrv.absVal < I.nodes[2].absVal$. We consider two cases according to what update operation created I.

Case 1: I is created by an InsertBefore operation. Let $\langle node$, -, $nxtNode$, $prvNode$, -, -$\rangle$ be the result returned by the call to UpdateCursor that belongs to I. Let $newNode$ and $nodeCopy$ be the Nodes that are created at line 30 and 31. Then, $I.newNxt = newNode$, $I.newPrv = nodeCopy$, $I.nodes[0] = prvNode$ and $I.nodes[2] = nxtNode$. Since $node.prv = prvNode$ and $node.nxt = nxtNode$ at line 90 inside the call to UpdateCursor (before s) and the invariant is true at all configurations before s, $prvNode.absVal < node.absVal < nxtNode.absVal$. By definition, $newNode.absVal = \frac{prvNode.absVal + node.absVal}{2}$ and $nodeCopy.absVal = node.absVal$. So, $prvNode.absVal < newNode.absVal$ and $nodeCopy.absVal = node.absVal < nxtNode.absVal$.

Case 2: I is created by a Delete operation. Let $\langle node$, -, $nxtNode$, $prvNode$,

-, -⟩ be the result returned by the call to UpdateCursor that belongs to I. Then, $I.nodes[0] = prvNode$, $I.nodes[2] = nxtNode$, $I.newNxt = nxtNode$ and $I.newPrv = prvNode$. Since $node.prv = prvNode$ and $node.nxt = nxtNode$ at line 90 inside the call to UpdateCursor (before s) and the invariant is true at all configurations before s, $prvNode.absVal < nxtNode.absVal$. □

Let ⟨$node$, -, $nxtNode$, $prvNode$, -, -⟩ be the result returned by the call to UpdateCursor that belongs to an Info object I. Since $node.prv = prvNode$ and $node.nxt = nxtNode$ at line 90 and $I.nodes$ is set to $\langle prvNode, node, nxtNode\rangle$ at line 27 or 41, by Invariant 7.8, we have the following corollary. Later, this corollary will be used to show that an update operation described by I tries to flag Nodes in $I.nodes$ in order of their abstract values.

Corollary 7.9. *Let I be an Info object. For $0 \leq i \leq 1$, $I.nodes[i].absVal < I.nodes[i+1].absVal$.*

When a new Node is created, it either copies $absVal$ from an existing Node or picks an $absVal$ between two existing Nodes' $absVal$s. The list initially includes $Head$, $EOLnode$ and $Tail$. Since $Head.absVal = 0$, $EOLnode.absVal = 1$ and $Tail.absVal = 2$, we have the following observation.

Observation 7.10. *For every Node x, $0 \leq x.absVal \leq 2$.*

By Invariant 7.8 and Observation 7.10, we have the following corollary that says the abstract values of reachable Nodes are distinct.

Corollary 7.11. *At all configurations, no two reachable Nodes have the same $absVal$, and for each reachable Node x, there is exactly one reachable Node whose nxt field is x (unless $x = Head$).*

7.2 Behaviour of Flag CAS Steps

In our implementation, before an update changes the list, it flags three Nodes to prevent others from changing the same part of the list simultaneously. To show our implementation is correct, we must show that while the update described by an Info object I is changing the list, the Nodes in $I.nodes$ are flagged for I. In this section, we show how flag CAS steps work.

Let I be an Info object created at line 33 or 45. The entries of $I.oldInfo$ are set to values that are read from the *info* field of some Nodes at line 89, 28 or 42. The elements of $I.oldInfo$ are used as the expected values for the flag CAS steps of I.

Observation 7.12. *Let I be an Info object created at line 33 or 45. Then, for each i, $I.oldInfo[i]$ was stored in the info field of $I.nodes[i]$ at some configuration before the call to* CheckInfo *that belongs to I.*

Since CAS steps are used to change the *info* fields of Nodes, some of the later

parts of our proof of correctness will rely on the fact that there is no ABA problem in this field. Next, we prove this fact.

Lemma 7.13. *Let x be a Node. Then, $x.info$ is never set to a value that it has had previously.*

Proof. Only a successful flag CAS at line 106 can change an *info* field. Consider such a step inside a call to HELP(I) that changes $x.info$ to I. By Lemma 7.5, I is created at line 33 or 45. By Observation 7.12, the expected value used for this flag CAS was read from the *info* field of x before I was created. So, every time $x.info$ is changed from some Info object I' to another Info object I, I was created after the creation of I'. So, $x.info$ is never set to a value that it has had previously. □

By Lemma 7.13, if a flag CAS step successfully sets the *info* field of a Node, the *info* field of the Node has not been changed since the step that read the expected value for the flag CAS. Thus, the ABA problem on the *info* fields is avoided.

In the implementation, update operations might help one another to set the *info* fields of some Nodes by calling the HELP routine at line 94. So, there might be several CAS steps that try to change the *info* field of some Node from a value *old* to some value *new*. We show that only the first CAS step among the CAS steps of this group can succeed in changing the *info* field of the Node from *old* to *new*.

Lemma 7.14. *Let x be a Node. Assume a group of CAS steps tries to change*

x.info from a value old to a value new. Only the first CAS step in this group can succeed.

Proof. Suppose a group of CAS steps each tries (at line 106) to change $x.info$ from old to new. Then, there is some i such that $new.nodes[i] = x$ and $new.oldInfo[i] = old$. By Observation 7.12, $x.info = old$ at some configuration before new is created. So, all CAS steps that try to change $x.info$ from old to new occur after a configuration in which the value of $x.info$ was old. After a CAS step changes $x.info$ from old to some value, by Lemma 7.13, $x.info$ is not set to old again. If the first CAS step among the CAS steps in the group succeeds, no subsequent CAS step in the group can change $x.info$ from old to new. If the first CAS step among the CAS steps in the group fails, $x.info$ has already been changed from old to some other value and no other CAS step in the group will succeed. □

To ensure flagging works properly, we show that when a flag CAS succeeds and changes the *info* field of a Node from an Info object I to another Info object I', the operation described by I is not in progress, but the operation described by I' is in progress.

Lemma 7.15. *Let x be a Node. If a flag CAS on Node x succeeds then x.info.status is not inProgress in the configuration immediately before the flag CAS.*

Proof. Only a flag CAS of I at line 106 can change $x.info$. If a flag CAS of I changes

$x.info$ from old to I, then an element of $I.oldInfo$ is old. Since I is not equal to dum (by Lemma 7.5), I is created at line 33 or 45. So, the call to CheckInfo that belongs to I returns true. So, $old.status$ was not inProgress at line 93 inside the call to CheckInfo. By Observation 7.2, $old.status$ is never set to inProgress after that, so $x.info.status$ is not inProgress in the configuration immediately before the flag CAS. □

Lemma 7.16. *Let I be an Info object. When a successful flag CAS of I occurs, $I.status$ is inProgress.*

Proof. By Lemma 7.5, Help(dum) is never called, so the claim is vacuously is satisfied when I is dum.

Every other Info object I is created at line 33 or 45 with $I.status$ initially inProgress. Suppose $I.status$ is not inProgress for the first time in configuration C_j. Then, the step preceding C_j is an execution of line 116 or 118 inside some invocation H of Help(I). Let k be an index between 0 and 2. We must show that no successful flag CAS of I on $I.nodes[k]$ occurs after C_j. We consider two cases.

If H performs a flag CAS on $I.nodes[k]$, it will do so before C_j, and no flag CAS of I on $I.nodes[k]$ can succeed after C_j (by Lemma 7.14), so the claim is true.

If H does not perform a flag CAS on $I.nodes[k]$, then, H must find that $I.nodes[m].info \neq I$ at line 107 of some iteration $m < k$ of the loop and set $doPtrCAS$ to false. Let C_i be the configuration preceding this execution of line

107 and let $x = I.nodes[m]$. Note that C_i precedes C_j.

By Lemma 7.15, $x.info$ cannot be I at any configuration before C_i, because then $x.info$ would have to be changed from I to some other value before C_i, and $I.status$ is inProgress at all configurations before C_i. Moreover, $x.info$ cannot be set to I at any time after C_i, since H performs a flag CAS of I on x before C_i, and no subsequent flag CAS can set $x.info$ to I by Lemma 7.14. Thus, $x.info$ is never equal to I. This means that no invocation of HELP(I) will reach the end of iteration m of the loop without setting $doPtrCAS$ to false, so no invocation of HELP(I) will reach iteration k of the loop. Thus, no flag CAS of I on $I.nodes[k]$ will ever be performed. □

Let I be an Info object. We say I is a *successful Info object* in an execution if a flag CAS of I on $I.nodes[2]$ succeeds in that execution. If $I.nodes[2].info$ is successfully set to I, then it follows from the code that the *info* fields of $I.nodes[0]$ and $I.nodes[1]$ were set to I earlier. So, if I is successful, there must be successful flag CAS steps of I on all three Nodes in $I.nodes$.

Lemma 7.17. *Let I be a successful Info object. Then, no invocation of* HELP*(I) sets $I.status$ to aborted at line 118.*

Proof. Let H be an invocation of HELP(I). To derive a contradiction, suppose H sets $I.status$ to aborted on line 118. Then, $doPtrCAS$ is false when H executes line

109 and $I.status$ is inProgress when H evaluates the test on line 117. Thus, H sets the $doPtrCAS$ variable to false at line 107 when it sees that $I.nodes[k].info \neq I$ (for some k). Since I is a successful Info object, $I.nodes[k]$ was set to I. By Lemma 7.14, the first flag CAS of I on $I.nodes[k]$ succeeds. Since H executes a flag CAS of I on $I.nodes[k]$ before setting $doPtrCAS$ to false, the first flag CAS of I on $I.nodes[k]$ succeeds before H executes line 107. Since $I.nodes[k].info \neq I$ at line 107, $I.nodes[k].info$ is changed from I to another value before H executes line 107. Since $I.status$ is inProgress when H executes line 117, $I.status$ is inProgress at all configurations before that (by Observation 7.2). So, $I.status$ is inProgress when $I.nodes[k].info$ is changed from I to another value, contradicting Lemma 7.15. □

If I is not a successful Info object, no call to HELP(I) can see $doPtrCAS =$ true at line 109. So, no call to HELP(I) sets $I.status$ to committed on line 116. Thus, we have the following corollary of Lemma 7.17, which states that if $I.status$ is set to committed, it cannot be set to aborted after that and vice versa.

Corollary 7.18. *Let I be an Info object. There cannot be an execution of line 116 and an execution of line 118 that both belong to I.*

Let s be the first execution of line 110, 112, 113, 114, 115 or 116 belonging to an Info object I. Next, we wish to show that s occurs only while all Nodes in $I.nodes$ are flagged by I. First, we show that s occurs only after all Nodes in $I.nodes$ get

flagged by flag CAS steps of I. Then, we prove that $I.status$ is inProgress when s occurs. After that, we show that all Nodes in $I.nodes$ are still flagged by I when s occurs.

Lemma 7.19. *Let I be an Info object and $\ell \in \{110, 112, 113, 114, 115, 116\}$. The info field of each Node in $I.nodes$ is set to I at some time before the first execution of line ℓ belonging to I.*

Proof. The HELP routine executes line ℓ only if the $doPtrCAS$ variable is true at line 109. If the *info* field of $I.nodes[i] \neq I$ for any i, $doPtrCAS$ gets set to false at line 107 and the loop terminates. So, $doPtrCAS$ is true at line 109 only if the *info* fields of all Nodes in $I.nodes$ were successfully set to I. □

Lemma 7.20. *Let I be an Info object, $\ell \in \{110, 112, 113, 114, 115, 116\}$ and suppose C_i is the configuration just prior to the first execution of line ℓ belonging to I. Then, $I.status$ is inProgress at C_i.*

Proof. Since HELP(I) is called, I is created at line 33 or 45 (by Lemma 7.5). So, $I.status$ is initially inProgress. Let H be the invocation of HELP(I) that first executes line ℓ. No call to HELP(I) can reach line 116 to set $I.status$ to committed before H executes line ℓ. Since H must have seen $doPtrCAS$ = true at line 109, the first flag CAS of I on $I.nodes[2]$ succeeds and I is a successful Info object. So, no call to HELP can ever set $I.status$ to aborted, by Lemma 7.17. Thus, $I.status$

is still inProgress just before H executes line ℓ. □

Lemma 7.21. *Let I be an Info object, $\ell \in \{110, 112, 113, 114, 115, 116\}$ and C_i be the configuration that immediately precedes the first execution of line ℓ belonging to I. Then, for each k, $I.nodes[k].info = I$ at C_i.*

Proof. By Lemma 7.19, $I.nodes[k].info$ is set to I some time before C_i. By Lemma 7.20, $I.status$ is inProgress at C_i. By Observation 7.2, $I.status$ is inProgress in every configuration before C_i. By Lemma 7.15, $I.nodes[k].info$ can be changed from I to another value only when $I.status$ is not inProgress. So, $I.nodes[k].info$ is not changed from I to another value before C_i and $I.nodes[k].info = I$ at C_i. □

Let I be an Info object. By Lemma 7.20 and 7.21, all three Nodes in $I.nodes$ are flagged for I at the configurations immediately before the first forward CAS of I and the first backward CAS of I. In the next section, we shall show that *only* the first forward CAS of I and the first backward CAS of I succeed among all pointer CAS steps of I. We wish to show if an operation described by I performs the first forward CAS of I or the first backward CAS of I, no other operation changes the pointers of any Node in $I.nodes$ while those Nodes are flagged for I. The following lemma will be used to argue later that the updates are atomic because no change to the two pointers affected by the pointer CAS steps of I has occurred since those pointers were read earlier.

Lemma 7.22. *Let I be a successful Info object and $\ell \in \{110,112,113,114,115,116\}$. Suppose, for some k, $I.nodes[k].info = I.oldInfo[k]$ in some configuration C_i. Then, at any configuration after C_i but not after the first execution of line ℓ belonging to I, $I.nodes[k].info$ is either $I.oldInfo[k]$ or I.*

Proof. Since I is successful, a flag CAS step S of I changes $I.nodes[k].info$ from $I.oldInfo[k]$ to I. By Lemma 7.13, $I.nodes[k].info$ is never set to $I.oldInfo[k]$ after S, so C_i is before S. Let C be a configuration after C_i but not after the first execution of line ℓ belonging to I. By Lemma 7.13, $I.nodes[k].info = I.oldInfo[k]$ at all configurations between C_i and S. So, if C is before S, $I.nodes[k].info = I.oldInfo[k]$ at C.

Suppose C is after S. Since there is no execution of line 116 belonging to I before C, $I.status$ is not set to committed before C. Since I is a successful Info object, by Lemma 7.17, $I.status$ is never set to aborted. So, $I.status$ is inProgress at all configurations before C. By Lemma 7.15, $I.nodes[k].info = I$ at all configurations between S and C. □

The following lemma shows that when $I.nodes[1].state$ is set for the first time inside a call to Help(I), $I.nodes[1].state$ is changed from ordinary to marked or copied. It follows that, for an Info object $I' \neq I$, $I.nodes[1].state$ cannot be set inside a call to Help(I') after that.

Lemma 7.23. *Let I be a successful Info object and x be $I.nodes[1]$. Then, in each configuration that is after x is created but is not after an execution of line 110 or 113 belonging to I, $x.state$ is ordinary.*

Proof. Let C be any configuration that is after x is created but is not after an execution of line 110 or 113 belonging to I. We show that $x.state$ is ordinary in C.

By Lemma 7.5, I is created at line 33 or 45. Thus, the invocation of CHECKINFO that belongs to I returns true. Let C_j be the configuration after the second execution of line 93 inside this CHECKINFO, which reads $I.oldInfo[1].status$ to be different from inProgress. When CHECKINFO executes line 97 for the second time (after C_j), $x.state$ is ordinary. So, by Observation 7.3, $x.state$ is ordinary at all configurations before C_j. So, if C is before C_j, the lemma is true. Suppose C is after C_j.

By Observation 7.12, the first configuration in which $x.info = I.oldInfo[1]$ is before the call to the invocation of CHECKINFO that belongs to I. Since I is a successful Info object, by Lemma 7.22, $x.info$ is either $I.oldInfo[1]$ or I between the first configuration in which $x.info = I.oldInfo[1]$ and C. So, $x.info$ is either I or $I.oldInfo[1]$ in all configurations between C_j and C.

To derive a contradiction, assume some invocation of HELP(I') writes $x.state$ between C_j and C. By definition of C, I' must be different from I. Let C_k be the configuration after the first such write. Then, C_k must be the first execution of line

110 or 113 belonging to I'. By Lemma 7.21, $x.info = I'$ at C_k. Since C_k is between C_j and C, $x.info$ is either I or $I.oldInfo[1]$ in C_k. Since $I' \neq I$, $I' = I.oldInfo[1]$. Since $I.oldInfo[1].status$ was not inProgress at C_j and $I' = I.oldInfo[1]$, $I'.status$ is not inProgress at C_k (by Observation 7.2), contradicting Lemma 7.20. □

The following lemma shows no other operation sets the *state* of $I.nodes[0]$ or $I.nodes[2]$ before the operation described by I completes its update.

Lemma 7.24. *Let I be a successful Info object. For $i = 0$ and 2, $I.nodes[i].state$ is ordinary in any configuration that is after the Node $I.nodes[i]$ is created and is not after a backward CAS of I.*

Proof. Let i be 0 or 2 and let $x = I.nodes[i]$. Let C be any configuration after x is created but not after a backward CAS of I. Since the call to CheckInfo that belongs to I returns true, $x.state$ is ordinary at an execution of line 97 inside that invocation of CheckInfo. Let C_k be the configuration that follows that execution of line 97. By Observation 7.3, $x.state$ is ordinary at all configurations before C_k. By Observation 7.12, $x.info$ was $I.oldInfo[i]$ at some configuration before C_k. Since I is a successful Info object, by Lemma 7.22, $x.info$ is either $I.oldInfo[i]$ or I between C_k and C. By Corollary 7.9, $x \neq I.nodes[1]$. So, $x.state$ is not set to marked or copied inside any call to Help(I).

To derive a contradiction, assume some invocation of Help(I') writes $x.state$

between C_k and C (for some Info object $I' \neq I$). Let C_j be the configuration after the first such write. Then, C_j must be the first execution of line 110 or 113 belonging to I'. By Lemma 7.21, $x.info = I'$ at C_j. Since $x.info$ is either $I.oldInfo[i]$ or I between C_k and C and $I \neq I'$, $I' = I.oldInfo[i]$. $I.oldInfo[i].status$ was not inProgress at an execution of line 93 before C_k. By Observation 7.2, $I.oldInfo[i].status$ is not inProgress at C_j, contradicting Lemma 7.20. □

We shall show in the next section that $I.nodes[1]$ is not reachable after the first forward CAS of I. The following lemma intuitively says that after the operation described by I removes $I.nodes[1]$ from the list, the Node remains permanently flagged for I.

Lemma 7.25. *Let I be an Info object. If there is any forward CAS of I, $I.nodes[1].info = I$ at all configurations after the first forward CAS of I.*

Proof. Let $x = I.nodes[1]$ and let C_i be the configuration preceding the first forward CAS of I. By Lemma 7.21, $x.info = I$ at C_i. To derive a contradiction, assume C_j is the first configuration after C_i in which $x.info \neq I$. The step preceding C_j is a successful flag CAS inside Help(I') for some Info object $I' \neq I$. Then, for some k, $I'.nodes[k] = x$ and $I'.oldInfo[k] = I$.

So, $x.state$ was ordinary at an execution of line 97 inside the invocation CI of CheckInfo that belongs to I'. Let C_k be the configuration that follows that

execution of line 97. By Observation 7.3, $x.state$ is set to marked or copied inside HELP(I) after C_k by Lemma 7.23 (since $x.state$ is ordinary in C_k), but before C_i. Since CI returned true, $I.status$ was not inProgress when CI executed line 93 before C_k and hence before C_i. By Observation 7.2, $I.status$ is not inProgress in C_i, contradicting Lemma 7.20. □

Before replacing a Node by a new copy, its *copy* field is set to the new copy of the Node. An operation that reaches the Node after it is replaced by a new copy can follow the *copy* field to reach the new copy of the Node. So, the *copy* field of a Node must not be set to two different Nodes. Note a new copy of a Node can also be replaced by another copy of that Node.

Lemma 7.26. *Executions of line 112 belonging to two different Info objects do not write to the same Node.*

Proof. To derive a contradiction, assume that executions of line 112 belonging to two different Info objects I and I' both write to the same Node x. Then, $x = I.nodes[1] = I'.nodes[1]$. By the pseudo-code, $x.state$ is set to copied inside HELP(I) and $x.state$ is set to copied inside HELP(I'). By Observation 7.3, either $x.state$ is not ordinary before the first execution of line 113 belonging to I or $x.state$ is not ordinary before the first execution of line 113 belonging to I', contradicting Lemma 7.23. □

7.3 Behaviour of Pointer CAS Steps

To perform an update operation, the nxt and prv field of some Nodes must be changed. In this section, we show the pointer CAS steps change the nxt and prv field of Nodes as expected and there is no ABA problem on the nxt or prv field of Nodes. We also establish some invariants regarding reachability.

First, we have the following lemma, which shows that each successful pointer CAS changes the nxt or prv field of a Node from some old value to some new value that is different from the old value.

Lemma 7.27. *Let I be an Info object. Then, $I.nodes[1]$ is neither $I.newNxt$ nor $I.newPrv$.*

Proof. If I is created by an InsertBefore operation on line 33, then $I.nodes[1]$ was read inside the invocation of UpdateCursor that belongs to I and $I.newNxt$ and $I.newPrv$ are created at line 30 and 31 respectively after UpdateCursor returns. So, $I.nodes[1]$ is not $I.newNxt$ or $I.newPrv$.

If I is created by a Delete operation on line 45, then the value of $I.newNxt$ and $I.newPrv$ are read from the nxt or prv field of $I.nodes[1]$ at line 90. By Invariant 7.8, $I.nodes[1]$ is not $I.newNxt$ or $I.newPrv$. □

Next, we show that several statements are true during any execution. Intuitively, the following lemma shows that the update operations change the list as described

in Figure 6.6 and 6.7. Since the lemma is quite technical, we give some intuition for its statements.

In the implementation, update operations might help one another to change the *nxt* or *prv* fields of Nodes by calling the HELP routine. So, there might be several CAS steps that try to change the *nxt* or *prv* field of some Node from a value *old* to some value *new*. Statement 1 and 2 of the following lemma show that only the first forward and first backward CAS of an Info object I succeed among all pointer CAS steps of I and there is no ABA problem on the *nxt* and *prv* field of a Node.

An insertion creates two Nodes at line 30 and 31. Intuitively, Statement 3 shows that these two new Nodes are not involved in pointer CAS steps of any other operation until the insertion that creates them is complete.

Statement 4 shows that the pointers that are read at line 90 preceding the creation of a successful Info object I are not changed after those reads, until the first forward CAS of I.

Statement 5a, 5b and 8 describe exactly which Nodes become reachable and unreachable by step 3 of Figure 6.6 and step 4 of Figure 6.7. Statement 6 shows that the *state* field of a Node is set before it becomes unreachable. Statement 7 states that if the three Nodes are successfully flagged for an update, the pointers of these Nodes look like as step 1 of Figure 6.6 and 6.7.

Statements 9 and 10 show that list pointers are consistent. Statement 10 says

that $x.prv.nxt = x$ for all reachable Nodes x. Statement 9 says that $x.nxt.prv = x$ except in between step 3 and step 4 of Figure 6.6 and step 4 and step 5 of Figure 6.7 when these pointers are temporarily inconsistent.

We show that $I.nodes[1]$ becomes unreachable by the first forward CAS of I (Statement 5a). Statement 11 proves that no process changes the nxt or prv field of $I.nodes[1]$ after that.

Statements 12 and 13 state some facts regarding reachability of Nodes. Statement 14 and 15 show that, for a Node x, the test $(x.prv.nxt \neq x)$ accurately determines whether x has become unreachable or not.

Lemma 7.28. *For all Info objects I and I', Cursors c, Nodes x and configurations C_i, the following statements are true.*

1. *The first forward CAS belonging to I and the first backward CAS belonging to I succeed. Each other pointer CAS belonging to I fails.*

2. *If a pointer CAS of the form CAS(-, -, x) succeeds, there is no successful pointer CAS of the form CAS(-, x, -) earlier.*

3. *Let I be an Info object created by an* InsertBefore *operation. The following types of steps can occur only after the first backward CAS of I:*

   (a) *a pointer CAS of $I' \neq I$ that changes the nxt or prv field of $I.newNxt$ or $I.newPrv$,*

(b) *a pointer CAS of* $I' \neq I$ *that changes the nxt or prv field of a Node to* $I.newNxt$ *or* $I.newPrv$*, or*

(c) *a pointer CAS of* $I' \neq I$ *that changes the nxt or prv field of a Node from* $I.newNxt$ *or* $I.newPrv$ *to another value.*

4. *Suppose* I *is a successful Info object. Let* C_i *and* C_j *be the two configurations after reading* $I.nodes[1].nxt$ *and* $I.nodes[1].prv$ *on line 90 inside the call to* UpdateCursor *that belongs to* I*, respectively. Then,*

(a) $I.nodes[1].nxt = I.nodes[2]$ *at all configurations that are after* C_i *but not after the first forward CAS of* I*.*

(b) $I.nodes[1].prv = I.nodes[0]$ *at all configurations that are after* C_j *but not after the first forward CAS of* I*.*

5. *The following statements are true.*

(a) *Suppose the step from* C_{i-1} *to* C_i *is the first forward CAS of an Info object* I*. Then, for all* j*,* $I.nodes[j]$ *is reachable at* C_{i-1} *and* $I.nodes[1]$ *is unreachable at* C_i *and both* $I.nodes[0]$ *and* $I.nodes[2]$ *are reachable at* C_i*.*

(b) *Suppose* x *is reachable at* C_{i-1} *and unreachable at* C_i*. Then, the step from* C_{i-1} *to* C_i *is the first forward CAS of an Info object* I *such that* $I.nodes[1] = x$*.*

6. *If x is unreachable at C_i and was reachable at C_{i-1}, $x.state$ is not ordinary at all configurations after C_{i-1}.*

7. *Suppose I is a successful Info object and C_j is the configuration after the return of the call to* CheckInfo *that belongs to I.*

   (a) *Any successful pointer CAS of $I' \neq I$ that is performed on $I.nodes[m]$ (for any m) or that changes the nxt or prv field of a Node from $I.nodes[m]$ to another value after C_j is preceded by a backward CAS of I.*

   (b) *For each m, $I.nodes[m]$ is reachable at all configurations that are after C_j but not after a forward CAS of I.*

   (c) *At C_j, $I.nodes[1].prv = I.nodes[0]$, $I.nodes[1].nxt = I.nodes[2]$, $I.nodes[0].nxt = I.nodes[1]$ and $I.nodes[2].prv = I.nodes[1]$.*

8. *Let I be an Info object that is created by an* InsertBefore *operation and C_i be the configuration after the first forward CAS of I. Then, the Nodes that are reachable at C_i that were not reachable at C_{i-1} are exactly $I.newNxt$ and $I.newPrv$.*

9. *Suppose (1) $x \neq Tail$,*

   (2) x is reachable in C_i and

   (3) there is no Info object I such that $x = I.newPrv$ and C_i is after a

forward CAS of I but not after any backward CAS of I.

Then, $x.nxt.prv = x$ in C_i.

10. *If $x \neq Head$ and x is reachable in C_i, then $x.prv.nxt = x$ in C_i.*

11. *Let I be a successful Info object. No pointer CAS of an Info object $I' \neq I$ changes the nxt or prv field of $I.nodes[1]$ after the first forward CAS of I.*

12. *Let node, nxtNode and prvNode be the three Nodes returned by a call to* UpdateCursor*, respectively. Then, node was reachable in some configuration before the last execution of line 82 and nxtNode was reachable in some configuration before reading the nxt pointer on line 90 and prvNode was reachable in some configuration before reading the prv pointer on line 90.*

13. *If c.node points to x in C_i, then x was reachable in some configuration before C_i.*

14. *If there are three configurations C_{i_1}, C_{i_2}, C_{i_3} with $i_1 \leq i_2 \leq i_3$ such that*

    - *$x \neq Head$ is reachable in C_{i_1},*
    - *$x.prv = y$ in C_{i_2}, and*
    - *$y.nxt \neq x$ in C_{i_3},*

    then, in all configurations after C_{i_3}, $x.prv.nxt \neq x$.

15. *(Node $x \neq Head$ was reachable at some configuration earlier than C_i and $x.prv.nxt \neq x$ in configuration C_i) if and only if there is an Info object I such that*

    (a) $x = I.nodes[1]$, *and*

    (b) the first forward CAS belonging to I occurred before C_i.

Proof. The initialization of the data structure (line 17-21) ensures Statement 9 and 10 are true in the initial configuration C_0. All other statements are trivially true for the prefix of an execution containing 0 steps.

Here, if we assume an Info object I is successful in a prefix of an execution, the first flag CAS of I on $I.nodes[2]$ succeeds in that prefix of the execution. We assume all statements hold in a prefix of an execution up to C_{k-1} ($k \geq 1$). We show that each statement is true for the prefix up to C_k. Let S_k be the step preceding C_k.

Statement 1 Suppose the step S_k is a pointer CAS of I. First, we show if S_k is neither the first forward nor the first backward CAS of I, S_k fails. If S_k is a forward CAS but not the first of I, induction hypothesis 1 and Lemma 7.27 imply that the first forward CAS of I changed $I.nodes[0].nxt$ from $I.nodes[1]$ to a different value before C_{k-1}. By induction hypothesis 2, $I.nodes[0].nxt$ is not set to $I.nodes[1]$ after the first forward CAS of I. Thus, S_k fails. An identical argument applies if S_k is a backward CAS.

Next, we show that S_k succeeds if it is the first forward CAS step of I. (An identical argument applies if S_k is the first backward CAS of I.) Let x be $I.nodes[0]$. Let C_j be the configuration at the end of the call to CheckInfo that belongs to I. By Lemma 7.21, I is successful in the prefix of the execution up to C_k. By induction hypothesis 7c, $x.nxt$ equals $I.nodes[1]$ in C_j. We show that $x.nxt$ remains equal to $I.nodes[1]$ until S_k. By Observation 7.12, $x.info = I.oldInfo[0]$ at some configuration before C_j. By Lemma 7.22, $x.info$ is either $I.oldInfo[0]$ or I between C_j and C_k. By Lemma 7.21 and induction hypothesis 1, only a forward CAS of $oldInfo[0]$ or I can change the nxt field of x between C_j and C_{k-1}. The call to CheckInfo that belongs to I returns true at line 29 or 43. So, $I.oldInfo[0].status$ was not inProgress at some time during this call to CheckInfo. By Observation 7.2, $I.oldInfo[0].status$ is not inProgress at all configurations after C_j. By Lemma 7.20, the first forward CAS of $I.oldInfo[0]$ does not occur between C_j and C_k. So, no forward CAS changes $x.nxt$ between C_j and C_{k-1}. So, $x.nxt = I.nodes[1]$ at C_{k-1} and S_k succeeds.

Statement 2 Suppose S_k is a successful pointer CAS of I of the form CAS(-, -, x). If S_k is the first backward CAS of I, which has the form CAS($I.nodes[2].prv$, -, $I.newPrv$), then the first forward CAS of I, which is of the form CAS($I.nodes[0].nxt$, $I.nodes[1]$, -), does not violate the claim since

$I.nodes[1] \neq I.newPrv$ (by Lemma 7.27). Since S_k is either the first forward CAS of I or the first backward CAS of I (by Statement 1), it remains to show that there is no earlier successful pointer CAS of $I' \neq I$ of the form CAS($I'.nodes[k].ptr$, x, -).

Suppose I is created by an InsertBefore operation. Then, x is either $I.newNxt$ or $I.newPrv$. By induction hypothesis 3c, no pointer CAS of I' changes the nxt or prv field of a Node from x to another value before C_{k-1}.

If I is created by a Delete operation, x is either $I.nodes[0]$ or $I.nodes[2]$. By Lemma 7.21, $x.info = I$ in configuration C_k. If there were a pointer CAS of $I' \neq I$ of the form CAS($I'.nodes[k].ptr$, x, -) before C_k, then $x = I'.nodes[1]$ and $x.info$ would be I' at C_k by Lemma 7.25, a contradiction.

Statement 3 Let I be an Info object created by an InsertBefore operation. Suppose S_k is a successful pointer CAS of $I' \neq I$ of the form (a), (b) or (c) described in Statement 3. We shall prove that the first backward CAS of I occurs before S_k.

Before I is created, $I.newNxt.prv$ is set to $I.nodes[0]$ at line 30, the nxt and prv field of $I.newPrv$ are set to $I.nodes[2]$ and $I.newNxt$ respectively at line 31 and $I.newNxt.nxt$ is set to $I.newPrv$ at line 32. By Lemma 7.21, I' is successful in the prefix of the execution up to C_k. By Statement 1, proved above, S_k must be the first forward or the first backward CAS of I'. Next, we have the following claim. Claim 1: If, for some l and m, $I'.nodes[l] = I.nodes[m]$, then there is a backward

CAS of I before S_k.

Proof of Claim 1. To derive a contradiction, assume there is no backward CAS of I before S_k. First, we show that the first forward CAS of I occurs before C_{k-1}. If S_k is the form (a) described in Statement 3, $I.newPrv$ or $I.newNxt$ is $I'.nodes[0]$ or $I'.nodes[2]$. If S_k is the form (b) described in Statement 3, since $I.newPrv$ and $I.newNxt$ are not created at line 30 or 31 that precedes the creation of I', I' is created by a DELETE operation. So, $I.newPrv$ or $I.newNxt$ is $I'.nodes[0]$ or $I'.nodes[2]$. If S_k is the form (c) described in Statement 3, $I.newPrv$ or $I.newNxt$ is $I'.nodes[1]$. So, by induction hypothesis 7b, either $I.newNxt$ or $I.newPrv$ is reachable at the configuration after the call to CHECKINFO that belongs to I' returns. Since there is no backward CAS of I before C_{k-1}, by induction hypothesis 3b, no pointer CAS of an Info object other than I makes $I.newPrv$ or $I.newNxt$ reachable before C_{k-1}. So, $I.newPrv$ and $I.newNxt$ become reachable by a forward CAS of I before C_{k-1}.

Since the first forward CAS of I occurs before C_{k-1}, I is successful in the prefix of the execution up to C_k (by Lemma 7.21). Since C_{k-1} is after the first forward CAS of I but not after the first backward CAS of I, by Lemma 7.22, $I.nodes[m].info$ is either I or $I.oldInfo[m]$ at C_{k-1}. Since $I.nodes[m].info$ is changed from $I.oldInfo[m]$ to I before C_{k-1}, $I.nodes[m].info \neq I.oldInfo[m]$ at C_{k-1} (by Lemma 7.13). So, $I.nodes[m].info = I$ at C_{k-1}. Since $I \neq I'$, this contradicts

the fact that $I.nodes[m].info = I'.nodes[l] = I'$ at C_{k-1} (by Lemma 7.21). This completes the proof of Claim 1.

Now, we consider different cases depending on the form of S_k.

Case 1: S_k changes the nxt or prv field of $I.newNxt$ or $I.newPrv$.

Case 1A: S_k is the first forward CAS of I' and changes $I.newNxt.nxt$. So, $I'.nodes[0] = I.newNxt$. Then, by induction hypothesis 7c, at line 101 preceding the creation of I', $I'.nodes[1].prv = I.newNxt$ and $I'.nodes[1].nxt = I'.nodes[2]$. By induction hypothesis 3b, if no backward CAS of I is before S_k, then before S_k, the only Node whose prv field is $I.newNxt$ is $I.newPrv$. So, $I'.nodes[1] = I.newPrv$. Since $I.newPrv.nxt$ can only be $I.nodes[2]$ before S_k, $I'.nodes[2] = I.nodes[2]$. By Claim 1, there is a backward CAS of I before S_k.

Case 1B: S_k is the first forward CAS of I' and changes $I.newPrv.nxt$. Then, $I'.nodes[0] = I.newPrv$ and induction hypothesis 7c implies that $I'.nodes[1].prv$ was $I.newPrv$ at line 101 preceding the creation of I'. By induction hypothesis 3b, if no backward CAS of I is before S_k, the prv field of no Node is $I.newPrv$ before S_k. So, there is a backward CAS of I before S_k.

Case 1C: S_k is the first backward CAS of I' and changes $I.newNxt.prv$. Then, $I'.nodes[2] = I.newNxt$ and induction hypothesis 7c implies that $I'.nodes[1].nxt = I.newNxt$ at line 101 preceding the creation of I'. By induction hypothesis 3b, if no backward CAS of I is before S_k, $I.nodes[0].nxt$ must have been set to $I.newNxt$

by the first forward CAS of I and $I'.nodes[1] = I.nodes[0]$. By Claim 1, there is a backward CAS of I before S_k.

Case 1D: S_k is the first backward CAS of I' and changes $I.newPrv.prv$. Then, $I'.nodes[2] = I.newPrv$ and induction hypothesis 7c implies that $I'.nodes[1].nxt = I.newPrv$ and $I'.nodes[1].prv = I'.nodes[0]$ at line 101 preceding the creation of I'. By induction hypothesis 3b, if no backward CAS of I is before S_k, $I'.nodes[1] = I.newNxt$. By induction hypothesis 3a, $I'.nodes[1].prv$ is equal to $I.nodes[0]$ at all times before S_k. So, $I'.nodes[0] = I.nodes[0]$. By Claim 1, there is a backward CAS of I before S_k.

Case 2: S_k sets the nxt or prv field of a Node to $I.newNxt$ or $I.newPrv$. Since S_k is a pointer CAS of I' that stores a Node that was created on line 30 or 31 before the creation of $I \neq I'$, I' must have been created by a DELETE operation.

Case 2A: S_k is the first forward CAS of I' and sets the nxt field of a Node to $I.newNxt$. Then, $I'.nodes[2] = I.newNxt$ and induction hypothesis 7c implies that $I'.nodes[1].nxt = I.newNxt$ at line 101 preceding the creation of I'. By induction hypothesis 3b, if no backward CAS of I is before S_k, $I.nodes[1].nxt$ must have been set to $I.newNxt$ by the first forward CAS of I. So, $I'.nodes[1] = I.nodes[0]$. By Claim 1, there is a backward CAS of I before S_k.

Case 2B: S_k is the first backward CAS of I' and sets the prv field of a Node to $I.newNxt$. Then, $I'.nodes[0] = I.newNxt$ and induction hypothesis 7c implies that

$I'.nodes[1].prv = I.newNxt$ and $I'.nodes[1].nxt = I'.nodes[2]$ at line 101 preceding the creation of I'. By induction hypothesis 3b, if no backward CAS of I is before S_k, $I'.nodes[1] = I.newPrv$ and by induction hypothesis 3b, $I.newPrv.nxt$ is equal to $I.nodes[2]$ at all times before S_k, so $I'.nodes[2] = I.nodes[2]$. By Claim 1, there is a backward CAS of I before S_k.

Case 2C: S_k is the first forward CAS of I' and sets the nxt field of a Node to $I.newPrv$. Then, $I'.nodes[2] = I.newPrv$ and induction hypothesis 7c implies that $I'.nodes[1].nxt = I.newPrv$ and $I'.nodes[0].nxt = I'.nodes[1]$ at line 101 preceding the creation of I'. By induction hypothesis 3b, if no backward CAS of I is before S_k, $I'.nodes[1] = I.newNxt$ and $I'.nodes[0].nxt$ must have been set to $I.newNxt$ by the first forward CAS of I. So, $I'.nodes[0] = I.nodes[0]$. By Claim 1, there is a backward CAS of I before S_k.

Case 2D: S_k is the first backward CAS of I' and sets the prv field of a Node to $I.newPrv$. Then, $I'.nodes[0] = I.newPrv$ and induction hypothesis 7c implies that $I'.nodes[1].prv$ was $I.newPrv$ at line 101 preceding the creation of I'. By induction hypothesis 3b, if no backward CAS of I is before S_k, the prv field of no Node is $I.newPrv$ before S_k. So, C_k is after a backward CAS of I.

Case 3: S_k changes the nxt or prv field of a Node from $I.newNxt$ or $I.newPrv$ to another value.

Case 3A: S_k is the first forward or first backward CAS of I' and changes the

nxt or prv field of a Node from $I.newNxt$ to another value. Then, $I'.nodes[1] = I.newNxt$ and induction hypothesis 7c implies that $I.newNxt.prv = I'.nodes[0]$ at line 101 preceding the creation of I'. By induction hypothesis 3a, if no backward CAS of I is before S_k, $I'.nodes[0] = I.nodes[0]$. By Claim 1, there is a backward CAS of I before S_k.

Case 3B: S_k is the first forward or backward CAS of I' and changes the nxt or prv field of a Node from $I.newPrv$ to another value. Then, $I'.nodes[1] = I.newPrv$ and induction hypothesis 7c implies that $I.newPrv.nxt = I'.nodes[2]$ at line 101 preceding the creation of I'. By induction hypothesis 3a, if no backward CAS of I is before S_k, $I'.nodes[2] = I.nodes[2]$. By Claim 1, there is a backward CAS of I before S_k.

Statement 4 Suppose I is a successful Info object in the prefix of an execution up to C_k. Let C_i and C_j be the earlier configurations after reading $I.nodes[1].nxt$ and $I.nodes[1].prv$ on line 90 inside the call to CheckInfo that belongs to I respectively. Suppose C_k is a configuration that is not after the first forward CAS of I. Since I is a successful Info object in the prefix of an execution up to C_k, the $info$ fields of all Nodes in $I.nodes$ are set to I by flag CAS steps of I. Since I is successful, by Lemma 7.22, $I.nodes[1].info$ is $I.oldInfo[1]$ or I between C_i and C_k. So, By Lemma 7.21, only a pointer CAS of $I.oldInfo[1]$ can succeed on $I.nodes[1]$

between C_i and C_k (since C_k is not after the first forward CAS of I). Next, we show that no forward CAS of $I.oldInfo[1]$ succeeds on $I.nodes[1]$ between C_i and C_k and no backward CAS of $I.oldInfo[1]$ succeeds on $I.nodes[1]$ between C_j and C_k.

At C_i, $I.nodes[1].nxt = I.nodes[2]$ and, at C_j, $I.nodes[1].prv = I.nodes[0]$. By Statement 1, proved earlier, only the first forward CAS of $I.oldInfo[1]$ and the first backward CAS of $I.oldInfo[1]$ succeed. If $I.nodes[1].nxt$ (or $I.nodes[1].prv$) is changed from $I.nodes[2]$ (or $I.nodes[0]$) to some other Node after line 90, $I.nodes[2].state$ (or $I.nodes[0].state$) is not ordinary at all times after that (by Lemma 7.23). Since the call to CheckInfo that belongs to I returns true, for all m, $I.nodes[m].state$ was ordinary at some time during the loop on line 96–97. So, $I.nodes[1].nxt = I.nodes[2]$ and $I.nodes[1].prv = I.nodes[0]$ at some time during that call to CheckInfo. Before that, $I.oldInfo[1].status$ is not inProgress at line 93. By Observation 7.2, $I.oldInfo[1].status$ is not inProgress at all times after that. By Lemma 7.20, no pointer CAS of $I.oldInfo[1]$ succeeds after that. So, $I.nodes[1].nxt = I.nodes[2]$ at all configuration between C_i and C_k, and $I.nodes[1].prv = I.nodes[0]$ at all configurations between C_j and C_k.

Statement 5a Suppose S_k is the first forward CAS of I. First, we show that $I.nodes[0]$, $I.nodes[1]$ and $I.nodes[2]$ are reachable at C_{k-1}. For all m, $I.nodes[m]$ is a Node returned by the call to UpdateCursor that belongs to I. By induc-

tion hypothesis 12, $I.nodes[m]$ was reachable before the call to UpdateCursor returned. By Lemma 7.24, for $m = 0$ and 2, $I.nodes[m].state$ = ordinary at C_{k-1}, so by induction hypothesis 6, $I.nodes[m]$ is reachable at C_{k-1}. By Statement 1, proved earlier, the first forward CAS of I succeeds. So, $I.nodes[0].nxt = I.nodes[1]$ at C_{k-1} and $I.nodes[1]$ is also reachable at C_{k-1} (since $I.nodes[0]$ is reachable at C_{k-1}).

Next, we show $I.nodes[0]$ is reachable at C_k and $I.nodes[1]$ is unreachable at C_k. Since the first forward CAS of I changes $I.nodes[0].nxt$ from $I.nodes[1]$ to $I.newNxt$, $I.nodes[0]$ is still reachable at C_k. By Corollary 7.11, $I.nodes[0]$ is the only reachable Node whose nxt field is $I.nodes[1]$ at C_{k-1}. So, $I.nodes[1]$ is unreachable at C_k.

Next, we show that $I.nodes[2]$ is reachable at C_k. If I is created by a Delete operation, S_k sets $I.nodes[0].nxt$ to $I.nodes[2]$. Since $I.nodes[0]$ is reachable at C_k, $I.nodes[2]$ is also reachable at C_k. Suppose I is created by an InsertBefore operation. Then, $I.newNxt$ and $I.newPrv$ are created at line 30 and 31 that precedes the creation of I respectively. Then, $I.newPrv.nxt$ is set to $I.nodes[2]$ at line 31 and $I.newNxt.nxt$ is set to $I.newPrv$ at line 32. By Statement 3a, proved earlier, $I.newPrv.nxt = I.nodes[2]$ and $I.newNxt.nxt = I.newPrv$ at C_{k-1}. S_k sets $I.nodes[0].nxt$ to $I.newNxt$. Since $I.nodes[0]$ is reachable at C_k, $I.nodes[2]$ is also reachable at C_k.

Statement 5b By Statement 3b, the steps on line 30, 31 or 32 do not make any Node reachable or unreachable. So, only the first forward CAS of an Info object I might make a Node unreachable (by Statement 1). Suppose S_k is the first forward CAS of I. Then, S_k changes $I.nodes[0].nxt$ from $I.nodes[1]$ to $I.newNxt$. By Statement 4, $I.nodes[1].nxt = I.nodes[2]$ at C_{k-1}. By Statement 5a, for all j, $I.nodes[j]$ is reachable at C_{k-1} and $I.nodes[0]$ and $I.nodes[2]$ are still reachable at C_k, but $I.nodes[1]$ is unreachable at C_k.

If I is created by a DELETE operation, S_k changes $I.nodes[0].nxt$ from $I.nodes[1]$ to $I.nodes[2]$. Since $I.nodes[0]$ is reachable at C_k, only $I.nodes[1]$ becomes unreachable at C_k. Suppose I is created by an INSERTBEFORE operation. Then, $I.newNxt$ and $I.newPrv$ are created at line 30 and 31 that precedes the creation of I respectively. Then, $I.newPrv.nxt$ is set to $I.nodes[2]$ at line 31 and $I.newNxt.nxt$ is set to $I.newPrv$ at line 32. By Statement 3a, proved earlier, $I.newNxt.nxt = I.newPrv$ and $I.newPrv.nxt = I.nodes[2]$ at C_{k-1}. Since $I.nodes[0]$ is reachable at C_k and S_k changes $I.nodes[0].nxt$ from $I.nodes[1]$ to $I.newNxt$, only $I.nodes[1]$ becomes unreachable at C_k. Thus, if x becomes unreachable by S_k, $x = I.nodes[1]$.

Statement 6 Suppose a Node x is unreachable at a configuration C_i and was reachable at C_{i-1}. Let C_k be a configuration after C_{i-1}. By Statement 5b, C_i is the configuration that follows the first forward CAS of an Info object I such that

$x = I.nodes[1]$. By the pseudo-code, $x.state$ is set to marked or copied before C_i. By Observation 7.3, $x.state$ is marked or copied at C_k.

Statement 7a Suppose I is a successful Info object in the prefix of an execution up to C_k. Let I' be an Info object different from I. Let C_j be the earlier configuration that follows the return of the call to CheckInfo that belongs to I. We consider two different cases according to what step S_k is.

Case 1: S_k is the (unique) successful flag CAS of I on $I.nodes[2]$. To derive a contradiction, assume that some pointer CAS of I' is performed on $I.nodes[m]$ or changes the nxt or prv field of a Node from $I.nodes[m]$ to another value between C_j and C_k. That CAS must be the first forward or the first backward CAS of I', by Statement 1, proved above. When this CAS occurs, $I.nodes[m].info = I'$, by Lemma 7.21. By Lemma 7.22, $I.nodes[m].info$ is either $I.oldInfo[m]$ or I at all configurations between C_j and C_k. So, $I' = I.oldInfo[m]$.

Since I is created at line 33 or 45, the call to CheckInfo that belongs to I returns true at C_j. So, $I.oldInfo[m].status$ was not inProgress at some prior execution of line 93. By Observation 7.2, $I.oldInfo[m].status$ is not inProgress at all configurations after C_j. By Lemma 7.20, no pointer CAS of $I.oldInfo[m]$ succeeds after C_j. Then, this contradicts our assumption that a pointer CAS of $I' = I.oldInfo[m]$ succeeds after C_j.

Case 2: S_k is not the successful flag CAS of I on $I.nodes[2]$. So, I is a successful Info object in the prefix of the execution up to C_{k-1}. By induction hypothesis 7a, any successful pointer CAS of I' that is performed on $I.nodes[m]$ or changes the nxt or prv field of a Node from $I.nodes[m]$ to other value between C_j and C_{k-1} is preceded by a backward CAS of I. To derive a contradiction assume S_k is a successful pointer CAS of I' that is performed on $I.nodes[m]$ or changes the nxt or prv field of a Node from $I.nodes[m]$ to other value, and no backward CAS of I has occurred before C_{k-1}. By Statement 1, proved earlier, S_k must be the first forward CAS of I' or the first backward CAS of I'. By Lemma 7.21, $I.nodes[m].info = I'$ at C_{k-1}. Since I is successful in the prefix of the execution up to C_{k-1}, by Lemma 7.22, $I.nodes[m].info$ is either $I.oldInfo[m]$ or I at configuration C_{k-1}. Since $I' \neq I$, $I.oldInfo[m] = I'$. Since the call to CheckInfo that belongs to I returns true, $I'.status$ is not inProgress at all configurations after C_j (by Observation 7.2). By Lemma 7.20, $I'.status$ is inProgress at C_{k-1}, which is a contradiction.

Statement 7b Suppose I is a successful Info object in the prefix of an execution up to C_k. Let C_j be the earlier configuration that follows the return of the call to CheckInfo that belongs to I.

Case 1: the first forward CAS of I occurs before C_k. Then, I is successful in the prefix up to C_{k-1} (by Lemma 7.21). By induction hypothesis 7b, $I.nodes[m]$

$(0 \leq m < 3)$ is reachable in all configurations that are after C_j but not after a forward CAS of I.

Case 2: no forward CAS of I occurs before C_k. We show that $I.nodes[m]$ is reachable in all configurations between C_j and C_k $(0 \leq m < 3)$. First, consider $I.nodes[0]$. We first show that $I.nodes[0]$ was reachable before C_j by considering two cases.

Case 2A: $I.nodes[0]$ is created at line 18, 19 or 20, so $I.nodes[0]$ was reachable in C_0.

Case 2B: $I.nodes[0]$ is created at line 30 or 31 that precedes the creation of some Info object I'. Then, $I.nodes[0]$ is $I'.newNxt$ or $I'.newPrv$. We show the first forward CAS of I' occurs before C_j. Prior to the creation of I, UpdateCursor is called at line 25 or 39. At the last execution of line 85 or 88 inside this call, the *node* field of a Cursor is set to $I.nodes[1]$. So, in some earlier configuration, $I.nodes[1]$ is reachable (by induction hypothesis 13). Since $I.nodes[1].prv$ is $I.nodes[0]$ at line 90 preceding the creation of I, $I.nodes[1].prv$ is set to $I.nodes[0]$ before C_j. If $I.nodes[1]$ is also created at line 30 or 31 preceding the creation of I', then the first forward CAS of I' occurs before C_j since $I.nodes[1]$ was reachable before C_j (by induction hypothesis 3b). Otherwise, $I.nodes[1].prv$ is set to $I.nodes[0]$ by a backward CAS before before C_j, but this can only happen after a forward CAS of I' (by induction hypothesis 3b). Since $I.nodes[0]$ is $I'.newNxt$ or $I'.newPrv$, by

induction hypothesis 8, $I.nodes[0]$ was reachable in the configuration after the first forward CAS of I' (before C_j).

Thus, in either case (2A or 2B), $I.nodes[0]$ was reachable before C_j. By Lemma 7.24, $I.nodes[0].state$ is ordinary at all configurations between C_j and C_k. By Statement 6, $I.nodes[0]$ is reachable at all configurations between C_j and C_k.

Next, consider $I.nodes[1]$. Since, at the last execution of line 82 preceding the creation of I, $c.node$ is $I.nodes[1]$, in some earlier configuration, $I.nodes[1]$ was reachable (by induction hypothesis 13). By Lemma 7.23, $I.nodes[1].state$ is ordinary when $I.nodes[1].prv$ is read on line 90. Since $I.nodes[1]$ was reachable before the last execution of line 82, $I.nodes[1]$ is reachable when $I.nodes[1].prv$ is read on line 90 (by induction hypothesis 6). Since $c.node$ is $I.nodes[1]$ on the last execution of line 82, $I.nodes[1] \neq Head$ (by Invariant 7.6.10). Since $I.nodes[1].prv = I.nodes[0]$ on line 90, $I.nodes[0].nxt = I.nodes[1]$ on line 90 (by induction hypothesis 10). If $I.nodes[0].nxt$ is changed from $I.nodes[1]$ to another value, $I.nodes[1].state$ is not ordinary after that (by Observation 7.3). By Lemma 7.23, $I.nodes[1].state$ is ordinary in C_j. So, $I.nodes[0].nxt = I.nodes[1]$ in C_j. By Statement 7a, $I.nodes[0].nxt = I.nodes[1]$ at all configurations between C_j and C_k. Since $I.nodes[0]$ is reachable between C_j and C_k, $I.nodes[1]$ is also reachable between C_j and C_k.

Now, consider $I.nodes[2]$. $I.nodes[2]$ is read from $I.nodes[1].nxt$ at line 90.

If $I.nodes[1].nxt$ is changed from $I.nodes[2]$ to another value, $I.nodes[2].state$ is not ordinary after that (by Observation 7.3). Since $I.nodes[2].state$ is ordinary in C_j (by Lemma 7.24), $I.nodes[1].nxt$ is still $I.nodes[2]$ in C_j. By Statement 7a, $I.nodes[1].nxt = I.nodes[2]$ at all configurations between C_j and C_k. Since $I.nodes[1]$ is reachable between C_j and C_k, $I.nodes[2]$ is also reachable between C_j and C_k.

Statement 7c Suppose I is a successful Info object in the prefix of an execution up to C_k. If I is a successful Info object in the prefix of the execution up to C_{k-1}, then the claim follows from induction hypothesis 7c. Otherwise, S_k is the successful flag CAS of I on $I.nodes[2]$. Let C_j be the earlier configuration after the return of the call to CheckInfo that belongs to I. We show Statement 7c holds at C_j.

By Statement 4, $I.nodes[1].prv = I.nodes[0]$ and $I.nodes[1].nxt = I.nodes[2]$ at C_j. Since I is successful in the prefix of the execution up to C_k, by Statement 7b, $I.nodes[1]$ is reachable at C_j. Since $I.nodes[1].prv = I.nodes[0]$ at C_j, $I.nodes[1] \neq Head$ (by Invariant 7.8 and Observation 7.10). By induction hypothesis 10, $I.nodes[1].prv.nxt = I.nodes[1]$ at C_j, so $I.nodes[0].nxt = I.nodes[1]$ at C_j.

It remains to prove that $I.nodes[2].prv = I.nodes[1]$ at C_j. We wish to apply induction hypothesis 9 to $I.nodes[1]$ at configuration C_j. Suppose C_j is after the first forward CAS of I' but not after the first backward CAS of I' for some Info

object $I' \neq I$. We shall show that $I'.newPrv \neq I.nodes[1]$, so that induction hypothesis 9 can be applied to $I.nodes[1]$ at C_j.

Since the first forward CAS of I' is executed before C_j, I' is a successful Info object (by Lemma 7.21). By Lemma 7.17, $I'.status$ is never set to aborted. Since there is no backward CAS of I' before C_j, no invocation of HELP(I') sets $I'.status$ to committed on line 116 before C_j. So, $I'.status$ is inProgress at C_j. Let m be 0, 1 or 2. Since $I.oldInfo[m].status$ is not inProgress at some time during the call to CHECKINFO that belongs to I, $I.oldInfo[m].status$ is not inProgress at C_j (by Observation 7.2). Since $I'.status$ is inProgress at C_j, $I' \neq I.oldInfo[m]$ for any $m \in \{0, 1, 2\}$.

Next, we show that, for each $l \in \{0, 1, 2\}$, $I'.nodes[l].info = I'$ at C_j. By Lemma 7.22, $I'.nodes[l].info$ is either $I'.oldInfo[l]$ or I' at C_j. Since $I'.nodes[l].info$ is changed from $I'.oldInfo[l]$ to I' before the first forward CAS of I' (by Lemma 7.21), which is before C_j and $I'.nodes[l].info$ is not set to $I'.oldInfo[l]$ after that (by Lemma 7.13), $I'.nodes[l].info = I'$ at C_j. We now consider two cases.

If I' is created by a DELETE operation, $I'.newPrv = I'.nodes[0]$. We have shown that $I'.nodes[0].info = I'$ at C_j. By Lemma 7.22, $I.nodes[1].info$ is either I or $I.oldInfo[1]$ at C_j. Since I is created after C_j, $I.nodes[1].info = I.oldInfo[1]$ at C_j. Since $I'.nodes[0].info = I'$ at C_j and $I.nodes[1].info = I.oldInfo[1]$ at C_j and $I.oldInfo[1] \neq I'$, $I.nodes[1] \neq I'.nodes[0] = I'.newPrv$.

If I' is created by an InsertBefore operation, $I'.newPrv.nxt = I'.nodes[2]$ at C_j (by induction hypothesis 3a). To derive a contradiction, suppose $I'.newPrv = I.nodes[1]$. Since $I.nodes[1].nxt = I.nodes[2]$ at C_j, $I'.nodes[2] = I.nodes[2]$. By Lemma 7.22, $I.nodes[2].info$ is either $I.oldInfo[2]$ or I at C_j. Since I is created after C_j, $I.nodes[2].info = I.oldInfo[2]$ at C_j. We have shown $I'.nodes[2].info = I'$ at C_j. This contradicts the fact that $I.oldInfo[2] \neq I'$ since $I'.nodes[2] = I.nodes[2]$. So, $I'.newPrv \neq I.nodes[1]$.

Thus, in either case, we have $I'.newPrv \neq I.nodes[1]$. Since I is successful in the prefix of the execution up to C_k, by Statement 7b, $I.nodes[1]$ is reachable at C_j. Since $I.nodes[1].nxt = I.nodes[2]$ at C_j, $I.nodes[1] \neq Tail$ (by Invariant 7.8 and Observation 7.10). By induction hypothesis 9, $I.nodes[1].nxt.prv = I.nodes[1]$ at C_j. Since $I.nodes[1].nxt = I.nodes[2]$ at C_j, $I.nodes[2].prv = I.nodes[1]$ at C_j.

Statement 8 Suppose I is an Info object that is created by an InsertBefore operation and S_k is the first forward CAS of I. We show exactly $I.newNxt$ and $I.newPrv$ become reachable at C_k. By Statement 1, S_k succeeds. So, $I.nodes[0].nxt = I.newNxt$ at C_k. By Statement 3a, $I.newNxt.nxt = I.newPrv$ and $I.newPrv.nxt = I.nodes[2]$ at C_{k-1}. Since $I.nodes[0]$ and $I.nodes[2]$ are reachable at C_k (by Statement 5a), exactly $I.newNxt$ and $I.newPrv$ become reachable at C_k.

Statement 9 If S_k is an execution of line 30, 31 or 32, then S_k preserves the truth of the statement since Nodes that are created at line 30 or 31 are unreachable at C_k (by Statement 3b, proved earlier). By Statement 1, the only other step that could cause Statement 9 to become false is the first forward or first backward CAS of an Info object I.

We consider two cases according to what operation created I. For each case, we first show the statement is true if S_k is the first forward CAS of I, and then we show the statement is true if S_k is the first backward CAS of I. Throughout the following cases, let Node x be any Node that satisfies conditions (1), (2) and (3) of Statement 9 in configuration C_k. We shall show that $x.nxt.prv = x$ in C_k.

Case 1: I is created by an InsertBefore operation. Then, $I.newNxt$ and $I.newPrv$ are created at line 30 and 31 preceding the creation of I.

Case 1A: S_k is the first forward CAS of I. The first forward CAS of I changes $I.nodes[0].nxt$ from $I.nodes[1]$ to $I.newNxt$.

C_k is after the first forward CAS of I but not after the first backward CAS of I, so $x \neq I.newPrv$ since x satisfies Condition (3) in C_k.

If $x = I.newNxt$, then by Statement 3a, $x.nxt = I.newPrv$ and $I.newPrv.prv = I.newNxt = x$ in C_k.

If $x = I.nodes[0]$, then $x.nxt = I.newNxt$ in C_k. By Statement 3a, $I.newNxt.prv = I.nodes[0] = x$ in C_k.

Otherwise, x is reachable at C_k and x is not $I.newPrv$, $I.newNxt$ or $I.nodes[0]$. By Statement 8, only $I.newNxt$ and $I.newPrv$ become reachable by S_k. Since x is not $I.newPrv$ or $I.newNxt$, x was reachable at C_{k-1}. Moreover, since x satisfies Condition (3) in C_k, it also does in C_{k-1}. Since S_k does not change $x.nxt$ or any Node's prv field, $x.nxt.prv$ is still x (by induction hypothesis 9).

Case 1B: S_k is the first backward CAS of I. The first backward CAS of I changes $I.nodes[2].prv$ from $I.nodes[1]$ to $I.newPrv$.

If $x = I.newPrv$, then by Statement 3a, $x.nxt = I.nodes[2]$ at C_k. Since $I.nodes[2].prv = x$ at C_k, $x.nxt.prv = I.newPrv = x$ at C_k.

Otherwise, x is reachable at C_k and x is not $I.newPrv$. By induction hypothesis 5a, $I.nodes[1]$ is unreachable at the configuration after the first forward CAS of I. Only a forward CAS can make a Node reachable because other instructions that change or initialize the nxt field of a Node only occur when the Node is unreachable (by Statement 3b). By Statement 1, only the first forward CAS of any Info object I'' succeeds. Next, we show that $I.nodes[1]$ does not become reachable again as a result of the first forward CAS of I'' (between the first forward CAS of I and C_k).

If I'' is created by a DELETE operation, since the first forward CAS of I'' sets $I''.nodes[0].nxt$ to $I''.nodes[2]$ and $I''.nodes[2]$ is reachable at the configuration before the first forward CAS of I'' (by induction hypothesis 5a), no Node becomes reachable by the first forward CAS of I''. Suppose I'' is created by an INSERT-

BEFORE operation. By induction hypothesis 8, only $I''.newNxt$ and $I''.newPrv$ become reachable by the first forward CAS of I''. Since $I.nodes[1]$ was reachable at the configuration before the first forward CAS of I earlier, $I.nodes[1]$ is not $I''.newNxt$ or $I''.newPrv$ (by induction hypothesis 3b).

So, $I.nodes[1]$ is unreachable at C_k and hence $x \neq I.nodes[1]$. Let x' be $x.nxt$ at C_{k-1}. Since no Node becomes reachable as a result of S_k, x is also reachable at C_{k-1}. Since x satisfies Condition (3) in C_k and $x \neq I.newPrv$, it also satisfies Condition (3) in C_{k-1}. Thus, by induction hypothesis 9, at C_{k-1}, $x.nxt.prv = x$. So, $x'.prv = x$ at C_{k-1}. Since S_k changes $I.nodes[2].prv$ from $I.nodes[1]$ to another value and $x \neq I.nodes[1]$, $x' \neq I.nodes[2]$. Since S_k does not change $x.nxt$ or $x'.prv$, $x.nxt.prv$ is still x at C_k.

Case 2: I is created by a DELETE operation. Then, $I.newNxt = I.nodes[2]$ and $I.newPrv = I.nodes[0]$.

Case 2A: S_k is the first forward CAS of I. The first forward CAS of I changes $I.nodes[0].nxt$ from $I.nodes[1]$ to $I.nodes[2]$.

C_k is after the first forward CAS of I but not after the first backward CAS of I, so $x \neq I.newPrv = I.nodes[0]$ since x satisfies Condition (3) in C_k. So, x is reachable at C_k and x is not $I.nodes[0]$. By induction hypothesis 7c and Statement 7a, at C_{k-1}, $I.nodes[0].nxt = I.nodes[1]$ and $I.nodes[1].nxt = I.nodes[2]$. Since S_k changes $I.nodes[0].nxt$ from $I.nodes[1]$ to $I.nodes[2]$, no Node becomes reachable

as a result of S_k. So, x was reachable at C_{k-1}. Moreover, since x satisfies Condition (3) in C_k, it also does in C_{k-1}. Since S_k does not change $x.nxt$ or any Node's prv field, $x.nxt.prv$ is still x (by induction hypothesis 9).

Case 2B: S_k is the first backward CAS of I. The first backward CAS of I changes $I.nodes[2].prv$ from $I.nodes[1]$ to $I.nodes[0]$.

If $x = I.newPrv = I.nodes[0]$, $x.nxt$ is set to $I.nodes[2]$ by the first forward CAS of I. By Statement 7a, proved earlier, $x.nxt$ is still $I.nodes[2]$ at C_k. Since $I.nodes[2].prv = x$ at C_k (by Statement 1), $x.nxt.prv = x$ at C_k.

Otherwise, x is reachable at C_k and x is not $I.newPrv$. By the same argument as in Case 1B, Statement 9 is true for x in C_k.

Statement 10 Let $x \neq Head$ be any reachable Node in C_k. Then, there exists a Node x' such that, in C_k, x' is reachable and $x'.nxt = x$. By Invariant 7.8 and Observation 7.10, $Tail.nxt$ is null, so x' is not $Tail$. If $x.prv = x'$ in C_k, then $x.prv.nxt = x'.nxt = x$ in C_k, as required.

Suppose $x.prv \neq x'$ in C_k. Then, $x'.nxt.prv \neq x'$ in C_k. By Statement 9, proved earlier, there is an Info object I such that $x' = I.newPrv$ and C_k is after the first forward CAS of I, but not after the first backward CAS of I. We first show that $x = I.nodes[2]$ by considering two cases.

If I is created by an InsertBefore operation, $x = x'.nxt = I.newPrv.nxt =$

$I.nodes[2]$ in C_k by Statement 3a, proved earlier.

If I is created by a DELETE operation, the first forward CAS of I sets $I.nodes[0].nxt$ to $I.newNxt$ (by Statement 1). So, we have $x = x'.nxt = I.newPrv.nxt = I.nodes[0].nxt = I.nodes[2]$ in C_k by Statement 7a, proved earlier.

Thus, in either case, $x = I.nodes[2]$. Since the first forward CAS of I occurs before C_k, I is successful in the prefix of the execution up to C_k (by Lemma 7.21). By induction hypothesis 7c, $I.nodes[2].prv = I.nodes[1]$ and $I.nodes[1].nxt = I.nodes[2]$ at the configuration after the return of the call to CHECKINFO that belongs to I. By Statement 7a, at C_k, $I.nodes[2].prv = I.nodes[1]$ and $I.nodes[1].nxt = I.nodes[2]$ (since C_k is after the first forward CAS of I, but not after the first backward CAS of I). So, in C_k, $x.prv.nxt = I.nodes[2].prv.nxt = I.nodes[1].nxt = I.nodes[2] = x$.

Statement 11 Suppose I is a successful Info object in the prefix of an execution up to C_k. If no forward CAS of I occurs before C_k, the statement is trivially true. Otherwise, a forward CAS of I occurs before C_k, so I is successful in the prefix of the execution up to C_{k-1}, by Lemma 7.21. By induction hypothesis 11, no pointer CAS of an Info object $I' \neq I$ changes the nxt or prv field of $I.nodes[1]$ after the first forward CAS of I and before C_{k-1}. Finally, to derive a contradiction, suppose S_k is a pointer CAS of an Info object I' that changes the nxt or prv field

of $I.nodes[1]$. By Statement 1, proved earlier, S_k is the first forward CAS of I' or the first backward CAS of I'. By Lemma 7.21, $I.nodes[1].info = I'$ at C_{k-1}. By Lemma 7.25, $I.nodes[1].info = I$ at all configurations after the first forward CAS of I. So, $I.nodes[1].info$ must be I at C_{k-1}. But, $I \neq I'$ which is a contradiction.

Statement 12 Induction hypothesis 12 shows that the claim is true for calls to UpdateCursor that return before C_{k-1}. Now, suppose S_k is the return, at line 90, of a call to UpdateCursor. Let $\langle node, \text{-}, nxtNode, prvNode, \text{-}, \text{-}\rangle$ be the result of the call to UpdateCursor. Since the *node* field of a Cursor is *node* on the last execution of line 82 before C_{k-1}, *node* was reachable at some configuration before that line (by induction hypothesis 13). Let x be $prvNode$ or $nxtNode$. If x is $Head$, $Tail$ or $EOLnode$ created at line 18, 19 or 20, x was reachable at C_0.

Otherwise, x is created at line 30 or 31 preceding the creation of an Info object I. If *node* was also created on line 30 or 31 preceding the creation of I, since *node* is reachable at an earlier configuration, the first forward CAS of I occurs before line 90 (by induction hypothesis 3b). Otherwise, since the *nxt* or *prv* field of *node* is equal to x on line 90, the *nxt* or *prv* field of *node* is set to x before reading x on that line. Since *node* was reachable earlier, the first forward CAS of I occurs before line 90 (by induction hypothesis 3b). By induction hypothesis 8, x was reachable at the configuration after the first forward CAS of I (before reading x on line 90).

Statement 13 We need only consider cases when S_k sets the *node* field of a Cursor. Suppose c is a Cursor and S_k sets $c.node$ to x. Then, we show x was reachable at some configuration before C_k. We consider different cases according to what line sets $c.node$ to x at S_k.

Case 1: S_k is an execution of line 35 after the creation of an Info object I at line 33. Then, $c.node$ is set to $I.newPrv$ by S_k after the call to HELP(I) returns true at line 34. So, $I.status$ is committed at line 119 of HELP(I). Since $I.status$ is initially inProgress (by Lemma 7.5), $I.status$ is set to committed at line 116 of some call to HELP(I). By the pseudo-code, the first forward CAS of I is executed earlier. By induction hypothesis 8, $I.newPrv$ is reachable at the configuration that follows the first forward CAS of I (before C_{k-1}).

Case 2: S_k is an execution of line 47. Let $\langle node$, -, $nxtNode$, -, -, - $\rangle$ be the result of the preceding call to UPDATECURSOR in line 39. Then, $c.node$ is set to $x = nxtNode$ by S_k. By induction hypothesis 12, $nxtNode$ was reachable at some configuration before C_{k-1}.

Case 3: S_k is an execution of line 54. Let $\langle node$, -, -, $prvNode$, -, -$\rangle$ be the result of the preceding call to UPDATECURSOR in line 50. By induction hypothesis 12, $prvNode$ was reachable at some configuration before the call to UPDATECURSOR returns. Since the MOVELEFT does not return at line 52, $prvNode \neq Head$. Then, by induction hypothesis 14, $prvNode.prv.nxt \neq prvNode$ at the configuration after

line 53 (which is before C_{k-1}). By induction hypothesis 15, for some Info object I, $prvNode = I.nodes[1]$ and the first forward CAS of I occurs earlier than C_{k-1}. Since $prvNode.state$ is copied at line 54, I is created by an InsertBefore operation (by Lemma 7.23). So, $prvNode.copy$ is set to $I.newPrv$ at line 112 before setting $prvNode.state$ inside Help(I). By Lemma 7.26, $prvNode.copy = I.newPrv$ at C_k. Then, $c.node$ is set to $x = I.newPrv$ by S_k. By induction hypothesis 8, $I.newPrv$ is reachable at the configuration after the first forward CAS of I (before C_{k-1}).

Case 4: S_k is an execution of line 58. Let $\langle node$, -, -, $prvNode$, -, - $\rangle$ be the result of the preceding call to UpdateCursor in line 50. By induction hypothesis 12, $prvNode$ was reachable at some configuration before the call to UpdateCursor returns. Since the MoveLeft does not return at line 52, $prvNode \neq Head$. Then, by induction hypothesis 14, $prvNode.prv.nxt \neq prvNode$ at the configuration after line 53 (which is before C_{k-1}). By induction hypothesis 15, for some Info object I, $prvNode = I.nodes[1]$ and the first forward CAS of I occurs earlier than C_{k-1}. Let C_j be the configuration after the first forward CAS of I (before C_{k-1}). By Lemma 7.21, I is successful in the prefix of the execution up to C_{k-1}. By induction hypothesis 7a and 7c, $I.nodes[1].prv = I.nodes[0]$ at C_j. By induction hypothesis 11, $I.nodes[1].prv = I.nodes[0]$ at line 56 before C_{k-1}. So, $prvPrvNode$ is set to $I.nodes[0]$ at line 56 before C_{k-1}. By induction hypothesis 5a, $x = I.nodes[0]$ was reachable at C_j (before C_{k-1}).

Case 5: S_k is an execution of line 59 or 65 that sets $c.node$ to x. Since x was returned by the preceding call to UpdateCursor on line 50 or 62 (before C_{k-1}), by induction hypothesis 12, x was reachable before C_{k-1}.

Case 6: S_k is an execution of line 68 or 73. Since x was $Head.nxt$ at configuration C_{k-1}, x was reachable at that configuration (by definition of reachability).

Case 7: S_k is an execution of line 85 or 88 inside a call to UpdateCursor. Let y be the value of $c.node$ in configuration C_{k-1}. By induction hypothesis 13, y was reachable at some configuration before the last execution of line 82. By Invariant 7.6.10, $y \neq Head$. Then, $y.prv.nxt \neq y$ at the configuration after line 82 before C_k (by induction hypothesis 14). By induction hypothesis 15, for some Info object I, $I.nodes[1] = y$ and the first forward CAS of I occurs earlier than C_k. Let C_j be the configuration that follows the first forward CAS of I ($j < k$).

If $y.state$ is copied at line 83, $c.node$ is set to x on line 85. Then, $y.state$ must have been set to copied at line 113 inside Help(I) (by Lemma 7.23). So, $y.copy$ is set to $I.newPrv$ at line 112 earlier. By Lemma 7.26, $y.copy = I.newPrv$ at line 85. Since $y.state$ is set to copied inside Help(I), I is created by an InsertBefore operation. By induction hypothesis 8, $x = I.newPrv$ is reachable at C_j.

If $y.state$ is marked at line 86, $c.node$ is set to x on line 88. Then, $y.state$ must have been set to marked inside Help(I) (by Lemma 7.23) and I is created by a Delete operation. Let $\langle node$, -, $nxtNode$, -, -, -$\rangle$ be the result of the call

to UpdateCursor that belongs to I. By induction hypothesis 12, $nxtNode$ was reachable at some configuration before C_k. Now, we show $x = nxtNode$. By Lemma 7.21, I is successful in the prefix of the execution up to C_{k-1}. By induction hypothesis 7a and 7c, $I.nodes[1].nxt = I.nodes[2]$ at C_j. By induction hypothesis 11, $y.nxt = I.nodes[1].nxt = I.nodes[2]$ at line 88. Since x is read from $y.nxt$ at line 88 and $nxtNode = I.nodes[2]$, $x = nxtNode$ which is reachable before C_k.

Statement 14 Assume three configurations $C_{i_1}, C_{i_2}, C_{i_3}$ satisfying the conditions exist before C_k. We first show that there is a forward CAS of an Info object I before C_{i_3} such that $x = I.nodes[1]$. To derive a contradiction, assume there is no forward CAS of an Info object I before C_{i_3} such that $x = I.nodes[1]$. So, x is still reachable at C_{i_2} by induction hypothesis 5b. By Statement 10, $x.prv.nxt = x$ at C_{i_2}, so $y.nxt = x$ at C_{i_2}. Since $y.nxt \neq x$ at C_{i_3}, a forward CAS of an Info object I' has changed $y.nxt$ from x to another value between C_{i_2} and C_{i_3}. So, $I'.nodes[1] = x$, which is a contradiction. So, there is a forward CAS of I before C_{i_3} such that $x = I.nodes[1]$.

By Statement 7a and 7c, proved earlier, $I.nodes[1].prv = I.nodes[0]$ at the configuration before the first forward CAS of I. By Statement 11, proved earlier, $I.nodes[1].prv = I.nodes[0]$ at all configurations after the first forward CAS of I. So, $I.nodes[1].prv = I.nodes[0]$ at C_{i_2}. Since $x = I.nodes[1]$, $y = I.nodes[0]$ and

$x.prv = y$ at C_k.

Since there is a forward CAS of I before C_{i_3}, $y.nxt$ is changed from x to another value before C_{i_3} (by Statement 1). By Statement 2, proved earlier, $y.nxt \neq x$ at C_k.

Statement 15 We first prove the "only if" direction. Suppose $x \neq Head$ was reachable in some configuration earlier than C_k and $x.prv.nxt \neq x$ in C_k. By Statement 10, x is not reachable at C_k. Since x was reachable in some configuration earlier than C_k, x became unreachable before C_k by the first forward CAS of an Info object I such that $I.nodes[1] = x$ (by Statement 5b).

Next, we prove the "if" direction. Suppose there is an Info object I such that $x = I.nodes[1]$ and C_k is a configuration after the first forward CAS of I. By Lemma 7.5, I is created at line 33 or 45. So, the *node* field of a Cursor was $I.nodes[1]$ at the last execution of line 82 inside the call to UpdateCursor that belongs to I. By Lemma 7.6.10, $x = I.nodes[1] \neq Head$.

By induction hypothesis 5a, $x = I.nodes[1]$ was reachable at the configuration before the first forward CAS of I (before C_k). By Lemma 7.21, I is successful in the prefix of the execution up to C_{k-1}. By induction hypothesis 7a and 7c, $I.nodes[0].nxt = I.nodes[1]$ and $I.nodes[1].prv = I.nodes[0]$ at the configuration before the first forward CAS of I. Since $I.nodes[0].nxt$ is changed from $I.nodes[1]$ to another value by the first forward CAS of I (by Statement 1), $I.nodes[0].nxt \neq$

$I.nodes[1]$ at all configurations after the first forward CAS of I (by Statement 2). By Statement 11, proved earlier, $I.nodes[1].prv = I.nodes[0]$ at all configurations after the first forward CAS of I. So, $I.nodes[1].prv.nxt = I.nodes[0].nxt \neq I.nodes[1]$ at C_k. □

In the rest of the subsection, we show some facts regarding reachability and we show what the test on line 53 and 82 tell about the reachability of a Node.

Lemma 7.29. *In any configuration C, $Tail$ is reachable.*

Proof. In C_0, $Tail$ is reachable. Let I be an Info object. Since $c.node = I.nodes[1]$ at the last execution of line 82 inside the call to UpdateCursor that belongs to I, $I.nodes[1] \neq Tail$ (by Invariant 7.6.10). By Lemma 7.28.5b, $Tail$ is reachable at any configuration. □

Lemma 7.30. *Let I be a successful Info object and let C be a configuration that is after a forward CAS of I and not after a backward CAS of I. Then, $I.nodes[0]$ and $I.nodes[2]$ are reachable at C.*

Proof. Let i be 0 or 2. By Lemma 7.28.5a, $I.nodes[i]$ is reachable at the configuration after the first forward CAS of I. By Lemma 7.24, $I.nodes[i].state =$ ordinary at C. By Lemma 7.28.6, $I.nodes[i]$ is reachable at C. □

Lemma 7.31. *Let I be a successful Info object that is created by an* InsertBefore *operation and let C be a configuration after a forward CAS of I and not after a backward CAS of I. Then, $I.newNxt$ and $I.newPrv$ are reachable at C.*

Proof. By Lemma 7.28.1, $I.nodes[0].nxt = I.newNxt$ at the configuration after the first forward CAS of I. By Lemma 7.28.7a, $I.nodes[0].nxt$ is equal to $I.newNxt$ at C. By Lemma 7.28.3a, $I.newNxt.nxt = I.newPrv$ at C. By Lemma 7.30, $I.nodes[0]$ is reachable at C. So, $I.newNxt$ and $I.newPrv$ are reachable at C. □

In the implementation, the expected Node that is used for a successful forward CAS is removed from the list and it is never added to the list after that.

Lemma 7.32. *If x is reachable at C_{i-1} and unreachable at C_i, then x is unreachable at all configurations after C_{i-1}.*

Proof. Only a successful forward CAS can make a Node reachable. Now, we show a Node that is made reachable by a forward CAS has never been reachable before. Let C_k be the configuration after a successful forward CAS of an Info object I. By Lemma 7.28.1, it is the first forward CAS of I. We consider two cases according to what operation created I.

Case 1: I is created by a Delete operation. Then, $I.nodes[0].nxt = I.nodes[2]$ at C_k (by Lemma 7.28.1). Since $I.nodes[2]$ was reachable at C_{k-1} (by Lemma 7.28.5a), no Node becomes reachable as a result of the forward CAS.

Case 2: I is created by an InsertBefore operation. By Lemma 7.28.3b, $I.newNxt$ and $I.newPrv$ are unreachable at all configurations before C_k. By Lemma 7.28.8, the only Nodes that become reachable at C_k are $I.newNxt$ and $I.newPrv$. □

In the implementation, we test whether a Node x has become unreachable by testing whether $x.prv.nxt = x$. The following lemma justifies this test.

Lemma 7.33. *Let $x \neq Head$ be a Node that was reachable in some configuration before C_i. Then, x is reachable at C_i if and only if $x.prv.nxt = x$ at C_i.*

Proof. First, we show if $x.prv.nxt \neq x$ at C_i, x is unreachable at C_i. For some Info object I, $I.nodes[1] = x$ and the first forward CAS of I occurs before C_i (by Lemma 7.28.15). So, $I.nodes[1]$ is unreachable at the configuration after the first forward CAS of I (by Lemma 7.28.5a). By Lemma 7.32, $x = I.nodes[1]$ is still unreachable at C_i (which is after the first forward CAS of I).

Now, we show if $x.prv.nxt = x$ at C_i, then x is reachable at C_i. By the hypothesis, $x \neq Head$ was reachable in some configuration before C_i. By Lemma 7.28.5b, x becomes unreachable after that only by a forward CAS of an Info object I such that $I.nodes[1] = x$. Since $x.prv.nxt = x$ at C_i, the first forward CAS of I does not occur before C_i (by Lemma 7.28.15). Thus, x is reachable at C_i. □

Testing whether $x.prv.nxt = x$ requires two shared-memory accesses, so it can-

not be done atomically. The following lemmas describe how this test behaves.

Lemma 7.34. *Let $x \neq Head$ be a Node. Suppose x was reachable at some configuration C and the test ($x.prv.nxt = x$) is carried out after C but before configuration C'. If, in C', $x.prv.nxt = x$, then the test evaluates to true.*

Proof. We prove the contrapositive. Suppose the test evaluates to false. Then, there are three configurations C_{i_1}, C_{i_2}, C_{i_3} (with $i_1 < i_2 < i_3 \leq k$) and a Node y such that x is reachable at C_{i_1}, $x.prv = y$ in C_{i_2} and $y.nxt \neq x$ in C_{i_3}. By Lemma 7.28.14, in C' (which is after C_{i_3}), $x.prv.nxt \neq x$. □

Lemma 7.35. *Let $x \neq Head$ be a Node. Suppose x was reachable at some configuration C. If, in configuration C' after C, $x.prv.nxt \neq x$ and the test ($x.prv.nxt = x$) is carried out after C', then the test evaluates to false.*

Proof. Suppose that in C', $x.prv.nxt \neq x$. By Lemma 7.28.15, there is an Info object I such that $x = I.nodes[1]$ and the first forward CAS of I occurred before C'.

Let y be the Node the test reads from $x.prv$ (after C'). By Lemma 7.28.7a and 7.28.7c, $I.nodes[1].prv = I.nodes[0]$ at the configuration before the first forward CAS of I. By Lemma 7.28.11, $I.nodes[1].prv = I.nodes[0]$ at all configurations after the first forward CAS of I. Since $x.prv = y$ after C' and $x = I.nodes[1]$, $y = I.nodes[0]$. Since the first forward CAS of I changes $y.nxt$ from x to another value (by Lemma 7.28.1), $y.nxt \neq x$ at all configurations after the first forward

CAS of I (by Lemma 7.28.2). So, $y.nxt \neq x$ when the test reads $y.nxt$. Thus, the test evaluates to false. □

By Lemma 7.33, 7.34 and 7.35, we have the following corollary.

Corollary 7.36. *Let $x \neq Head$ be a Node. Suppose the test $(x.prv.nxt \neq x)$ is carried out and x was reachable at some configuration before the test.*

If the test evaluates to false, x is reachable at the configuration before the test. Otherwise, x is not reachable at the configuration after the test.

Lemma 7.37. *Consider a successful forward CAS of an Info object I. Then, $I.nodes[1]$ is reachable at the configuration before the last execution of line 82 inside the call to* UpdateCursor*(c) that belongs to I.*

Proof. Let C_j be the configuration before the last execution of line 82 inside the call to UpdateCursor that belongs to I. Since $c.node = I.nodes[1]$ at C_j, $I.nodes[1] \neq Head$ (by Invariant 7.6.10) and $I.nodes[1]$ was reachable at some configuration before C_j (by Lemma 7.28.13). Since the test $(I.nodes[1].prv.nxt \neq I.nodes[1])$ on line 82 evaluates to false, $I.nodes[1]$ is reachable at C_j (by Corollary 7.36). □

7.4 Linearizability

In this section, we show the implementation is linearizable. First, we have several lemmas that will be used to show that the MoveLeft or MoveRight operation is linearizable if it sets the *node* field of its Cursor or returns false. Since an update operation that succeeds to change the list will be linearized when its forward CAS changes the *nxt* field of a Node (but before its backward CAS changes the *prv* field of another Node), the *prv* field of Nodes might not be up to date in all configurations. Thus, assigning a linearization point to a MoveLeft operation is trickier, compared to assigning a linearization point to a MoveRight. To linearize MoveLeft, we require the following lemmas.

Lemma 7.38. *Consider an execution of line 53 where the test evaluates to true. Let* $\langle node, -, -, prvNode, -, - \rangle$ *be the result of the preceding call to* UpdateCursor *on line 50. Then, prvNode becomes unreachable before the end of line 53 by a forward CAS of an Info object I such that* $I.nodes[1] = prvNode$ *and* $I.nodes[2] = node$ *and no backward CAS of I occurs before the call to* UpdateCursor *on line 50 reads c.node.prv on line 90.*

Proof. Let C be the configuration after reading the *prv* and *nxt* field on line 53. By Lemma 7.28.12, $prvNode$ is reachable at some configuration before C. Since the MoveLeft does not return on line 52, $prvNode \neq Head$. Since

$prvNode.prv.nxt \neq prvNode$ on line 53, by Lemma 7.28.14, $prvNode.prv.nxt \neq prvNode$ at C. By Lemma 7.33, $prvNode$ is not reachable at C. By Lemma 7.28.5b, $prvNode$ becomes unreachable by the first forward CAS of an Info object I before C such that $I.nodes[1] = prvNode$.

Next, we show that there is no backward CAS of I before line 90 inside the call to UpdateCursor returns on line 50. Let s be the first forward CAS of I. By Lemma 7.28.7a and 7.28.7c, $I.nodes[1].nxt = I.nodes[2]$ at the configuration after s. By Lemma 7.28.11, $I.nodes[1].nxt = I.nodes[2]$ at all configurations after s. Since $prvNode.nxt = node$ at line 53 (after s) and $I.nodes[1] = prvNode$, $I.nodes[2] = node$. By Lemma 7.28.1, $node.prv = prvNode$ just before the first backward CAS of I. By Lemma 7.28.2, $node.prv \neq prvNode$ at all configurations after the first backward CAS of I. Since $node.prv = prvNode$ on line 90, no backward CAS of I occurs before that. □

Lemma 7.39. *If* MoveLeft*(c) changes c.node from y to x at line 54, 58 or 59, then there is a configuration during the* MoveLeft *between the last execution of line 82 inside the call to* UpdateCursor *on line 50 and setting c.node on line 54, 58 or 59 in which*

- *x is reachable, and*
- $x.nxt = y$.

Proof. When the call to UpdateCursor on line 50 executes line 90, $c.node = y$ because there is no change to $c.node$ between line 50 and line 54, 58 or 59. So, the first component returned by updateCursor is y. Let $\langle y, \text{-}, \text{-}, prvNode, \text{-}, \text{-}\rangle$ be the result of the call to UpdateCursor. We consider different cases according to what line sets $c.node$.

Case 1: MoveLeft(c) sets $c.node$ on line 54 or 58. Then, the condition at line 53 is true. By Lemma 7.38, $prvNode$ becomes unreachable before line 53 completes by a forward CAS of an Info object I such that $I.nodes[1] = prvNode$ and $I.nodes[2] = y$ and no backward CAS of I occurs before the call to UpdateCursor on line 50 reads $c.node.prv$ on line 90. Thus, there is a configuration C_z between line 90 inside the call to UpdateCursor and the end of line 53 during MoveLeft that is after the first forward CAS of I but not after the first backward CAS of I.

Now, we consider two cases according to what operation creates I. For each case, we show the lemma is true at C_z.

Case 1A: I is created by an InsertBefore operation. By the pseudo-code, $prvNode.state$ is set to copied on line 113 before the first forward CAS of I (since $I.nodes[1] = prvNode$). Since the first forward CAS of I occurred before the end of line 53, $prvNode.state$ is also set to copied before the end of line 53. By Lemma 7.23, $prvNode.state$ is copied at line 54. So, $c.node$ is set at line 54. Now, we show $c.node$ is set to $I.newPrv$ at line 54. By the pseudo-code, $I.nodes[1].copy$

is set to $I.newPrv$ on line 112 before the first forward CAS of I. By Lemma 7.26, $I.nodes[1].copy = I.newPrv$ at line 54. Since $prvNode = I.nodes[1]$, $c.node$ is set to $I.newPrv$ at line 54, so $x = I.newPrv$. By Lemma 7.28.3a, $x.nxt = I.newPrv.nxt = I.nodes[2] = y$ at C_z. By Lemma 7.31, $x = I.newPrv$ is reachable at C_z.

Case 1B: I is created by a DELETE operation. By the pseudo-code, $prvNode.state$ is set to marked on line 110 before the first forward CAS of I (since $I.nodes[1] = prvNode$). By Lemma 7.23, $prvNode.state$ is marked at line 54. So, $c.node$ is set on line 58. Now, we show $c.node$ is set to $I.nodes[0]$ at line 58. By Lemma 7.28.7a and 7.28.7c, $I.nodes[1].prv = I.nodes[0]$ at the configuration after the first forward CAS of I. By Lemma 7.28.11, $I.nodes[1].prv = I.nodes[0]$ at line 56. Since $prvNode = I.nodes[1]$, $c.node$ is set to $I.nodes[0]$ at line 58, so $x = I.nodes[0]$. By Lemma 7.28.1, $I.nodes[0].nxt = I.nodes[2]$ at the configuration after the first forward CAS of I. By Lemma 7.28.7a, $x.nxt = I.nodes[0].nxt = I.nodes[2] = y$ at C_z. By Lemma 7.30, $x = I.nodes[0]$ is reachable at C_z.

Case 2: MOVELEFT(c) sets $c.node$ to $prvNode$ at line 59. Thus, $x = prvNode$. Let C_u be the configuration after reading x from $y.prv$ on line 90 during the call to UPDATECURSOR on line 50. We consider three different cases.

Case 2A: $x.nxt \neq y$ at C_u. Since $c.node$ is equal to y at the last execution of line 82, $y \neq Head$ (by Invariant 7.6.10). Since $y.prv = x$ at C_u, y is not reachable at C_u

(by Lemma 7.28.10). Since $c.node = y$ at the configuration before the last execution of line 82, y was reachable at some earlier configuration (by Lemma 7.28.13). Since the test ($y.prv.nxt \neq y$) evaluates to false in the last execution of line 82, y is reachable at the configuration before the last execution of line 82 (by Corollary 7.36). So, y becomes unreachable between the last execution of line 82 and C_u. By Lemma 7.28.5b, y becomes unreachable when the first forward CAS of some Info object I is performed, and $I.nodes[1] = y$.

Let C_k be the configuration before the first forward CAS of I. Next, we show the claim is true at C_k. By Lemma 7.28.7a and 7.28.7c, $I.nodes[1].prv = I.nodes[0]$ at C_k. By Lemma 7.28.11, $I.nodes[1].prv = I.nodes[0]$ at all configurations after C_k. Since $I.nodes[1] = y$ and $y.prv = x$ at C_u (after C_k), $I.nodes[0] = x$. By Lemma 7.28.5a, $x = I.nodes[0]$ is reachable at C_k. By Lemma 7.28.1, $x.nxt = I.nodes[0].nxt = I.nodes[1] = y$ at C_k. Since C_k is between the last execution of line 82 and C_u, the claim is true at C_k.

Case 2B: $x.nxt = y$ at C_u and the test ($prvNode.prv.nxt \neq prvNode$) evaluates to false on line 53. By Lemma 7.28.12, $prvNode = x$ is reachable at some configuration before C_u. Since the MoveLeft does not return on line 52, $prvNode \neq Head$. Since the test ($prvNode.prv.nxt \neq prvNode$) evaluates to false on line 53, $prvNode$ is reachable at the configuration before line 53 (by Corollary 7.36) which is after C_u. By Lemma 7.32, $x = prvNode$ is reachable at C_u. Since $x.nxt = y$ at C_u, the

claim holds at C_u.

Case 2C: $x.nxt = y$ at C_u and the test ($prvNode.prv.nxt \neq prvNode$) evaluates to true on line 53. Then, the test ($prvNode.nxt = node$) evaluates to false on line 53 (since $c.node$ is set on line 59). Thus, $x.nxt \neq y$ when the last step of line 53 is performed. Thus, $x.nxt$ is changed from y to another value by a forward CAS during the MoveLeft. Consider the first such forward CAS, and let I be the Info object it belongs to. Then, $I.nodes[0] = x$. At the configuration before the first forward CAS of I, x was reachable (by Lemma 7.28.5a) and $x.nxt = y$. So, the claim is true at that configuration. □

Lemma 7.40. *If* MoveLeft*(c) returns false, then there is a configuration during the* MoveLeft *after the last execution of line 82 inside the call to* UpdateCursor *on line 50 in which* $Head.nxt = c.node$.

Proof. Let $\langle y, \text{-}, \text{-}, prvNode, \text{-}, \text{-}\rangle$ be the result of the call to UpdateCursor on line 50. Then, $c.node$ is equal to y at the last execution of line 82 and $c.node$ is not changed from y to another value after the return of UpdateCursor during the MoveLeft operation.

We consider two cases according to what line returns false.

Case 1: MoveLeft returns false at line 52. Then, $prvNode = Head$. So, $y.prv = prvNode = Head$ when $y.prv$ is read on line 90. Since $c.node = y$ at the last execution of line 82, y is reachable at some earlier configuration (by Lemma

7.28.13). If y is reachable when $y.prv$ is read on line 90, since $y.prv = Head$ at that time, $Head.nxt = y$ at that time (by Lemma 7.28.10) which establishes the claim.

Now, suppose y is not reachable when $y.prv$ is read on line 90. The test ($y.prv.nxt \neq y$) on the last execution of line 82 evaluates to false. Since $y.prv = prvNode = Head$ at line 90, $y \neq Head$ (by Invariant 7.8). So, y is reachable at the configuration before the last execution of line 82 (by Corollary 7.36). By Lemma 7.28.5b, y becomes unreachable by the first forward CAS of an Info object I between the last execution of line 82 and line 90 such that $I.nodes[1] = y$.

Let C_k be the configuration before that first forward CAS of I. By Lemma 7.28.7a and 7.28.7c, $I.nodes[1].prv = I.nodes[0]$ at C_k. By Lemma 7.28.11, $I.nodes[1].prv = I.nodes[0]$ at all configurations after C_k. Since $I.nodes[1] = y$ and $y.prv = Head$ at line 90 (after C_k), $I.nodes[0] = Head$. By Lemma 7.28.1, at C_k, $Head.nxt = I.nodes[0].nxt = I.nodes[1] = y$.

Case 2: MoveLeft returns false at line 57. By Lemma 7.38, since the test on line 53 evaluates to true, $prvNode$ becomes unreachable before the end of line 53 by a forward CAS of an Info object I such that $I.nodes[1] = prvNode$ and $I.nodes[2] = y$ and no backward CAS of I occurs before the call to UpdateCursor on line 50 reads $c.node.prv$ on line 90. Thus, there is a configuration C_x during the MoveLeft that is after the first forward CAS of I but not after the first backward CAS of I.

Let C_k be the configuration after the first forward CAS of I. Since $prvNode = I.nodes[1]$, $prvNode.prv = I.nodes[0]$ at C_k (by Lemma 7.28.7a and 7.28.7c). By Lemma 7.28.11, $prvNode.prv = I.nodes[0]$ at all configurations after C_k. So, $prvPrvNode$ is set to $I.nodes[0]$ at line 56 (after C_k). Since $I.nodes[1].state$ is set to marked or copied inside a call to HELP(I) before C_k, $I.nodes[1].state$ is not ordinary at all configurations after C_k (by Observation 7.3). Since $I.nodes[1].state \neq$ copied on line 54 (after C_k), $I.nodes[1].state =$ marked on line 54. So, by Lemma 7.23, I is created by a DELETE operation. Thus, by Lemma 7.28.1, $I.nodes[0].nxt = I.nodes[2]$ at C_k. By Lemma 7.28.7a, $I.nodes[0].nxt = I.nodes[2]$ at C_x. Since the MOVELEFT returns at line 57, $I.nodes[0] = prvPrvNode = Head$. Since $I.nodes[2] = y$, $Head.nxt = y$ at C_x. □

Lemma 7.41. *If* MOVERIGHT*(c) changes c.node from x to y at line 65, then there is a configuration during the* MOVERIGHT *between the last execution of line 82 inside the call to* UPDATECURSOR *on line 62 and setting c.node on line 65 in which*

- *x is reachable, and*
- *$x.nxt = y$.*

Proof. Suppose MOVERIGHT changes *c.node* from x to y at line 65. Then, the call to UPDATECURSOR on line 62 returns $\langle x, \text{-}, y, \text{-}, \text{-}, \text{-}\rangle$. Let C_i be the configuration

before the last execution of line 82 during that call to UpdateCursor. Since *c.node* is equal to x at C_i, x is reachable at some earlier configuration (by Lemma 7.28.13) and $x \neq Head$ (by Invariant 7.6.10). Since the test ($x.prv.nxt \neq x$) evaluates to false at the last execution of line 82, x is reachable at C_i (by Corollary 7.36). If x is reachable when $x.nxt$ is read on line 90, since $x.nxt = y$ at that line, the claim is true.

Otherwise, by Lemma 7.28.5b, x becomes unreachable (between C_i and line 90) by the first forward CAS of an Info object I with $I.nodes[1] = x$. Let C_k be the configuration before the first forward CAS of I. By Lemma 7.28.7a and 7.28.7c, $I.nodes[1].nxt = I.nodes[2]$ at C_k. By Lemma 7.28.11, $I.nodes[1].nxt = I.nodes[2]$ at all configurations after C_k. Since $x = I.nodes[1]$ and $x.nxt = y$ at line 90 (after C_k), $y = I.nodes[2]$. Since $x.nxt = y$ at C_k and $x = I.nodes[1]$ is reachable at C_k (by Lemma 7.28.5a), the claim is true at C_k. □

Lemma 7.42. *If* MoveRight*(c) returns false and c.node = x when the operation returns, then there is a configuration during the* MoveRight *after the last execution of line 82 inside the call to* UpdateCursor *on line 62 when x.value = EOL, c.node = x and x is reachable.*

Proof. Let $\langle x$, -, *nxtNode*, -, -, -$\rangle$ be the result of the call to UpdateCursor at line 62. Let C_k be the configuration before the last execution of line 82 during that call to UpdateCursor. Then, *c.node* is equal to x at C_k. Since the operation

returns false at line 64 , $x.value$ = EOL at all configurations (by Observation 7.1). Since $c.node = x$ at C_k, x was reachable at some earlier configuration (by Lemma 7.28.13) and $x \neq Head$ (by Invariant 7.6.10). Since the test $x.prv.nxt \neq x$ at the last execution of line 82 evaluates to false, x is reachable at C_k (by Corollary 7.36). □

Let I be an Info object that is created by an update operation. An update operation is *successful* if there is a forward CAS of some Info object that is created by the update. A move operation is *successful* if the operation sets the Cursor's *node* field at line 54, 58, 59 or 65.

Below, we define the linearization point of each operation to be a step in the execution. In order to ensure that concurrent move and successful update operations are not linearized at the same step, we insert into the execution a null step in between every two consecutive real steps. These null steps do not change the shared memory, but are only used as the linearization points of some move operations.

Linearization Points of Operations

L1 Each InitializeCursor and ResetCursor operation that terminates is linearized when $Head.nxt$ is read at line 68 or 73, respectively.

L2 Each DestroyCursor is linearized at line 71.

L3 Each Get operation that terminates is linearized at the first step of the last execution of line 82 inside the call to UpdateCursor on line 76.

L4 Each move or update operation that returns invalidCursor is linearized at the first step of the last execution of line 82 inside the last call to UpdateCursor.

L5 Each successful update operation is linearized at the first forward CAS of an Info object I that is created by the update.

L6 Each Delete operation that returns false at line 44 is linearized at the first step of the last execution of line 82 inside the last call to UpdateCursor.

L7 Each move operation that returns false is linearized at the null step after the configuration defined by Lemma 7.40 or 7.42.

L8 Each successful move operation is linearized at the null step after the configuration defined by Lemma 7.39 or 7.41.

If several concurrent operations are linearized at exactly the same step during an execution, the order of their linearization points is arbitrary. The following lemma shows the linearization points are properly defined.

Lemma 7.43. *No operation is assigned more than one linearization point. Each operation that terminates is assigned a linearization point. The linearization point assigned to an operation is during the operation.*

Proof. It follows from linearization Rules L1, L2 and L3 that the claims hold for InitializeCursor, ResetCursor, DestroyCursor and Get.

A terminating MoveRight operation is linearized by Rule L4, L7 or L8 if it returns at line 63, 64 or 66, respectively. A terminating MoveLeft operation is linearized by Rule L4, L7 or L8 if it returns at line 51, returns at line 52 or 57 or returns at line 60, respectively. If a move operation does not terminate, it can only be linearized by Rule L7. The claims follow.

It remains to prove the claims for update operations. We show an update operation *op* can only be linearized during the last iteration of its loop at line 24–36 or 38–48. Suppose *op* creates an Info object I on line 33 or 45 of an iteration of its loop which is not the last iteration. We show that no forward CAS of I is ever executed. After creating I, *op* calls Help(I) on line 34 or 46. Since *op* does not return on line 36 or 48 of that loop iteration, $I.status$ is aborted on line 119 of that call to Help(I). By Lemma 7.17, I is not successful, so $I.nodes[2].info$ is

never set to I. So, *doPtrCAS* is true on line 109 of no invocation of HELP(I). So, no forward CAS of I is ever executed. Thus, it follows from Rules L4, L5 and L6 that a linearization point can only be assigned in the last iteration of the loop of an update operation.

If the update operation returns invalidCursor or false in the last iteration, it does not create an Info object in that iteration, so Rule L5 cannot be applied. Thus, an update cannot be assigned different linearization points by the rules.

Next, we show if an update operation terminates, it has a linearization point. If an update operation returns invalidCursor or false, it is linearized by Rule L4 or L6. Suppose an update operation *op* returns true on line 36 or 48. Let I be the Info object *op* creates on line 33 or 45 of that iteration. We show that a forward CAS of I is executed during *op*. After creating I, *op* calls HELP(I) on line 34 or 46. That call to HELP(I) returns true and $I.status$ is committed on line 119 of that call to HELP(I). So, $I.status$ is set to committed on some execution of line 116 belonging to I. Then, a forward CAS of I must also be executed on line 114. Thus, if an update operation returns true, the update is linearized during its operation by Rule L5. □

To prove the correctness of our implementation, we use an auxiliary variable of type list, which we call the *abstract list*, denoted $(\mathfrak{L}, \mathfrak{S})$. It is initially empty. Each time an operation is linearized, we update the state of the abstract list by

applying that operation (according to the sequential specification of a list). We refer to the elements of $\mathfrak{S}$ as *abstract cursors*. If c is a Cursor object, we use $\mathfrak{c}$ to denote the abstract cursor that is created at the linearization point of the InitializeCursor(c) operation.

Each item $\mathfrak{m}$ in $\mathfrak{L}$ has an associated positive real value, called its *abstract value* and denoted by $\mathfrak{m}.absVal$. The abstract value of $\mathfrak{m}$ containing EOL is 1. If $\mathfrak{m}$ is the first item in the list, and then a new item $\mathfrak{m}'$ is inserted at the beginning of the list, $\mathfrak{m}'.absVal = \frac{\mathfrak{m}.absVal}{2}$. If $\mathfrak{m}_1$ and $\mathfrak{m}_2$ are two consecutive items in the list and then a new item $\mathfrak{m}$ is inserted between them, $\mathfrak{m}.absVal = \frac{\mathfrak{m}_1.absVal+\mathfrak{m}_2.absVal}{2}$.

Since Cursors are stored in local memory, updates by one process cannot update another process's local Cursor. Thus, Cursors sometimes become out of date. A Cursor c can sometimes point to a Node that is no longer reachable. The Cursor c is later brought up to date by a call to UpdateCursor. We formally define $realNode(x)$ to be the reachable Node that c should point to if the Cursor is pointing at x. In other words, it is the location the Cursor would be updated to if UpdateCursor were called on it. For any configuration, we define

$$realNode(x) = \begin{cases} realNode(x.copy) & \text{if } x.state = \text{copied and } x \text{ is unreachable,} \\ realNode(x.nxt) & \text{if } x.state = \text{marked and } x \text{ is unreachable,} \\ x & \text{otherwise.} \end{cases}$$

Next, we argue that $realNode(x)$ is well-defined and non-null. Consider a configuration C. Since $x.state$ is initially ordinary (by Observation 7.3), if $x.state \neq$ ordinary at C, $x.state$ is set earlier inside a call to the HELP routine and x was reachable just before that (by Lemma 7.28.7b). If, at C, $x.state =$ copied and x is unreachable, x became unreachable by a forward CAS of an Info object that is created by an INSERTBEFORE operation (by Lemma 7.28.5b). Since $x.copy$ is set on line 112 before the forward CAS, $x.copy$ is not null at C. Moreover, at C, $x.copy$ points to a Node that was created after x on line 31. Similarly, if, at C, $x.state =$ marked and x is unreachable, x became unreachable by a forward CAS of an Info object that is created by DELETE (by Lemma 7.28.5b). By Lemma 7.28.11, $x.nxt$ is not changed after the forward CAS, so at all configurations after the forward CAS, $x.nxt$ is the Node that was next in the list at the configuration before the forward CAS.

In C, there are a finite number of Nodes that have ever been created. We define the following total order on all Nodes in C: $x <^* y \equiv (x.absVal > y.absVal)$ or ($x.absVal = y.absVal$ and x was created after y). So, for a Node x, if $x.nxt$ is not null, $x.nxt <^* x$ (by Invariant 7.8) and if $x.copy$ is not null, $x.copy <^* x$ (since $x.copy$ is set to a Node whose $absVal$ is equal to $x.absVal$ and that is created after x). Note that, if $realNode(x)$ in configuration C is defined to be $realNode(y)$ then $y <^* x$. So, the recursive definition of $realNode$ is well-defined.

We say *the realNode path* of a Node x at a configuration C is the sequence of Nodes that UPDATECURSOR would follow to reach a reachable Node if it were performed atomically in configuration C on a Cursor pointing at x. More formally,

$$realNodePath(x) = \begin{cases} \langle x \rangle \cdot realNodePath(x.copy) & \text{if } x.state = \text{copied and } x \text{ is unreachable,} \\ \langle x \rangle \cdot realNodePath(x.nxt) & \text{if } x.state = \text{marked and } x \text{ is unreachable,} \\ \langle x \rangle & \text{otherwise.} \end{cases}$$

Lemma 7.44. *Let x be a Node. Only a successful forward CAS can change $realNodePath(x)$ (or $realNode(x)$).*

Proof. When the *state* field of a Node is changed from ordinary to marked or copied on line 110 or 113, the Node is reachable (by Lemma 7.23 and 7.28.7b). By Lemma 7.28.5b), only a forward CAS of an Info object I with $I.nodes[1] = x$ can make a Node x unreachable. Then, $x.state$ is set to marked or copied on line 110 or 113 before the first forward CAS of I and $x.state$ is not changed after that (by Lemma 7.23). If I is created by an INSERTBEFORE operation, $x.copy$ is set before the first forward CAS of I and $x.copy$ is not changed after that (by Lemma 7.26). If I is created by a DELETE operation, $x.nxt$ is not changed after the first forward CAS of I (by Lemma 7.28.11). □

Suppose *op* is an InitializeCursor, ResetCursor, successful move or successful update operation that is called with Cursor c. If *op* terminates, it changes $\mathfrak{c}.item$ at its linearization point and then later it sets $c.node$. Between these two steps, c's location does not correspond to $\mathfrak{c}$'s. To keep track of this difference, we define a prophecy variable $c.updatedNode$ for each Cursor c as follows.

The state of $c.updatedNode$ in a configuration C is equal to $c.node$ except in the following cases.

- If C is after the linearization point of an InsertBefore(c, -), but not after a step of that InsertBefore(c, -) that sets $c.node$ on line 35, and the InsertBefore is linearized at the first forward CAS of an Info object I, then $c.updatedNode = I.newPrv$. (This is the Node that $c.node$ will be updated to on line 35 if the InsertBefore ever reaches that line.)

- If C is after the linearization point of a Delete(c) but not after a step of that Delete(c) that sets $c.node$ on line 47, and the Delete is linearized at the first forward CAS of an Info object I, then $c.updatedNode = I.nodes[2]$. (This is the Node that $c.node$ will be updated to on line 47 if the Delete ever reaches that line.)

- If C is

  – between the linearization point of a MoveLeft(c) and a step of that

MoveLeft(c) that sets $c.node$ on line 54, 58, 59,

- between the linearization point of a MoveRight(c) and a step of that MoveRight(c) that sets $c.node$ on line 65,
- between the linearization point of an InitializeCursor(c) and a step of that InitializeCursor that sets $c.node$ on line 68, or
- between the linearization point of a ResetCursor(c) and a step of that ResetCursor that sets $c.node$ on line 73,

then $c.updatedNode$ is the Node that $c.node$ is set to by the step on line 54, 58, 59, 65, 68 or 73, respectively.

We shall prove an invariant that the reachable Node that Cursor c should "really" be pointing to is $realNode(c.updatedNode)$. The difference between $c.node$ and $c.updatedNode$ is the result of a Cursor operation on c that has been linearized but has not yet updated $c.node$, and the difference between $c.updatedNode$ and $realNode(c.updatedNode)$ is the result of update operations by other processes that may have made $c.node$ out of date.

Lemma 7.45. *For each Cursor c, a step can change $realNodePath(c.updatedNode)$ only if it is a linearization point of an operation or an execution of line 85 or 88.*

Proof. By Lemma 7.44, only a successful forward CAS can change $realNodePath(y)$ for a Node y. Since only the first forward CAS of an Info object succeeds (by Lemma 7.28.1), a successful update operation is linearized at that step. By definition of $c.updatedNode$, $c.updatedNode$ can be changed only by a linearization point of an operation and by a step that sets $c.node$. Let S be a step that changes $c.node$ and C be the configuration before S and C' be the configuration after S. We show that S does not change $realNodePath(c.updatedNode)$ if S is not an execution of line 85 or 88.

If S is an execution of line 54, 58, 59, 65, 68 or 73, S does not change $c.updatedNode$ (by definition of $c.updatedNode$).

Suppose S is an execution of line 35 or 47 and an operation op executes S. Then, before S, op creates an Info object I on line 33 or 45 and op's call to Help(I) on line 34 or 46 returns true. So, $I.status$ is set to committed on line 116 inside a call to Help(I) and before that, the first forward CAS of I is executed. By Lemma 7.43 and Rule L5, op is linearized at the first forward CAS of I. If S is an execution of line 35, S sets $c.node$ to $I.newPrv$. If S is an execution of line 47, S sets $c.node$ to $I.nodes[2]$. So, S does not change $c.updatedNode$ (by definition of $c.updatedNode$). □

Lemma 7.46. *For each Cursor c, a step can change $realNode(c.updatedNode)$ only if it is a linearization point of an operation.*

Proof. By Lemma 7.45, a step can change $realNodePath(c.updatedNode)$ only if it is a linearization point of an operation or an execution of line 85 or 88. Let S be a step that changes $c.node$ on line 85 or 88 and C be the configuration before S and C' be the configuration after S. We show that S does not change $realNode(c.updatedNode)$.

Let x be the value of $c.node$ in C. By Lemma 7.28.13, x was reachable at an earlier configuration. Since the test ($x.prv.nxt \neq x$) evaluates to true in the execution of line 82 that precedes S, x is not reachable in the configuration after that execution (by Corollary 7.36). By Lemma 7.32, x is unreachable in C. By Lemma 7.28.5b, x becomes unreachable by the first forward CAS of an Info object I and $x = I.nodes[1]$. Then, $I.nodes[1].state$ is set to copied or marked before the first forward CAS of I. By Lemma 7.23, $x.state$ is not changed after the first execution of line 110 or 113 belonging to I.

If S is an execution of line 85, the test ($x.state =$ copied) evaluates to true in line 83. Then, $x.state =$ copied in C. Since x is unreachable in C, $realNode(x) = realNode(x.copy)$ in C. Since S changes $c.node$ from x to $x.copy$, S does not change $realNode(c.node)$.

If S is an execution of line 88, the test ($x.state =$ marked) evaluates to true in line 86. Then, $x.state =$ marked in C. Since x is unreachable in C, $realNode(x) = realNode(x.nxt)$ in C. Since S changes $c.node$ from x to $x.nxt$, S does not change

$realNode(c.node)$. □

Lemma 7.47. *If an operation op other than* DestroyCursor *is called with Cursor c, c.updatedNode is reachable at the configuration after op's linearization point.*

Proof. Let S be the linearization point of an operation *op* called with Cursor c and C be the configuration after S. We show that $c.updatedNode$ is reachable in C.

Suppose *op* is an operation that returns invalidCursor, or a Delete operation that returns false, or a Get operation. By Rule L4, L6 and L3, S is the first step of the last execution of line 82 inside *op*'s last call to UpdateCursor. By definition of $c.updatedNode$, $c.updatedNode = c.node$ in C. By Corollary 7.36, $c.node$ is reachable at the configuration before S, so $c.node$ is reachable in C.

Suppose *op* is a successful MoveRight or MoveLeft operation. Then, *op* changes $c.node$ to some Node x on line 54, 58, 59 or 65. By Rule L8, S is the null step after the configuration defined by Lemma 7.39 or 7.41. By definition of $c.updatedNode$, $c.updatedNode = x$ in C. By Lemma 7.39 and 7.41, x is reachable in C.

Suppose *op* is a MoveRight or MoveLeft operation that returns false. By Rule L7, S is the null step after the configuration defined by Lemma 7.40 or 7.42. By definition of $c.updatedNode$, $c.updatedNode = c.node$ in C. By Lemma 7.40 and 7.42, $c.node$ is reachable in C.

Suppose *op* is a successful update operation. By Rule L5, S is the first forward

CAS of an Info object I. If op is an InsertBefore operation, by definition of $c.updatedNode$, $c.updatedNode = I.newPrv$ in C. By Lemma 7.31, $I.newPrv$ is reachable in C. If op is a Delete operation, by definition of $c.updatedNode$, $c.updatedNode = I.nodes[2]$ in C. By Lemma 7.28.5a, $I.nodes[2]$ is reachable in C.

Suppose op is an InitializeCursor or ResetCursor operation. By Rule L1, S is when $Head.nxt$ is read on line 68 or 73. Let x be the value of $Head.nxt$ in C. So, x is reachable in C. By definition of $c.updatedNode$, $c.updatedNode = x$ in C. □

We now prove a collection of related statements by induction to show that changes to the data structure match changes in the abstract list, where operations are performed atomically at their linearization points, and that each operation returns the same result as the corresponding abstract operation. This is formalized in Statement 1 of the following lemma. The rest of the statements describes how a configuration of our implementation represent the state of the abstract list. Statement 2 shows the elements of the abstract list match the reachable Nodes and Statement 3, 4 and 5 show how each Cursor c matches the abstract cursor $\mathfrak{c}$.

Lemma 7.48. *For an execution α, the following statements are true.*

1. *Suppose an operation op is linearized at step S in α. Then,*

   (a) *If op terminates in α with result r, the corresponding abstract operation*

applied atomically to the state of $(\mathfrak{L}, \mathfrak{S})$ *in the configuration before* S *also returns* r.

(b) *If op does not terminate in* α, *the corresponding abstract operation applied atomically to the state of* $(\mathfrak{L}, \mathfrak{S})$ *in the configuration before* S *returns true.*

2. *In every configuration of* α, *the sequence of abstract values and values of Nodes that are reachable (excluding* $Head$ *and* $Tail$*) and of items in* $\mathfrak{L}$ *are equal.*

3. *In every configuration of* α, *for each* $\mathfrak{c}$ *in* $\mathfrak{S}$, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal$.

4. *In every configuration* C *of* α, $\mathfrak{c}.invIns$ *is true if and only if either*

   (a) *there is a Node* x *on* $realNodePath(c.updatedNode)$ *such that* $x.state =$ *copied and* x *is unreachable, or*

   (b) C *is between the invocation of an operation op called with Cursor* c *and the linearization point of op and the local variable* $invIns$ *of the process performing that operation is true.*

5. *In every configuration* C *of* α, $\mathfrak{c}.invDel$ *is true if and only if either*

(a) there is a Node x on $realNodePath(c.updatedNode)$ such that $x.state =$ marked and x is unreachable, or

(b) C is between the invocation of an operation op called with Cursor c and the linearization point of op and the local variable $invDel$ of the process performing that operation is true.

Proof. Consider an execution $\alpha = C_0\ S_1\ C_1\ S_2\ C_2....$ In configuration C_0, the only reachable Node other than $Head$ and $Tail$ is $EOLnode$ and $\mathfrak{L}$ is $\langle$EOL$\rangle$. So, Statement 2 is true in C_0. Statements 3, 4 and 5 are trivially true in C_0. Let $k > 0$. We assume Statements 2–5 are true in C_{k-1}. We then prove that Statement 1 holds for any operation linearized at step S_k and that Statement 2–5 are true in C_k.

Statement 1 Suppose an operation op is linearized at S_k in α. Then, the corresponding abstract operation is applied to the list's state in C_{k-1}. We consider different cases according to the type of op.

Case 1: op is an InitializeCursor(c), DestroyCursor(c) or ResetCursor(c) operation. Then. op returns ack on line 69, 71 or 74. By the specification of InitializeCursor, DestroyCursor and ResetCursor (see Table 6.1), the corresponding abstract operation would also return ack.

Case 2: op is an operation that is called with Cursor c and returns invalidCursor in α. We show that if op is an InsertBefore, $\mathfrak{c}.invDel$ or $\mathfrak{c}.invIns$ is true in

C_{k-1}; otherwise $\mathfrak{c}.invDel$ is true in C_{k-1}. By Rule L3 or L4, S_k is the first step of the last execution of line 82 inside the last call to UpdateCursor. If *op* is an InsertBefore operation, since *op* returns on line 26, *op*'s local variable $invDel$ or $invIns$ is true in C_{k-1}. So, $\mathfrak{c}.invDel$ or $\mathfrak{c}.invIns$ is true in C_{k-1} (by induction hypothesis 4 and 5). Otherwise, since *op* returns on line 40, 51, 63, 77, *op*'s local variable $invDel$ is true in C_{k-1}. So, $\mathfrak{c}.invDel$ is true in C_{k-1} (by induction hypothesis 5). Hence, by the specification of operations (see Table 6.1–6.3), in either case, the corresponding abstract operation would also return invalidCursor.

Case 3: *op* is a successful Delete(c) in α. We show that in C_{k-1}, $\mathfrak{c}.invDel =$ false and $\mathfrak{c}.item.value \neq$ EOL. By Rule L5, S_k is the first forward CAS of an Info object I created by *op*. Then, $c.node = I.nodes[1]$ in C_{k-1}. By definition of $c.updatedNode$, $c.updatedNode = c.node = I.nodes[1]$ in C_{k-1}. By Lemma 7.28.5a, $I.nodes[1]$ is reachable at C_{k-1}. So, $realNodePath(c.updatedNode) = realNodePath(I.nodes[1]) = \langle I.nodes[1] \rangle$ in C_{k-1} (by definition of $realNodePath$). Since *op* does not return on line 40, *op*'s local variable $invDel$ is false in C_{k-1}. Since, in C_{k-1}, $realNodePath(c.updated) = \langle I.nodes[1] \rangle$, $I.nodes[1]$ is reachable and *op*'s local variable $invDel$ is false, $\mathfrak{c}.invDel$ is false in C_{k-1} (by induction hypothesis 5). Since $c.updatedNode = I.nodes[1]$ in C_{k-1} and $I.nodes[1]$ is reachable at C_{k-1}, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(I.nodes[1]).absVal = I.nodes[1].absVal$ (by induction hypothesis 3). By

Corollary 7.11, no two reachable Nodes have the same abstract values. Since $I.nodes[1]$ is reachable in C_{k-1} and $I.nodes[1].absVal = \mathfrak{c}.item.absVal$ in C_{k-1}, $\mathfrak{c}.item.value = I.nodes[1].value$ in C_{k-1} (by induction hypothesis 2). Since DELETE does not return on line 44 and $c.node = I.nodes[1]$ in C_{k-1}, $I.nodes[1].value \neq$ EOL in C_{k-1} (by Observation 7.1). Since $\mathfrak{c}.item.value = I.nodes[1].value$ in C_{k-1}, $\mathfrak{c}.item.value$ is not EOL in C_{k-1}. Hence, by the specification of DELETE (see Table 6.2), the corresponding abstract operation returns true.

Case 4: *op* is a successful INSERTBEFORE(c, -) in α. We show that, in C_{k-1}, $\mathfrak{c}.invDel =$ false and $\mathfrak{c}.invIns =$ false. By Rule L5, S_k is the first forward CAS of an Info object I created by *op*. Then, $c.node = I.nodes[1]$ in C_{k-1}. By definition of $c.updatedNode$, $c.updatedNode = c.node = I.nodes[1]$ in C_{k-1}. By Lemma 7.28.5a, $I.nodes[1]$ is reachable at C_{k-1}. So, $realNodePath(c.updatedNode) = realNodePath(I.nodes[1]) = \langle I.nodes[1] \rangle$ in C_{k-1} (by definition of $realNodePath$). Since *op* does not return on line 26, *op*'s local variable $invDel$ and $invIns$ are false in C_{k-1}. Since, in C_{k-1}, $realNodePath(c.updated) = \langle I.nodes[1] \rangle$, $I.nodes[1]$ is reachable and *op*'s local variable $invIns$ and $invDel$ are false, $\mathfrak{c}.invDel$ and $\mathfrak{c}.invIns$ are false in C_{k-1} (by induction hypothesis 4 and 5). Hence, by the specification of INSERTBEFORE (see Table 6.1), the corresponding abstract operation returns true.

Case 5: *op* is a DELETE(c) operation that returns false in α. By Rule L6, S_k is the first step of the last execution of line 82 inside the last call to UP-

DATECURSOR. We show that, in C_{k-1}, $\mathfrak{c}.invDel$ is false and $\mathfrak{c}.item.value$ = EOL. Let x be the value of $c.node$ in C_{k-1}. By definition of $c.updatedNode$, in C_{k-1}, $c.updatedNode = c.node = x$. Since the test $(x.prv.nxt \neq x)$ in the last execution of line 82 inside the last call to UPDATECURSOR evaluates to false, x is reachable in C_{k-1} (by Corollary 7.36). Since $c.updatedNode = x$ in C_{k-1}, $realNodePath(c.updatedNode) = realNodePath(x) = \langle x \rangle$ in C_{k-1} (by definition of $realNodePath$). Since op does not return on line 40, op's local variable $invDel$ is false in C_{k-1}. Since, in C_{k-1}, $realNodePath(c.updated) = \langle x \rangle$, x is reachable and op's local variable $invDel$ is false, $\mathfrak{c}.invDel$ is false in C_{k-1} (by induction hypothesis 5). Since $c.node = x$ is reachable in C_{k-1}, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). By Corollary 7.11, no two reachable Nodes have the same abstract values. Since x is reachable in C_{k-1}, $\mathfrak{c}.item.value = x.value$ in C_{k-1} (by induction hypothesis 2). Since DELETE returns false on line 44, $x.value$ = EOL in C_{k-1} (by Observation 7.1). Since $\mathfrak{c}.item.value = x.value$ in C_{k-1}, $\mathfrak{c}.item.value$ = EOL in C_{k-1}. Hence, by the specification of DELETE (see Table 6.2), the corresponding abstract operation returns false.

Case 6: op is a MOVERIGHT(c) operation that returns false in α. We show that, in C_{k-1}, $\mathfrak{c}.invDel$ = false and $\mathfrak{c}.item$ is the last item in $\mathfrak{L}$. By Rule L7, S_k is the null step after the configuration defined by Lemma 7.42. Let x be the value

of $c.node$ in C_{k-1}. By Lemma 7.42, in C_{k-1}, $x.value$ = EOL and x is reachable. By definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1}. Since, in C_{k-1}, x is reachable, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). By Corollary 7.11, no two reachable Nodes have the same abstract values. Since x is reachable in C_{k-1} and $\mathfrak{c}.item.absVal = x.absVal$ in C_{k-1}, $\mathfrak{c}.item.value = x.value$ in C_{k-1} (by induction hypothesis 2). Since $x.value$ = EOL in C_{k-1}, $\mathfrak{c}.item.value$ = EOL in C_{k-1} and $\mathfrak{c}.item$ is the last item in $\mathfrak{L}$ in C_{k-1} (by specification of $\mathfrak{L}$).

Next, we show that $\mathfrak{c}.invDel$ is false at C_{k-1} using induction hypothesis 5. Since $c.updatedNode = x$ in C_{k-1} and x is reachable in C_{k-1}, $realNodePath(c.updatedNode) = realNodePath(x) = \langle x \rangle$ in C_{k-1}. Since MoveRight does not return on line 63, op's local variable $invDel$ is false in C_{k-1}. Since, in C_{k-1}, $realNodePath(c.updated) = \langle x \rangle$ and x is reachable and op's local variable $invDel$ is false, $\mathfrak{c}.invDel$ is false in C_{k-1} (by induction hypothesis 5). Hence, by the specification of MoveRight (see Table 6.2), the corresponding abstract operation also returns false.

Case 7: op is a MoveLeft(c) operation that returns false in α. We show that, in C_{k-1}, $\mathfrak{c}.invDel$ = false and $\mathfrak{c}.item$ is the first item in $\mathfrak{L}$. By Rule L7, S_k is the null step after the configuration defined by Lemma 7.40. Let x be the value of $c.node$ in C_{k-1}. By Lemma 7.40, in C_{k-1}, $Head.nxt = c.node = x$. By

definition of $c.updatedNode$, in C_{k-1}, $c.updatedNode = c.node = x$. Since x is reachable in C_{k-1}, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). By Corollary 7.11, no two reachable Nodes have the same abstract values. Since $Head.nxt = x$ in C_{k-1}, $\mathfrak{c}.item$ is the first item in $\mathfrak{L}$ in C_{k-1} (by induction hypothesis 2).

Next, we show that $\mathfrak{c}.invDel$ is false in C_{k-1}. Since $c.updatedNode = x$ in C_{k-1} and x is reachable in C_{k-1}, $realNodePath(c.updatedNode) = realNodePath(x) = \langle x \rangle$ in C_{k-1} (by definition of $realNodePath$). Since MoveLeft does not return on line 51, op's local variable $invDel$ is false in C_{k-1}. Since, in C_{k-1}, $realNodePath(c.updated) = \langle x \rangle$, x is reachable and op's local variable $invDel$ is false, $\mathfrak{c}.invDel$ is false in C_{k-1} (by induction hypothesis 5). Hence, by the specification of MoveLeft (see Table 6.3), the corresponding abstract operation also returns false.

Case 8: op is a successful MoveRight(c) operation in α. Then, op changes $c.node$ from some Node x to some Node y on line 65. We show that, in C_{k-1}, $\mathfrak{c}.item$ is not the last item in $\mathfrak{L}$ and $\mathfrak{c}.invDel$ = false. By Rule L8, S_k is the null step after the configuration defined by Lemma 7.41. Since $c.node$ is not changed between C_{k-1} and the execution of line 65, $c.node = x$ in C_{k-1}. By definition of $c.updatedNode$, in C_{k-1}, $c.updatedNode = c.node = x$. By Lemma 7.41, x is reachable in C_{k-1}. So, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal =$

$realNode(x).absVal = x.absVal$ (by induction hypothesis 3). Since MoveRight does not return on line 64, $x.value \neq$ EOL in C_{k-1} (by Observation 7.1). By Corollary 7.11, no two reachable Nodes have the same abstract values. Since $\mathfrak{c}.item.absVal = x.absVal$ in C_{k-1} and x is reachable in C_{k-1}, $\mathfrak{c}.item.value = x.value \neq$ EOL in C_{k-1} (by induction hypothesis 2). So, $\mathfrak{c}.item$ is not the last item in $\mathfrak{L}$ in C_{k-1} (by specification of $\mathfrak{L}$).

Next, we show that $\mathfrak{c}.invDel$ is not true at C_{k-1} using induction hypothesis 5. Since, in C_{k-1}, x is reachable and $c.updatedNode = x$, $realNodePath(c.updatedNode) = realNodePath(x) = \langle x \rangle$ in C_{k-1} (by definition of $realNodePath$). Since MoveRight does not return on line 63, op's local variable $invDel$ is false in C_{k-1}. Since, in C_{k-1}, $realNodePath(c.updated) = \langle x \rangle$ and x is reachable and op's local variable $invDel$ is false, $\mathfrak{c}.invDel$ is false in C_{k-1} (by induction hypothesis 5). Hence, by the specification of MoveRight (see Table 6.2), the corresponding abstract operation also returns true.

Case 9: op is a successful MoveLeft(c) operation in α. Then, op changes $c.node$ from some Node x to some Node y on line 54, 58 or 59. We show that, in C_{k-1}, $\mathfrak{c}.invDel =$ false and $\mathfrak{c}.item$ is not the first item in $\mathfrak{L}$. By Rule L8, S_k is the null step after the configuration defined by Lemma 7.39. Since $c.node$ is not changed between C_{k-1} and the execution of line 54, 58 or 59 (by Lemma 7.39), $c.node = x$ in C_{k-1}. By definition of $c.updatedNode$, $c.updatedNode = c.node = x$

in C_{k-1}. By Lemma 7.39, y is reachable and $y.nxt = x$ in C_{k-1}. Since $c.updatedNode = x$ in C_{k-1} and x is reachable in C_{k-1}, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). Since $c.node$ is set to y on line 54, 58 or 59, $y \neq Head$ (by Invariant 7.6.10). Since $y.nxt = x$ in C_{k-1} and $y \neq Head$ and $\mathfrak{c}.item.absVal = x.absVal$ in C_{k-1}, $\mathfrak{c}.item$ is not the first item in $\mathfrak{L}$ in C_{k-1} (by induction hypothesis 2).

Next, we show that $\mathfrak{c}.invDel$ is false at C_{k-1}. Since $c.updatedNode = x$ in C_{k-1} and x is reachable in C_{k-1}, $realNodePath(c.updatedNode) = realNodePath(x) = \langle x \rangle$ in C_{k-1}. Since MoveLeft does not return on line 51, op's local variable $invDel$ is false in C_{k-1}. Since, in C_{k-1}, $realNodePath(c.updated) = \langle x \rangle$ and x is reachable and op's local variable $invDel$ is false, $\mathfrak{c}.invDel$ is false in C_{k-1} (by induction hypothesis 5). Hence, by the specification of MoveLeft (see Table 6.3), the corresponding abstract operation also returns true.

Case 10: op is a Get operation that returns $v \neq$ invalidCursor (already covered in Case 2) in α. We show that, in C_{k-1}, $\mathfrak{c}.invDel =$ false and $\mathfrak{c}.item.value = v$. Suppose the call U to UpdateCursor on line 76 returns $\langle x, -, -, -, invDel, - \rangle$. Then, $x.value = v$ in C_{k-1} (by Observation 7.1). By Rule L3, S_k is the first step of the last execution of line 82 inside U. By definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1}. Since the test $(x.prv.nxt \neq x)$ evaluates to false on the last execution of line 82, x is reachable in C_{k-1} (by Corollary 7.36). Since

$c.updatedNode = x$, $realNodePath(c.updatedNode) = realNodePath(x) = \langle x \rangle$ in C_{k-1} (by definition of $realNodePath$). Since GET does not return invalidCursor on line 77, $invDel$ is false. Since, in C_{k-1}, $realNodePath(c.updated) = \langle x \rangle$ and x is reachable and op's local variable $invDel$ is false, $\mathfrak{c}.invDel$ is false in C_{k-1} (by induction hypothesis 5). Since, in C_{k-1}, $c.updatedNode = x$ and x is reachable, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). By Corollary 7.11, no two reachable Nodes have the same abstract values. Since x is reachable in C_{k-1}, $\mathfrak{c}.item.value = x.value = v$ in C_{k-1} (by induction hypothesis 2). Hence, by the specification of GET (see Table 6.1), the corresponding abstract operation returns v.

Statement 2 By Statement 1, proved above, the abstract operations corresponding to unsuccessful update operations that return invalidCursor or false do not change $\mathfrak{L}$. So, $\mathfrak{L}$ is changed only at the linearization point of a successful update operation in α, which is the first forward CAS of an Info object created by that operation. By Lemma 7.28.1, this forward CAS is successful. Likewise, the sequence of reachable Nodes can only be changed by a successful forward CAS. Thus, if S_k is not a successful forward CAS, Statement 2 is trivially true by induction hypothesis 2.

Now, suppose S_k is a successful forward CAS of an Info object I created by operation *op*. By Lemma 7.28.1, S_k is the first forward CAS of I. By Lemma 7.43, S_k is the first forward CAS of any Info object created by *op*, so that S_k is the linearization point of *op* and hence changes $\mathfrak{L}$.

In C_{k-1}, $I.nodes[0].nxt = I.nodes[1]$ (by Lemma 7.28.7a and 7.28.7c). So, $I.nodes[1] \neq Head$ (by Invariant 7.6.7). Since, in C_{k-1}, $I.nodes[1].prv = I.nodes[0]$ and $I.nodes[2].prv = I.nodes[1]$ (by Lemma 7.28.7a and 7.28.7c), $I.nodes[0] \neq Tail$ and $I.nodes[1] \neq Tail$ (by Invariant 7.6.8). By Lemma 7.28.5a, $I.nodes[0]$, $I.nodes[1]$ and $I.nodes[2]$ are reachable in C_{k-1}. By induction hypothesis 2, in C_{k-1}, the sequence of abstract values and values of Nodes that are reachable (excluding $Head$ and $Tail$) and items in $\mathfrak{L}$ are equal. So, in C_{k-1}, there is an item $\mathfrak{m}_1$ in $\mathfrak{L}$ such that $I.nodes[1].absVal = \mathfrak{m}_1.absVal$ and $I.nodes[1].value = \mathfrak{m}_1.value$. If $I.nodes[0]$ is not $Head$, then in C_{k-1}, there is an item $\mathfrak{m}_0$ just before $\mathfrak{m}_1$ in $\mathfrak{L}$ such that $I.nodes[0].absVal = \mathfrak{m}_0.absVal$ and $I.nodes[0].value = \mathfrak{m}_0.value$.

Next, we show that $\mathfrak{c}.item = \mathfrak{m}_1$ in C_{k-1}. Since $c.node = I.nodes[1]$ in C_{k-1}, $c.updatedNode = c.node = I.nodes[1]$ (by definition of $c.updatedNode$). Since $I.nodes[1]$ is reachable in C_{k-1}, in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(I.nodes[1]).absVal = I.nodes[1].absVal$ (by induction hypothesis 3). So, $\mathfrak{c}.item.absVal = I.nodes[1].absVal = \mathfrak{m}_1.absVal$ in C_{k-1}. Since the abstract values of reach-

able Nodes are unique (by Invariant 7.8), the abstract values of items in $\mathfrak{L}$ are unique in C_{k-1} (by induction hypothesis 2). So, $\mathfrak{c}.item = \mathfrak{m}_1$ in C_{k-1}. Next, we consider two cases according to what type of operation created I.

Case 1: I is created by an InsertBefore(c, v) operation. Since $\mathfrak{c}.item = \mathfrak{m}_1$ in C_{k-1}, S_k inserts a new item $\mathfrak{m}'$ with value is v just before $\mathfrak{m}_1$ in $\mathfrak{L}$. If $\mathfrak{m}_1$ is the first element in the list, $\mathfrak{m}'.absVal = \frac{\mathfrak{m}_1.absVal}{2}$. Otherwise, $\mathfrak{m}'.absVal$ is $\frac{\mathfrak{m}_0.absVal+\mathfrak{m}_1.absVal}{2}$. By Lemma 7.28.1, S_k sets $I.nodes[0].nxt$ to $I.newNxt$. By Lemma 7.28.3a, in C_k, $I.newNxt.nxt = I.newPrv$ and $I.newPrv.nxt = I.nodes[2]$. By Lemma 7.28.5b, only $I.nodes[1]$ becomes unreachable by S_k. By Lemma 7.28.8, only $I.newPrv$ and $I.newNxt$ become reachable by S_k. Next, we show that $I.newPrv.value = \mathfrak{m}_1.value$, $I.newPrv.absVal = \mathfrak{m}_1.absVal$, $I.newNxt.value = \mathfrak{m}'.value$ and $I.newNxt.absVal = \mathfrak{m}'.absVal$. Since $I.newPrv$ is created on line 31, $I.newPrv.value = I.nodes[1].value = \mathfrak{m}_1.value$ and $I.newPrv.absVal = I.nodes[1].absVal = \mathfrak{m}_1.absVal$. Since $I.newNxt$ is created on line 30, $I.newNxt.value = v = \mathfrak{m}'.value$ and $I.newNxt.absVal = \frac{I.nodes[0].absVal+I.nodes[1].absVal}{2}$. If $I.nodes[0] = Head$, since, in C_{k-1}, $Head.nxt = I.nodes[1]$ and $I.nodes[1].absVal = \mathfrak{m}_1.absVal$, $\mathfrak{m}_1$ is the first item in $\mathfrak{L}$ in C_{k-1} (by induction hypothesis 2). Since $Head.absVal = 0$, $I.newNxt.absVal = \frac{I.nodes[1].absVal}{2} = \frac{\mathfrak{m}_1.absVal}{2} = \mathfrak{m}'.absVal$. Otherwise, $I.newNxt.absVal = \frac{I.nodes[0].absVal+I.nodes[1].absVal}{2} = \frac{\mathfrak{m}_0.absVal+\mathfrak{m}_1.absVal}{2} = \mathfrak{m}'.absVal$.

Thus, $I.newNxt.absVal = \mathfrak{m}'.absVal$.

S_k inserts the new item $\mathfrak{m}'$ just before $\mathfrak{m}_1$ in $\mathfrak{L}$. By Lemma 7.28.3a, 7.28.7a and 7.28.7c, in C_k, $I.nodes[0].nxt = I.newNxt$, $I.newNxt.nxt = I.newPrv$ and $I.newPrv.nxt = I.nodes[2]$. Thus, S_k replaces $I.nodes[1]$ by $I.newNxt$ and $I.newPrv$ in the list of reachable Nodes. The $value$ and $absVal$ of $I.nodes[1]$ and $I.newPrv$ are equal, and the $value$ and $absVal$ of $I.newNxt$ are the same as for $\mathfrak{m}'$. Thus, Statement 2 is true in C_k.

Case 2: I is created by a DELETE(c) operation. Since $\mathfrak{c}.item = \mathfrak{m}_1$ in C_{k-1}, S_k removes $\mathfrak{m}_1$ from $\mathfrak{L}$. Since S_k sets $I.nodes[0].nxt$ to $I.nodes[2]$ and $I.nodes[2]$ was reachable at C_{k-1}, no Node becomes reachable by S_k. By Lemma 7.28.5b, only $I.nodes[1]$ becomes unreachable at C_k. Since $\mathfrak{m}_1.value = I.nodes[1].value$ and $\mathfrak{m}_1.absVal = I.nodes[1].absVal$, Statement 2 is true at C_k.

Statement 3 The only step that adds a new abstract cursor to $\mathfrak{S}$ is the linearization point of INITIALIZECURSOR. For each abstract cursor $\mathfrak{c}$ in $\mathfrak{S}$, the only step that can change $\mathfrak{c}.item$ is a linearization point of an operation. By Lemma 7.46, for each Cursor c, the only step that can change $realNode(c.updatedNode)$ is a linearization point of an operation. So, if S_k is not a linearization point of an operation, Statement 3 is trivially true (by induction hypothesis 3). So, suppose S_k is a linearization point of an operation op. Then, we show that Statement 3 is

true in C_k. We consider different cases according to what kind of operation *op* is.

Case 1: *op* is an InitializeCursor(c) or ResetCursor(c) that terminates. By Rule L1, S_k is when $Head.nxt$ is read on line 68 or 73. By Statement 1, proved earlier, the corresponding abstract operation sets $\mathfrak{c}.item$ to the first item in $\mathfrak{L}$. Let x be the value of $Head.nxt$ in C_{k-1}. Since $Head.nxt = x$ in C_k, the $absVal$ of the first item in $\mathfrak{L}$ is $x.absVal$ in C_k (by Statement 2, proved earlier). So, $\mathfrak{c}.item.absVal = x.absVal$ in C_k. After C_k, *op* sets $c.node$ to x on line 68 or 73. So, by the definition of $c.updatedNode$, $c.updatedNode = x$ in C_k. Since, in C_k, $\mathfrak{c}.item.absVal = x.absVal$ and x is reachable and $c.updatedNode = x$, in C_k, $\mathfrak{c}.item.absVal = x.absVal = realNode(x).absVal = realNode(c.updatedNode).absVal$.

Case 2: *op* is a DestroyCursor(c). By Rule L2, S_k is line 71. By Statement 1, proved earlier, the corresponding abstract operation removes $\mathfrak{c}$ from $\mathfrak{S}$. So, Statement 3 is vacuously satisfied for $\mathfrak{c}$ in C_k.

Case 3 : *op* is an operation that is called with c and returns invalidCursor or *op* is a Delete(c) operation that returns false or *op* is a Get(c) operation that terminates. By Rule L4, L6 and L3, S_k is the first step of the last execution of line 82 inside the last call to updateCursor. By Statement 1, the corresponding abstract operation does not change $\mathfrak{c}.item$. By induction hypothesis 3, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal$ in C_{k-1}. Since S_k does not change $c.node$, S_k does not change $c.updatedNode$ (by definition of $c.updatedNode$). So, by

Lemma 7.44, S_k does not change $realNode(c.updatedNode)$. Since $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal$ in C_{k-1}, the same equality holds in C_k.

Case 4: *op* is a successful DELETE(c) operation. By Rule L5, S_k is the first forward CAS of an Info object I. Then, $c.node = I.nodes[1]$ in C_{k-1}. By definition of $c.updatedNode$, $c.updatedNode = c.node = I.nodes[1]$ in C_{k-1}. Since $I.nodes[1]$ is reachable in C_{k-1} (by Lemma 7.28.5a), in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(I.nodes[1]).absVal = I.nodes[1].absVal$ (by induction hypothesis 3). By Statement 1, proved above, the corresponding abstract operation is also successful, so the step S_k advances $\mathfrak{c}.item$ to the next item in $\mathfrak{L}$ according to Table 6.2. By Lemma 7.28.7a and 7.28.7c, $I.nodes[1].nxt = I.nodes[2]$ in C_{k-1}. By Lemma 7.28.5a, $I.nodes[2]$ is reachable in C_{k-1}. Let $\mathfrak{m}$ be the item in $\mathfrak{L}$ in C_{k-1} corresponding to $I.nodes[2]$ such that $\mathfrak{m}.absVal = I.nodes[2].absVal$. By Corollary 7.11, no two reachable Nodes have the same abstract values. Since, in C_{k-1}, $I.nodes[1].nxt = I.nodes[2]$ and $I.nodes[1]$ is reachable and $\mathfrak{c}.item.absVal = I.nodes[1].abVal$, in C_{k-1}, $\mathfrak{m}$ is the next item after $\mathfrak{c}.item$ in $\mathfrak{L}$ in C_{k-1} (by induction hypothesis 2). So, S_k advances $\mathfrak{c}.item$ to $\mathfrak{m}$. Next, we show that $\mathfrak{m}.absVal = realNode(c.updatedNode).absVal$ in C_k. By definition of $c.updatedNode$, $c.updatedNode = I.nodes[2]$ in C_k. Since, in C_k, $I.nodes[2]$ is reachable, $\mathfrak{c}.item = \mathfrak{m}$ and $\mathfrak{m}.absVal = I.nodes[2].absVal$, it follows

that in C_k, $realNode(c.updatedNode).absVal = realNode(I.nodes[2]).absVal = I.nodes[2].absVal = \mathfrak{m}.absVal = \mathfrak{c}.item.absVal$.

Case 5: *op* is a successful InsertBefore(c, -) operation. By Rule L5, S_k is the first forward CAS of an Info object I. Then, $c.node = I.nodes[1]$ in C_{k-1}. By definition of $c.updatedNode$, $c.updatedNode = c.node = I.nodes[1]$ in C_{k-1}. Since $I.nodes[1]$ is reachable in C_{k-1} (by Lemma 7.28.5a), in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(I.nodes[1]).absVal = I.nodes[1].absVal$ (by induction hypothesis 3). By the specification of InsertBefore (see Table 6.1), the step S_k does not change $\mathfrak{c}.item$. Next, we show that $\mathfrak{c}.item.absVal = I.nodes[1].absVal = realNode(c.updatedNode).absVal$ in C_k. By definition of $c.updatedNode$, $c.updatedNode = I.newPrv$ in C_k. Since $I.newPrv$ is created on line 31 preceding the creation of I, $I.newPrv.absVal = I.nodes[1].absVal$. By Lemma 7.31, $I.newPrv$ is reachable in C_k. Since, in C_k, $c.updatedNode = I.newPrv$, $I.newPrv$ is reachable and $\mathfrak{c}.item.abVal = I.nodes[1].abVal = I.newPrv.absVal$, it follows in C_k, $realNode(c.updatedNode) = realNode(I.newPrv).absVal = I.newPrv.absVal = I.nodes[1].absVal = \mathfrak{c}.item.absVal$.

Case 6: *op* is a MoveRight(c) or MoveLeft(c) operation that returns false. By Rule L7, S_k is the null step after the configuration defined by Lemma 7.42 or 7.40. Let x be the value of $c.node$ in C_{k-1} and C_k. By the defini-

tion of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1} and C_k. Since x is reachable in C_{k-1} (by Lemma 7.42 and 7.40), in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatednode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). By Statement 1, proved above, the corresponding abstract operation does not change $\mathfrak{c}.item$. Since S_k is the null step after C_{k-1}, x is still reachable in C_k. Since, in C_k, $\mathfrak{c}.item.absVal = x.absVal$, x is reachable and $c.updatedNode = x$, it follows that in C_k, $\mathfrak{c}.item.absVal = x.absVal = realNode(x).absVal = realNode(c.updatedNode)$.

Case 7: *op* is a successful MoveRight(c). By Rule L8, S_k is the null step after the configuration defined by Lemma 7.41. Then, this MoveRight subsequently changes $c.node$ from some Node x to some Node y on line 65. By Lemma 7.41, $c.node = x$ in C_{k-1}. By definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1} and $c.updatedNode = y$ in C_k. Since x is reachable in C_{k-1} (by Lemma 7.41), in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). By Statement 1, proved above, the corresponding abstract operation is also successful, so the step S_k advances $\mathfrak{c}.item$ to the next item in $\mathfrak{L}$ by Table 6.2. By Lemma 7.41, in C_{k-1}, x is reachable and $x.nxt = y$. Since *op* does not return on line 64, $x.value \neq$ EOL. Since $x.nxt = y$ in C_{k-1}, y is neither $Tail$ nor $Head$ (by Invariant 7.6.1 and 7.6.7). So, there is the item in $\mathfrak{L}$ in C_{k-1} corresponding to y in C_{k-1} such that

$\mathfrak{m}.absVal = y.absVal$ (by induction hypothesis 2). Let $\mathfrak{m}$ be that item. By Corollary 7.11, no two reachable Nodes have the same abstract values. Since, in C_{k-1}, $x.nxt = y$ and x is reachable and $\mathfrak{c}.item.absVal = x.absVal$, $\mathfrak{m}$ is the next item after $\mathfrak{c}.item$ in $\mathfrak{L}$ in C_{k-1} (by induction hypothesis 2). So, S_k advances $\mathfrak{c}.item$ to $\mathfrak{m}$. Thus, in C_k, $\mathfrak{c}.item.absVal = \mathfrak{m}.absVal = y.absVal = realNode(y).absVal$ (since y is reachable in C_k by Lemma 7.41). Since, in C_k, $c.updatedNode = y$, it follows that in C_k, $\mathfrak{c}.item.absVal = realNode(y).absVal = realNode(c.updatedNode).absVal$.

Case 8: *op* is a successful MoveLeft(c). By Rule L8, S_k is the null step after the configuration defined by Lemma 7.39. Then, this MoveLeft subsequently changes $c.node$ from some Node x to some Node y on line 54, 58 or 59. By Lemma 7.39, $c.node = x$ in C_{k-1}. By the definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1} and $c.updatedNode = y$ in C_k. Since x is reachable in C_{k-1} (by Lemma 7.39), in C_{k-1}, $\mathfrak{c}.item.absVal = realNode(c.updatednode).absVal = realNode(x).absVal = x.absVal$ (by induction hypothesis 3). By Statement 1, proved above, the corresponding abstract operation is also successful, so the step S_k sets $\mathfrak{c}.item$ to the previous item before $\mathfrak{c}.item$ in $\mathfrak{L}$ by Table 6.3. If *op* sets $c.node$ to y on line 58 or 59, since *op* read y from the *prv* field of a Node on line 56 or 90, y is not $Tail$ (by Invariant 7.6.8). Since *op* does not return on line 52 or 57, y is not $Head$. If *op* sets $c.node$ to y on line 54, since y is read from the *copy* field of a Node on line 54, y is neither $Head$ nor $Tail$ (by Invariant 7.6.9). So, there is the item

corresponding to y in $\mathfrak{L}$ in C_{k-1} such that $\mathfrak{m}.absVal = y.absVal$. Let $\mathfrak{m}$ be that item. By Lemma 7.39, in C_{k-1}, y is reachable and $y.nxt = x$. By Corollary 7.11, no two reachable Nodes have the same abstract values. In C_{k-1}, $y.nxt = x$ and y is reachable and $\mathfrak{c}.item.absVal = x.abVal$, so $\mathfrak{m}$ is the item before $\mathfrak{c}.item$ in $\mathfrak{L}$ in C_{k-1} (by induction hypothesis 2). So, S_k changes $\mathfrak{c}.item$ to $\mathfrak{m}$. Since, in C_k, $\mathfrak{c}.item = \mathfrak{m}$, $\mathfrak{m}.absVal = y.absVal$ and y is reachable, it follows in C_k, $\mathfrak{c}.item.absVal = \mathfrak{m}.absVal = y.absVal = realNode(y).absVal$. So, in C_k, $c.updatedNode = y$ and $\mathfrak{c}.item.absVal = realNode(y).absVal = realNode(c.updatedNode).absVal$.

Statement 4 Let c be a Cursor. We show that if S_k changes $\mathfrak{c}.invIns$, or the truth value of Statement 4a or 4b, Statement 4 is true in C_k. The value of $\mathfrak{c}.invIns$ can be changed only by the linearization points of an operation called with c or the linearization point of a successful InsertBefore operation called with Cursor $c' \neq c$ such that $\mathfrak{c}.item = \mathfrak{c}'.item$ at the configuration before the linearization point of the InsertBefore.

Next, we determine what steps can change the truth value of Statement 4a. When the *state* of a Node is set to copied on line 113, the Node is reachable (by Lemma 7.28.7b). So, this step does not affect the truth value of Statement 4a. By Lemma 7.28.5b, the only step that makes a Node x unreachable is the first forward CAS of an Info object I, which is the linearization point of the successful

update operation *op* that created I (by Lemma 7.43). By Lemma 7.45, only a linearization point of an operation or an execution of line 85 or 88 can change $realNodePath(c.updatedNode)$.

The truth value of Statement 4b can be changed only by the invocation of an operation called with c, the linearization of an operation called with c and setting the local variable $invIns$ of an operation that is called with c on line 84.

Thus, the only steps S_k that we must consider are linearization points of operations, executions of line 84, 85 or 88 and invocations of an operation called with Cursor c. We consider these in turn.

Case 1: S_k is the linearization point of some operation *op* called with c. Then, $\mathbf{c}.invIns$ is false in C_k (by the specifications of operations in Table 6.1–6.3). Since C_k is after *op*'s linearization point, Statement 4b is false in C_k. We show that Statement 4a is also false in C_k. By Lemma 7.47, $c.updatedNode$ is reachable in C_k. So, by definition of $realNodePath$, the only Node on $realNodePath(c.updatedNode)$ in C_k is $c.updatedNode$. Since $c.updatedNode$ is reachable in C_k, Statement 4a is false.

Case 2: S_k is the linearization point of an operation that is called with Cursor $c' \neq c$ and is not a successful InsertBefore or Delete. Then, S_k does not change $\mathbf{c}.invIns$ or the truth value of Statement 4a or 4b.

Case 3: S_k is the linearization point of a successful Delete operation *op* that is called with Cursor $c' \neq c$. By the specification of a Delete operation (in Table

6.2), *op* does not change $\mathfrak{c}.invIns$. By Rule L5, S_k is the first forward CAS of an Info object I. By Lemma 7.28.5b, the only Node that becomes unreachable by S_k is $I.nodes[1]$ and $I.nodes[1].state$ is set to marked before S_k. By Lemma 7.23, $I.nodes[1].state$ is marked in C_k. So, if $realNode(c.updatedNode)$ was $I.nodes[1]$ in C_{k-1}, then it become $I.nodes[2]$ in C_k and $I.nodes[2]$ is reachable in C_k (by Lemma 7.28.5a). Thus, S_k does not change the truth value of Statement 4a and the claim is true in C_k.

Case 4: S_k is the linearization point of a successful InsertBefore operation *op* that is called with Cursor $c' \neq c$. By Rule L5, S_k is the first forward CAS of an Info object I created by *op*. By definition of $c'.updatedNode$, $c'.updatedNode = c'.node = I.nodes[1]$ in C_{k-1}. So, in C_{k-1}, $\mathfrak{c}'.item.absVal = realNode(c'.updatedNode).absVal = realNode(I.nodes[1]).absVal$ (by Statement 3). By Lemma 7.28.5a, $I.nodes[1]$ is reachable in C_{k-1}. By the definition of $realNode$, in C_{k-1}, $\mathfrak{c}'.item.absVal = realNode(I.nodes[1]).absVal = I.nodes[1].absVal$. We consider two cases.

Case 4A: $\mathfrak{c}.item \neq \mathfrak{c}'.item$ in C_{k-1}. By the specification of an InsertBefore (in Table 6.2), S_k does not change $\mathfrak{c}.invIns$. We show that S_k does not change the truth value of Statement 4a. By the definition of $realNodePath$ and $realNode$, the only reachable Node on $realNodePath(c.updatedNode)$ is $realNode(c.updatedNode)$. By Lemma 7.28.5b, the only Node that be-

comes unreachable by S_k is $I.nodes[1]$. By Statement 3, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal \neq I.nodes[1].absVal$ in C_{k-1}. Since the only reachable Node on $realNodePath(c.updatedNode)$ is $realNode(c.updatedNode)$, S_k does not change the truth value of Statement 4a.

Case 4B: $\mathfrak{c}.item = \mathfrak{c}'.item$ in C_{k-1}. By the specification of InsertBefore (in Table 6.1), S_k sets $\mathfrak{c}.invIns$ to true. We now show that Statement 4a is true in C_k. Since $\mathfrak{c}.item = \mathfrak{c}'.item$ in C_{k-1} and $\mathfrak{c}'.item.absVal = I.nodes[1].absVal$ in C_{k-1}, $\mathfrak{c}.item.absVal = I.nodes[1].absVal$ in C_{k-1}. Since $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal$ in C_{k-1} (by induction hypothesis 3) and $I.nodes[1]$ is reachable in C_{k-1} (by Lemma 7.28.5a), $realNode(c.updatedNode) = I.nodes[1]$. So, $I.nodes[1]$ is on $realNodePath(c.updatedNode)$ in C_{k-1} (by definition of $realNodePath$). Next, we show that, in C_k, $I.nodes[1]$ is unreachable and $I.nodes[1].state$ is copied. Since I is created by an InsertBefore operation, $I.nodes[1].state$ is set to copied before S_k. By Lemma 7.23, $I.nodes[1].state =$ copied in C_k. By Lemma 7.28.5a, $I.nodes[1]$ is unreachable in C_k. Thus, Statement 4a is true in C_k.

Case 5: S_k is an execution of line 84 or 85. Let x be the value of $c.node$ in C_{k-1}. By Corollary 7.36, x is unreachable at the configuration after the execution of line 82 that precedes S_k. By Lemma 7.32, x is unreachable in C_{k-1}. Since the test ($x.state =$ copied) evaluates to true in line 83, $x.state$ is copied in C_{k-1} (by

Lemma 7.23). By the definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1}. By the definition of $realNodePath$, x is on $realNodePath(c.updatedNode)$ in C_{k-1}. So, Statement 4a is true in C_{k-1} and, by induction hypothesis 4, $\mathfrak{c}.invIns$ is true in C_{k-1} and hence in C_k. Next, we show that Statement 4b is true in C_k. Let op be the operation that executes S_k. If S_k is the execution of line 84, S_k sets op's local variable $invIns$ to true. Otherwise, op's local variable $invIns$ is set to true on line 84 before S_k. By the definition of linearization points of operations, op is not linearized before S_k. Since S_k cannot be the linearization point of any operation called with c and op's local variable $invIns$ is true in C_k, Statement 4b is true in C_k.

Case 6: S_k is an execution of line 88. Since the test ($x.state$ = marked) evaluates to true in line 86, $x.state$ is marked in C_{k-1} (by Lemma 7.23). By the definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1}. By the definition of $realNodePath$, x is on $realNodePath(c.updatedNode)$ in C_{k-1}. Since S_k changes $c.node$ from x to $x.nxt$, $c.updatedNode = x.nxt$ in C_k (by the definition of $c.updatedNode$). So, x might be removed from $realNodePath(c.updatedNode)$. Since $x.state \neq$ copied in C_{k-1}, S_k does not affect Statement 4a.

Case 7: S_k is the invocation of an operation called with c. Then, Statement 4b is false in C_k. By induction hypothesis 4, $\mathfrak{c}.invIns$ was true in C_{k-1} if and only if Statement 4a was true in C_{k-1}. So, S_k affects neither $\mathfrak{c}.invIns$ nor Statement 4a,

and Statement 4b is false in C_k, so Statement 4 holds in C_k.

Statement 5 The proof of Statement 5 is very similar to the proof of Statement 4. Let c be a Cursor. We show that if S_k changes $\mathbf{c}.invDel$, or Statement 5a or 5b, Statement 5 is true in C_k. As in proof of Statement 4, the only steps that can affect $\mathbf{c}.invDel$, the truth value of Statement 5a or 5b are the linearization points of operations, the invocation of an operation called with c and an execution of line 85, 87 or 88. We consider different cases according to what step is S_k.

Case 1: If S_k is the linearization point of some operation op called with c, then $\mathbf{c}.invDel$ is false in C_k (by the specifications of operations in Table 6.1–6.3). Since C_k is after op's linearization point, Statement 5b is false in C_k. We show that Statement 5a is also false in C_k. By Lemma 7.47, $c.updatedNode$ is reachable in C_k. So, by definition of $realNodePath$, the only Node on $realNodePath(c.updatedNode)$ is $c.updatedNode$ in C_k. Since $c.updatedNode$ is reachable in C_k, Statement 5a is false.

Case 2: S_k is the linearization point of an operation that is called with Cursor $c' \neq c$ and is not a successful InsertBefore or Delete. Then, S_k does not change $\mathbf{c}.invDel$, the truth value of Statement 5a or 5b.

Case 3: S_k is the linearization point of a successful InsertBefore operation op that is called with Cursor $c' \neq c$. By the specification of an InsertBefore operation (in Table 6.1), op does not change $\mathbf{c}.invDel$. By Rule L5, S_k is the first

forward CAS of an Info object I. By Lemma 7.28.5b, the only Node that becomes unreachable by S_k is $I.nodes[1]$ and $I.nodes[1].state$ is set to copied before S_k. By Lemma 7.23, $I.nodes[1].state$ is copied in C_k. Since $I.nodes[1].copy$ is set to $I.newPrv$ before S_k, $I.nodes.copy = I.newPrv$ in C_{k-1} (by Lemma 7.26). So, if $realNode(c.updatedNode)$ was $I.nodes[1]$ in C_{k-1}, then it become $I.newPrv$ in C_k and $I.newPrv$ is reachable in C_k (by Lemma 7.28.8). Thus, S_k does not change the truth value of Statement 5a and the claim is true in C_k.

Case 4: Suppose S_k is the linearization point of a successful DELETE operation op that is called with Cursor $c' \neq c$. By Rule L5, S_k is the first forward CAS of an Info object I created by op. By definition of $c'.updatedNode$, $c'.updatedNode = c'.node = I.nodes[1]$ in C_{k-1}. So, in C_{k-1}, $\mathfrak{c}'.item.absVal = realNode(c'.updatedNode).absVal = realNode(I.nodes[1]).absVal$ (by Statement 3). By Lemma 7.28.5a, $I.nodes[1]$ is reachable in C_{k-1}. By the definition of $realNode$, in C_{k-1}, $\mathfrak{c}'.item.absVal = realNode(I.nodes[1]).absVal = I.nodes[1].absVal$. We consider two cases.

Case 4A: $\mathfrak{c}.item \neq \mathfrak{c}'.item$ in C_{k-1}. By the specification of a DELETE (in Table 6.2), S_k does not change $\mathfrak{c}.invDel$. We show that S_k does not change the truth value of Statement 5a. By the definition of $realNodePath$ and $realNode$, the only reachable Node on $realNodePath(c.updatedNode)$ is $realNode(c.updatedNode)$. By Lemma 7.28.5b, the only Node that be-

comes unreachable by S_k is $I.nodes[1]$. By Statement 3, $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal \neq I.nodes[1].absVal$ in C_{k-1}. Since the only reachable Node on $realNodePath(c.updatedNode)$ is $realNode(c.updatedNode)$, S_k does not change the truth value of Statement 5a.

Case 4B: $\mathfrak{c}.item = \mathfrak{c}'.item$ in C_{k-1}. By the specification of a DELETE operation (in Table 6.2), S_k sets $\mathfrak{c}.invDel$ to true. We show that Statement 5a is true in C_k. Since $\mathfrak{c}.item = \mathfrak{c}'.item$ in C_{k-1} and $\mathfrak{c}'.item.absVal = I.nodes[1].absVal$ in C_{k-1}, $\mathfrak{c}.item.absVal = I.nodes[1].absVal$ in C_{k-1}. Since, in C_{k-1}, $I.nodes[1]$ is reachable and $\mathfrak{c}.item.absVal = realNode(c.updatedNode).absVal$ in C_{k-1} (by induction hypothesis 3), $realNode(c.updatedNode) = I.nodes[1]$. So, $I.nodes[1]$ is on $realNodePath(c.updatedNode)$ in C_{k-1} (by definition of $realNodePath$). Next, we show that, in C_k, $I.nodes[1]$ is unreachable and $I.nodes[1].state$ is marked. Since I is created by a DELETE operation, $I.nodes[1].state$ is set to marked before S_k. By Lemma 7.23, $I.nodes[1].state$ = marked in C_k. By Lemma 7.28.5a, $I.nodes[1]$ is unreachable in C_k. Thus, Statement 5a is true in C_k.

Case 5: Suppose S_k is an execution of line 87 or 88. Let x be the value of $c.node$ in C_{k-1}. By Corollary 7.36, x is unreachable at the configuration after the execution of line 82 that precedes S_k. By Lemma 7.32, x is unreachable in C_{k-1}. Since the test ($x.state$ = marked) evaluates to true in line 86, $x.state$ is marked in C_{k-1} (by Lemma 7.23). By the definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in

C_{k-1}. By the definition of $realNodePath$, x is on $realNodePath(c.updatedNode)$ in C_{k-1}. So, Statement 5a is true in C_{k-1} and, by induction hypothesis 5, $\mathfrak{c}.invDel$ is true in C_{k-1} and hence in C_k. Next, we show that Statement 5b is true in C_k. Let op be the operation that executes S_k. If S_k is an execution of line 87, S_k sets op's local variable $invDel$ to true. Otherwise, op's local variable $invDel$ is set to true on line 87 before S_k. By definition of linearization points of operations, op is not linearized before S_k. Since S_k cannot be the linearization point of any operation called with c and op's local variable $invDel$ is true in C_k, Statement 5b is true in C_k.

Case 6: S_k is an execution of line 85. Since the test ($x.state$ = copied) evaluates to true in line 83, $x.state$ is copied in C_{k-1} (by Lemma 7.23). By the definition of $c.updatedNode$, $c.updatedNode = c.node = x$ in C_{k-1}. By the definition of $realNodePath$, x is on $realNodePath(c.updatedNode)$ in C_{k-1}. Since S_k changes $c.node$ from x to $x.copy$, $c.updatedNode = x.copy$ in C_k (by the definition of $c.updatedNode$). So, x might be removed from $realNodePath(c.updatedNode)$. Since $x.state \neq$ marked in C_{k-1}, S_k does not affect Statement 4a.

Case 7: If S_k is the invocation of an operation called with c, then Statement 5b is false in C_k. By induction hypothesis 5, $\mathfrak{c}.invDel$ was true in C_{k-1} if and only if Statement 5a was true in C_{k-1}. So, S_k affects neither $\mathfrak{c}.invDel$ nor Statement 5a, and Statement 5b is false in C_k, so Statement 5 holds in C_k. □

By Lemma 7.43, each operation that terminates is assigned to exactly one lin-

earization point and no operation can be assigned to more than one linearization point. Move operations called with different Cursors might be linearized at the same step in the execution. In this case, their linearization points can be ordered arbitrarily without affecting the results of operations. No other two operations can be assigned the same linearization point. So, by Lemma 7.48.1, we have the following theorem.

Theorem 7.49. *The implementation is linearizable.*

8 Performance of Doubly-linked List

There are different ways to evaluate the performance of a concurrent implementation. One way is *empirical evaluation* which can be done, for example, by running experiments on multi-core machines. The empirical evaluation shows how an implementation behaves and scales in practice under some specific workload. It can also be used to compare the performance of different implementations on specific machines using some specific benchmarks.

On the other hand, a theoretical analysis of the worst-case complexity of an implementation provides a performance guarantee under all possible workloads. In a non-blocking implementation, some processes might starve, so we cannot provide a bound on the number of the steps of an individual operation. Instead, we provide an *amortized analysis*. To analyze the amortized complexity of an implementation, the sequences of operations in all possible finite executions are considered. Then, a worst-case bound on the total number of steps performed by the sequence of operations is computed.

Definition 8.1. *For each operation op and every finite execution α, the function $f(op, \alpha)$ is an amortized bound on the number of steps of an implementation if, for all α, the total number of steps in α is at most $\sum_{op \in \alpha} f(op, \alpha)$ where the sum is taken over all operations op invoked in α.*

For example, consider a tree implementation. Let $h(\alpha)$ be the maximum height of the tree at any time during the execution α. If the total number of steps taken by all operations in any execution α (in which n operations are invoked) is at most $n(h(\alpha))^3$, then $h(\alpha)^3$ is an amortized bound on the number of steps of each operation.

In the remainder of this chapter, we provide both an amortized analysis and an empirical evaluation of our doubly-linked list.

8.1 Amortized Analysis of the Doubly-linked List

Here, we provide an amortized bound for our doubly-linked list implementation. Some parts of our analysis are similar to the analysis of search trees by Ellen et al. [12], though the parts of the analysis that deal with cursors and move operations are original. Ellen et al. used a combination of an aggregate analysis and an accounting method argument. We present our analysis in a simpler way using the potential method and show how to generalize their argument to handle operations that flag more than two nodes. At the end of this section, using the amortized complexity

of the implementation, we also prove that our doubly-linked list implementation is non-blocking.

A Cursor c is *active* if it has been initialized by calling InitializeCursor, but not yet destroyed by calling DestroyCursor. Let $\dot{c}(op)$ be the maximum number of active Cursors at any configuration during operation op. We show that the amortized complexity of each update (InsertBefore or Delete) operation op is $O(\dot{c}(op))$ and the amortized complexity of each other operation is $O(1)$. More precisely, we fix a finite execution α and prove that the total number of steps performed in α is $O(\sum_{op \text{ is an update in } \alpha} \dot{c}(op) + \sum_{op \text{ is other than update in } \alpha} 1)$.

To prove the amortized bound, we use the *potential method*: We define a potential function and show the changes to the potential function caused by each update op is $O(\dot{c}(op))$ and by each other operation is $O(1)$.

We start with some definitions. Each loop iteration of line 24–36 or 38–48 inside an update operation is called an *attempt* of the update operation. A complete attempt is *resolved* if it returns on line 26, 36, 40, 44 or 48; otherwise the complete attempt is *unresolved*. So, a complete attempt is unresolved if its call to CheckInfo on line 29 or 43 or its call to Help on line 34 or 46 returns false. Each iteration of line 82–88 inside UpdateCursor and each attempt of an update operation (excluding the call to UpdateCursor) take $O(1)$ steps. For simplicity in our proof, we assume they take one unit of time. In addition, since each move op-

eration (excluding the call to UpdateCursor) takes $O(1)$ steps, we assume each move operation (excluding the call to UpdateCursor) takes one unit of time.

If an operation performs a step that is not inside a call to Help, we say that step *belongs to* the operation. A move operation that sets *c.node* on line 54, 58, 59 or 65 is linearized at the null step after the configuration that is defined by Lemma 7.39 or 7.41. We say that null step *belongs to* the move operation. Let I be an Info object created by an update operation *op*. We say that any step inside any call to Help(I) *belongs* to *op*, even if it is performed by a process different from the one that invoked *op*.

To bound the amortized cost of operations, we first show that the steps belonging to each iteration of lines 82–88 inside UpdateCursor decrease the potential function by 1. Then, we show that the steps belonging to a move operation (excluding the call to UpdateCursor) do not change the potential. It follows the amortized complexity of a move operation is $O(1)$. After that, we show the amortized cost on each update operation *op* is $O(\dot{c}(op))$: We first show that the steps belonging to each unresolved attempt of *op* (excluding the call to UpdateCursor) decrease the potential function by at least 1. In addition, we show that the steps belonging to the last attempt of *op* increase the potential function by $O(\dot{c}(op))$.

8.1.1 The Potential Function

Here, we define the potential function Φ for our doubly-linked list implementation. The potential function Φ consists of three functions Φ_{cursor}, Φ_{state} and Φ_{flag}, which we define in turn.

The Potential Φ_{cursor} Each update operation deletes or replaces at most one Node during its last attempt. Any Cursor c whose true location (which is $realNode(c.updatedNode)$) is at that Node will have to perform one iteration of line 82–88 when UpdateCursor(c) is next called to follow the nxt or $copy$ pointer of the Node. So, the total number of iterations of line 82–88 in the execution is at most $\sum_{op \text{ is an update}} \dot{c}(op)$. The goal is to define Φ_{cursor} so that it bounds the total number of iterations of line 82–88 in the execution: Φ_{cursor} decreases by steps belonging to each iteration of loop 82–88 inside UpdateCursor and increases only by steps belonging to the last attempt of an update. Let c be a Cursor. Consider a call to UpdateCursor(c), denoted U. Let x be the value of $c.node$ on an execution of line 82 during U. Suppose U performs an iteration of loop 82–88, denoted itr. Then, the test ($x.prv.nxt \neq x$) evaluated to true at that execution of line 82. Since $c.node = x$ on line 82 of itr, $x \neq Head$ (by Invariant 7.6.10) and x was reachable earlier (by Lemma 7.28.13). So, x is unreachable at the configuration after line 82 of itr (by Corollary 7.36). By Lemma 7.28.5b, x became unreachable earlier by a

successful forward CAS of an Info object I such that $I.nodes[1] = x$. Then, $x.state$ is set to marked or copied inside a call to HELP(I) before the successful forward CAS of I. Thus, U performs itr and updates $c.node$ on line 85 or 88 inside itr only because an update performed the successful forward CAS of I earlier. Let u be a Node. Recall that

$$realNode(u) = \begin{cases} realNode(u.copy) & \text{if } u.state = \text{copied and } u \text{ is unreachable,} \\ realNode(u.nxt) & \text{if } u.state = \text{marked and } u \text{ is unreachable,} \\ u & \text{otherwise.} \end{cases}$$

Each iteration of the loop 82–88 inside a call to the UPDATECURSOR(c) corresponds to one step of the recursive definition of $realNode(c.node)$. (Note that $c.node = c.updatedNode$ at any configuration during a call to UPDATECURSOR by the definition of $c.updatedNode$.) Below, we define the $length$ of a Node that counts the number of pointers that the $realNode$ of the Node goes through to get to a reachable Node. The value of $length(c.node)$ increases when the forward CAS of I succeeds and then decreases when $c.node$ is updated on line 85 or 88 during itr. Below, we use $length$ to define Φ_{cursor}.

$$length(u) = \begin{cases} length(u.copy) + 1 & \text{if } u.state = \text{copied and } u \text{ is unreachable,} \\ length(u.nxt) + 1 & \text{if } u.state = \text{marked and } u \text{ is unreachable,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\phi_{cursor}(c) = \begin{cases} length(u) & \text{if there is a move called with } c \text{ that sets } c.node \\ & \text{to } u \text{ on line 54, 58, 59 or 65 and the configuration} \\ & \text{is between the linearization point of that move} \\ & \text{and its execution of line 54, 58, 59 or 65,} \\ length(c.node) & \text{otherwise.} \end{cases}$$

$$\Phi_{cursor} = \sum_{c \text{ is active}} \phi_{cursor}(c).$$

The Potential Φ_{state} The goal is to define Φ_{state} so that it bounds the total number of attempts that are unresolved because one of the Nodes to be flagged is observed to be marked or copied when CheckInfo returns false on line 97. Consider a Cursor c that is active when an update op sets $x.state$ to copied or marked. This causes at most two attempts of c's updates to be unresolved: if an attempt of an update op' fails when reading $x.state$ at line 97, line 94 of the next attempt ensures op is completed and no subsequent attempt reaches x. To pay for these attempts, op stores $2\dot{c}(op)$ of potential when op sets $x.state$. There is one other possibility: an operation op' might be called with a Cursor that is created *after* op sets $x.state$ to marked or copied. Again, at most two attempts of op' might fail because of reading $x.state$ at line 97. To pay for these attempts, op' stores $2\dot{c}(op')$ of potential when it is invoked, since there are at most $\dot{c}(op')$ reachable Nodes that are marked or copied when op' begins. Thus, we shall prove that the

total number of such unresolved attempts is $O(\sum_{op \text{ is an update}} \dot{c}(op))$.

To bound such unresolved attempts, Φ_{state} decreases by steps belonging to an unresolved attempt whose call to CheckInfo on line 29 or 43 returns false on line 97 and increases only by steps belonging to the last attempt of an update. Since a successful forward CAS step might change the Node whose *state* an attempt wishes to set, such a step also increases Φ_{state}.

For an active update operation *op*, we define an auxiliary variable $\phi_{state}(op)$ that is initially set to 2 when *op* is invoked and is updated as follows:

set to 2	when a forward or backward CAS succeeds,
set to 2	when a Node's *state* is changed from ordinary to marked or copied,
decremented	when $\phi_{state}(op) > 0$ and *op* reads marked or copied from a Node's *state* field on line 97.

Let

$$\Phi_{state} = \sum_{op} \phi_{state}(op)$$

where the sum is taken over all active update operations *op*.

The Potential Φ_{flag} The potential function Φ_{flag} is the most intricate function of our analysis. Φ_{flag} is used to bound the number of attempts that are unresolved because the call to CheckInfo returns false on line 95 or 100 or the call to Help on line 34 or 46 returns false. The goal is to define Φ_{flag} so that the steps belonging

to such an unresolved attempt decrease Φ_{flag} and only the steps belonging to the last attempt of an update increase Φ_{flag}.

Each of the types of unresolved attempts described above are caused by other operations flagging a Node the unresolved attempt wishes to flag. However, since a successful flag CAS step might, itself, belong to an unresolved attempt, Φ_{flag} cannot simply be increased when a flag CAS succeeds. Instead, we come up with a fairly complex scheme where Φ_{flag} is increased by (1) successful forward and backward CAS steps, (2) changing the *status* of an Info object from inProgress to committed and (3) invocations of update operations. This potential is used to pay for attempts whose calls to CheckInfo return false on line 95 or 100 or whose calls to Help on line 34 or 46 return false.

To define Φ_{flag}, we first define some auxiliary variables. An operation is *active* if the operation has been invoked but it has not yet terminated. For an active update operation op that is called with Cursor c, we have the following definitions.

$$\begin{aligned} node_1(op) = &\quad realNode(c.node) \\ node_0(op) = &\quad \text{the reachable Node whose } nxt \text{ pointer is } node_1(op) \\ node_2(op) = &\quad \text{the Node that } node_1(op).nxt \text{ points to} \end{aligned}$$

These are three adjacent Nodes in the list, where the middle one is the "true" location of c (as defined by $realNode(c.node)$).

Next, for $i = 0, 1, 2$, we define an auxiliary variable $lose_i(op)$ that is initially

set to 3 when op is invoked and is updated as follows:

set to 3	when a forward or backward CAS succeeds,
set to 3	when some other operation sets the *info* of $node_i(op)$,
set to 2	when, for some Info object I created by op, the first flag CAS of I on $I.nodes[i]$ fails,
decremented	when $lose_i(op) > 0$ and the read of $oldInfo[i].status$ in op's line 93 reads inProgress,
decremented	when $lose_i(op) > 0$ and the read of $nodes[i].info$ in op's line 99 reads a value different from $oldInfo[i]$.

Let u be a Node and let

$$\text{flag}(u) = \begin{cases} 1 & \text{if } u.info.status \text{ is inProgress, (i.e., } u \text{ is flagged)} \\ 0 & \text{otherwise.} \end{cases}$$

Next, we define an auxiliary variable abort(u) that is initially set to 0 and is updated as follows:

set to 1	when a flag CAS on u's successor succeeds and $u = node_i(op')$ for some $0 \leq i < 3$ and some active update operation op',
set to 0	when $u.nxt$ is set by a forward CAS,
set to 0	when $u.info.status$ is changed from inProgress to committed or aborted,

set to 0 when an update op' terminates and $u = node_i(op')$ for some $0 \leq i < 3$ and there is no update op'' other than op' such that $u = node_j(op')$ for some $0 \leq j < 3$.

Let v and w be Nodes. A Node w is *after v in the list* if w and v both are reachable and $w.absVal \geq v.absVal$. (Note that, by definition, a reachable Node is after itself in the list.) Let

$$\phi_{flag}(v) \;=\; \sum_{w \text{ is after } v \text{ in the list}} (\text{abort}(w) - \text{flag}(w)).$$

Let $\dot{u}$ at a configuration be the number of active update operations at that configuration. Since each update operation is called with a distinct Cursor, $\dot{u}$ at a configuration is $\leq \dot{c}(op)$ for any active update operation op at that configuration.

$$\Phi_{flag} \;=\; \sum_{op}(3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op)) + \sum_{i=0}^{2} lose_i(op)) + 27 \cdot \dot{u}^2$$

where the first sum is taken over all active update operations op.

By definition, $lose_i(op)$ is never negative. Moreover, at any one time, at most three Nodes might be flagged by an Info object created by an active operation and this could contribute $-3\dot{u}$ to each $\phi_{flag}(v)$ and hence $-27\dot{u}$ to $3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op))$ and $-27\dot{u}^2$ to Φ_{flag}. The addition of the term $27\dot{u}^2$ ensures that Φ_{flag} is never negative.

The Potential Φ For our analysis, we use the sum of the three potential functions we have defined.

$$\Phi = \Phi_{cursor} + \Phi_{flag} + \Phi_{state}$$

Next, we show what steps in the implementation might change Φ and later we show how those steps actually change Φ.

Lemma 8.2. *Let I be an Info object and oldInfo be an array of Info objects. Only the steps listed in Table 8.1 might change Φ.*

Proof. A Node x becomes unreachable only after the first forward CAS of an Info object I such that $x = I.nodes[1]$ (by Lemma 7.28.5b). By Lemma 7.23, $x.state$ is changed from ordinary to marked or copied on line 110 or 113 inside a call to HELP(I) before the first forward CAS of I and $x.state$ is not changed after that. By Lemma 7.28.7b, x is reachable when its *state* is changed from ordinary to copied or marked. So, changing the *state* of a Node from ordinary to copied or marked does not change the *realNode* of any Node or the *length* of any Node.

By observation of the definitions of the potential functions, the only steps that can affect the potential are those listed in the statement of the lemma. □

By Lemma 8.2, we have the following corollary.

Corollary 8.3. *Excluding the steps inside* UPDATECURSOR*, the following steps do not change Φ: (1) steps that belong to the last attempt of an update operation*

Step	Φ_{cursor}	Φ_{flag}	Φ_{state}
invocation of an update operation	-	✓	✓
reading inProgress on line 93	-	✓	-
reading marked or copied on line 97	-	-	✓
reading value on line 99 that differs from $oldInfo[i]$	-	✓	-
failure of the first flag CAS of I on $I.nodes[i]$	-	✓	-
successful flag CAS on line 106	-	✓	-
changing the *state* of a Node on line 110 or 113	-	-	✓
successful forward CAS on line 114	✓	✓	✓
successful backward CAS on line 115	-	✓	✓
changing *info.status* from inProgress to committed or aborted on line 116 or 118	-	✓	-
termination of an update operation	-	✓	✓
creation of a new Cursor or destroying a Cursor	✓	-	-
writing to $c.node$ for some Cursor c	✓	✓	-
linearization point of a MoveRight(c) or MoveLeft(c) that updates $c.node$ on line 54, 58, 59 or 65	✓	-	-

Table 8.1: The steps that might change Φ

that returns invalidCursor on line 26 or 40, and (2) steps that belong to a move operation that returns invalidCursor or false on line 51, 52, 57, 63 or 64.

8.1.2 Changes to Φ by Steps within UpdateCursor

The following table shows the changes to the potential function by steps within UpdateCursor. ($\Delta\Phi_x$ shows the changes to Φ_x by a call to UpdateCursor.) The next lemma proves the table is correct.

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 85 and 88	-1	0	0

Table 8.2: Changes to Φ by steps wihtin UpdateCursor (Lemma 8.4)

Lemma 8.4. *Steps within each complete iteration of line 82–88 inside* UpdateCursor *decrease* Φ *by 1.*

Proof. Consider an iteration *itr* of loop 82–88 inside a call to UpdateCursor(c). By Lemma 8.2, only the execution of line 85 or 88 of *itr* might change Φ_{cursor} and Φ_{flag} and none can change Φ_{state}. Next, we show that execution of one of those lines decreases $length(c.node)$ by 1 and does not change $realNode(c.node)$.

If the UpdateCursor is called by a move operation that sets the *node* of its Cursor later on line 54, 58, 59 or 65, the move is linearized after the last iteration

of loop 82–88 (by Lemma 7.39 and 7.41). So, $\phi_{cursor}(c) = length(c.node)$ at any configuration during itr. Let x be the value of $c.node$ at the beginning of itr. By Lemma 7.28.13, x was reachable at some configuration before itr. Since the test $(x.prv.nxt \neq x)$ evaluates to true at the beginning of itr, x is not reachable at the configuration after line 82 during itr (by Corollary 7.36). By Lemma 7.32, x is unreachable at all configurations after that line. By Lemma 7.28.5b, x becomes unreachable by the first forward CAS of an Info object I such that $x = I.nodes[1]$. So, $x.state$ is set to marked or copied on line 110 or 113 inside a call to Help(I) before the first forward CAS of I (which is before the line 82 of itr). By Lemma 7.23, $x.state$ is not changed from marked or copied to another value after $x.state$ is set for the first time inside a call to Help(I). So, either line 85 or 88 is executed during itr.

Since, during itr, x is unreachable and $x.state$ is either marked or copied, by definition of $length(c.node)$, $length(c.node)$ decreases by 1 when $c.node$ is set on line 85 or 88, which decreases Φ_{cursor} by 1. By definition, $realNode(c.node)$ is not changed by any step of itr, so Φ_{flag} is not changed by any step of itr. □

8.1.3 Changes to Φ by Steps Belonging to Move Operations

Here, we show that the amortized complexity of each move operation is $O(1)$.

Lemma 8.5. *No step performed by a* MoveLeft*(c) or* MoveRight*(c) operation*

(excluding the call to UpdateCursor*) changes* Φ.

Proof. If the move operation returns invalidCursor or false on line 51, 52, 57, 63 or 64, the claim follows by Corollary 8.3. If the process running the move operation crashes before setting *c.node* on line 54, 58, 59 or 65, no step that belongs to the move operation (excluding the call to UpdateCursor) changes Φ (by Lemma 8.2).

Otherwise, *c.node* is set on line 54, 58, 59 or 65. By Lemma 8.2, among steps belonging to the move, only the linearization point of the move and setting *c.node* on line 54, 58, 59 or 65 might change Φ_{cursor}. (Since Φ_{flag} is defined by a sum only over update operations, setting *c.node* does not change Φ_{flag} and none of the steps affect Φ_{state}.) We show that these two steps do not, in fact, change Φ_{cursor}. Let C be the configuration that is defined by Lemma 7.39 or 7.41 and C' be the configuration after *c.node* is set on line 54, 58, 59 or 65. Note that the move operation is linearized at the null step after C. Let u be *c.node* at C'. By definition, $\phi_{cursor}(c) = length(u)$ in the configuration before C' and $\phi_{cursor}(c) = length(c.node)$ at C'. Since *c.node* is set to u by the step before C', setting *c.node* does not change $\phi_{cursor}(c)$.

Let v be *c.node* at C. By Lemma 7.39 and 7.41, v and u are reachable at C, so $\phi_{cursor}(c) = length(v) = 0$ at C. Since the move is linearized at the null step after C, u is also reachable at the configuration after the linearization point of the move, so $\phi_{cursor}(c) = length(u) = 0$ at that configuration. Thus, the linearization point of the move does not change Φ_{cursor}. □

By Lemma 8.4, the steps within each iteration of loop 82–88 inside a move's call to UpdateCursor decrease Φ by 1. So, by Lemma 8.5, we have the following theorem.

Theorem 8.6. *The amortized complexity of each move operation is at most* 1.

8.1.4 Changes to Φ by Steps that Belong to Update Operations

Here, we show that the amortized cost of each update operation *op* is $O(\dot{c}(op))$. Recall that a complete attempt *att* is unresolved if *att*'s call to CheckInfo on line 29 or 43 or *att*'s call to Help on line 34 or 46 returns false. Next, we show that each unresolved attempt of an update operation decreases Φ. By Lemma 8.2, we have the following corollaries.

Corollary 8.7. *Each unresolved attempt (excluding the call to* UpdateCursor*) whose call to* CheckInfo *returns false on line 29 or 43 has exactly one step (on line 93, 97 or 99) that could change* Φ.

For an Info object I, consider an unresolved attempt of an update operation whose call to Help(I) on line 34 or 46 returns false. Then, $I.status$ is aborted at line 119 inside that call to Help(I). By Lemma 7.17, all calls to Help(I) set the *doPtrCAS* variable to false on an execution of line 107. So, no call to Help(I) executes line 110–116.

Corollary 8.8. *Consider an unresolved attempt whose call to* Help *returns false on line 34 or 46. The only steps belonging to the attempt (excluding the call to* UpdateCursor*) that could change* Φ *are line 106 and 118.*

To determine the changes to Φ_{flag} by steps belonging to an update operation op, we need to determine what $node_i(op)$ is, for all $0 \leq i < 3$, at some configurations during op.

Lemma 8.9. *Let op be an update operation and* $\langle y, -, z, -, -, - \rangle$ *be the result of a call to* UpdateCursor *by op and let C be the configuration before the last execution of line 82 inside the call to* UpdateCursor*. Then,* $y = node_1(op)$ *at C and* $z = node_2(op)$ *at some configuration during the call to* UpdateCursor*.*

Proof. Suppose op is called with Cursor c. Since $c.node = y$ at C, the test $(y.prv.nxt \neq y)$ evaluates to false at the last execution of line 82. By Invariant 7.6.10, $y \neq Head$. By Lemma 7.28.13, y was reachable at some configuration before C. So, by Corollary 7.36, y is reachable at C. Then, at C, $node_1(op) = realNode(c.node) = realNode(y) = y$.

Next, we show that $z = node_2(op)$ at some configuration during the UpdateCursor. If $y = node_1(op)$ when op reads the nxt field of y on line 90, $z = node_2(op)$ at the configuration before that read. Otherwise, $node_1(op)$ is changed from y to another Node between C and reading the nxt field of y on line 90. Let C_i be the

configuration after $node_1(op)$ is changed from y to another Node. By Lemma 7.44, y becomes unreachable at C_i by a successful forward CAS of an Info object I. By Lemma 7.28.5b, $I.nodes[1] = y$. By Lemma 7.28.7a and 7.28.7c, $y.nxt = I.nodes[2]$ at C_{i-1}. By Lemma 7.28.11, $y.nxt = I.nodes[2]$ at all configurations after C_{i-1}. Since $y.nxt = z$ when op reads the nxt field of y on line 90 (after C_i), $z = I.nodes[2]$. Since $node_1(op) = y$ at C_{i-1} and $y.nxt = z$ at C_{i-1}, $node_2(op) = z$ at C_{i-1}. □

Lemma 8.10. *Let op be an update operation and $\langle y, -, -, x, -, - \rangle$ be the result of a call to* UpdateCursor *by op. Then, either*

1. *there is a configuration during the call to* UpdateCursor *when $x = node_0(op)$ and $y = node_1(op)$, or*

2. *for some Info object I with $I.nodes[1] = x$ and $I.nodes[2] = y$, a forward CAS of I succeeds before the call to* UpdateCursor *reads $y.prv$ on line 90, but no backward CAS of I occurrs before that read.*

Proof. Let C be the configuration before the last execution of line 82 inside the call to UpdateCursor and C' be the configuration after op reads $y.prv$ on line 90. By Lemma 8.9, $y = node_1(op)$ at C. We consider two cases.

Case 1: $node_1(op)$ is not y at C'. Then, $node_1(op)$ is changed from y to another Node between C and C'. Let C_i be the configuration after $node_1(op)$ is changed from y to another Node. By Lemma 7.44, y becomes unreachable at C_i by a

successful forward CAS of an Info object I'. By Lemma 7.28.5b, $I'.nodes[1] = y$. By Lemma 7.28.7a and 7.28.7c, $y.prv = I'.nodes[0]$ at C_i. By Lemma 7.28.11, $y.prv = I'.nodes[0]$ at all configurations after C_i. Since $y.prv = x$ at C' (after C_i), $x = I'.nodes[0]$. Since $node_1(op) = y$ at C_{i-1} and $x.nxt = y$ at C_{i-1} (by Lemma 7.28.7a and 7.28.7c) and x is reachable at C_{i-1} (by Lemma 7.28.5a), $node_0(op) = x$ at C_{i-1}, so the statement is true.

Case 2: $node_1(op) = y$ at C'. Then, at C', $realNode(c.node) = y$ and y is reachable. Since $c.node = y$ at C, $y \neq Head$ (by Invariant 7.6.10). Since $y.prv = x$ at C' and y is reachable at C', $x.nxt = y$ at C' (by Lemma 7.28.10). If x is reachable at C', $node_0(op) = x$ at C'.

Suppose x is not reachable at C'. By Lemma 7.28.12, x was reachable in a configuration before C'. So, x becomes unreachable by a successful forward CAS of an Info object I and $x = I.nodes[1]$ (by Lemma 7.28.5b). Let C_j be the configuration before this forward CAS of I. We show that $I.nodes[2] = y$. Since $x = I.nodes[1]$, $x.nxt = I.nodes[2]$ at C_j (by Lemma 7.28.7a and 7.28.7c). By Lemma 7.28.11, $x.nxt = I.nodes[2]$ in all configurations after C_j. Since $x.nxt = y$ at C' (after C_j), $y = I.nodes[2]$. If the first backward CAS of I occurs before C', it changes $I.nodes[2].prv$ from $I.nodes[1]$ to another value (by Lemma 7.28.1). Then, by Lemma 7.28.2, $I.nodes[2].prv \neq I.nodes[1]$ at all configurations after the first backward CAS of I. Since $I.nodes[2].prv = I.nodes[1]$ at C', no backward CAS of

I has occurred before C'. □

Consider a call to CheckInfo that is called by an operation *op*. To show that the $lose_i(op) > 0$ when *op* reads inProgress on the $(i+1)$th execution of line 93 or reads a value different from $oldInfo[i]$ on the $(i+1)$th execution of line 99, we prove the following lemma.

Lemma 8.11. *Consider two unresolved attempts att and att′ of an update operation op, where att precedes att′. Let nodes and nodes′ be the local array used as the first argument to* CheckInfo *during att and att′ respectively.*

1. *If no forward or backward CAS succeeds between the beginning of att and att′'s call to* CheckInfo*, then nodes = nodes′.*

2. *If no forward or backward CAS succeeds between the beginning of att and the end of att′, then, for each $i \in \{0, 1, 2\}$, $nodes[i] = node_i(op)$ at all configurations between the beginning of att and the end of att′.*

3. *If no flag CAS on $nodes[i]$ succeeds and no forward or backward CAS succeeds between the beginning of att and the end of att′, then att and att′ cannot both read inProgress on the $(i+1)$th execution of line 93.*

Proof. **Statement 1** By Lemma 8.9, there are two configurations during *att*'s call to UpdateCursor when $node_1(op) = nodes[1]$ and $node_2(op) = nodes[2]$, and there

are two configurations during att''s call to UpdateCursor when $node_1(op) = nodes'[1]$ and $node_2(op) = nodes'[2]$. Let C_k be the configuration before the beginning of att and C_l be the configuration after att''s call to CheckInfo. Since there is no successful forward CAS between C_k and C_l, $node_1(op)$ and $node_2(op)$ are not changed between C_k and C_l (by Lemma 7.44), so $nodes[1] = nodes'[1]$ and $nodes[2] = nodes'[2]$. Since $nodes[1] = nodes'[1]$, $nodes[0]$ and $nodes'[0]$ are read from $nodes[1].prv$ on line 90 during att and att' respectively. Since there is no successful backward CAS between C_k and C_l, the prv field of $nodes[1]$ does not change between C_k and C_l. So, $nodes[0] = nodes'[0]$.

Statement 2 Let C_k be the configuration before the beginning of att and C_j be the configuration after the end of att'. Since there is no successful forward CAS between C_k and C_j, $nodes = nodes'$ (by Statement 1, proved above). By Lemma 8.9, there are two configurations during att's call to UpdateCursor when $node_1(op) = nodes[1]$ and $node_2(op) = nodes[2]$. Since there is no successful forward CAS between C_k and C_j, $node_1(op)$ and $node_2(op)$ are not changed between C_k and C_j (by Lemma 7.44). So, $node_1(op) = nodes[1]$ and $node_2(op) = nodes[2]$ at all configurations between C_k and C_j.

It remains to show that $node_0(op) = nodes[0]$ at all configurations between C_k and C_j. To derive a contradiction, assume $node_0(op) \neq nodes[0]$ at some configuration between C_k and C_j. Since there is no successful forward CAS between C_k

and C_j, $node_0(op)$ is not changed between C_k and C_j. So, $node_0(op) \neq nodes[0]$ at every configuration during att. By Lemma 8.10, there is an Info object I such that $I.nodes[1] = nodes[0]$ and a forward CAS of I succeeds before C_k but no backward CAS of I occurs before C_j. So, $nodes[0].info = I$ at all configurations after C_k (by Lemma 7.25). Let *oldInfo* be the local array used as the second argument to CheckInfo during att. So, att sets $oldInfo[0]$ to I on line 28 or 42. By Lemma 7.21, $I.status$ is inProgress at all configurations between C_k and C_j. So, att reads inProgress from $I.status$ on the first execution of line 93 and then calls Help(I) on line 94. Then, $I.status$ is set to committed or aborted by the end of this call to Help. (The first execution of line 116 or 118 inside any call to Help(I) sets $I.status$ to committed or aborted.) This contradicts the fact that $I.status$ is inProgress at all configurations between C_k and C_j.

Statement 3 Let *oldInfo* and *oldInfo*$'$ be the local array used as the second argument to CheckInfo during att and att' respectively. By Statement 1, proved above, $nodes = nodes'$. So, both $oldInfo[i]$ and $oldInfo'[i]$ are read from the *info* field of $nodes[i]$ on line 28, 42 or 89 during att and att'. Since there is no successful flag CAS on $nodes[i]$ between C_k and C_j, $oldInfo[i] = oldInfo'[i]$. If att reads inProgress from $oldInfo[i].status$ in its $(i+1)$th iteration of line 93, then att calls Help($oldInfo[i]$) on line 94 and $oldInfo[i].status$ is set to committed or aborted by

the end of the call to HELP. By Observation 7.2, $oldInfo[i].status$ is not inProgress at all configurations after that, so att' cannot read inProgress from $oldInfo[i].status$ in its $(i+1)$th iteration of line 93. □

Tables 8.3, 8.4, 8.5, 8.6, 8.7 and 8.8 show the changes to Φ by steps belonging to different scenarios of unresolved attempts of update operations (excluding the calls to UPDATECURSOR). In these tables, $\Delta\Phi_x$ shows the changes to Φ_x by an unresolved attempt att (excluding att's call to UPDATECURSOR). If att creates an Info object I_{att} on line 33 or 45, a line number of the HELP routine listed in one of these tables represents the first execution of that line inside all calls to HELP(I_{att}).

Let $\Delta\Phi(att)$ be the changes in Φ due to steps belonging to att, excluding att's call to UPDATECURSOR. We now show that each completed unresolved attempt att has $\Delta\Phi(att) < 0$.

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 93 reads inProgress	0	-1	0

Table 8.3: CHECKINFO returns false on line 95 (Lemma 8.12)

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 99 reads a different Info object from the *info* field	0	-1	0

Table 8.4: CheckInfo returns false on line 100 (Lemma 8.12)

Lemma 8.12. *If att is an attempt whose call to* CheckInfo *returns false on line 95 or 100,* $\Delta\Phi(att) < 0$.

Proof. Let $i^* \in \{0, 1, 2\}$. Consider some attempt *att* that returns false after reading inProgress on line 93 when $i = i^*$ or reading a value other than $oldInfo[i^*]$ on line 99 when $i = i^*$. Let C be the configuration before this read and let *op* be the update operation that performs attempt *att*. We show that the step following C decrements $lose_{i^*}(op)$. To derive a contradiction, assume $lose_{i^*}(op) = 0$ in C. Then, there are at least two earlier attempts that fail due to the read on line 93 (when $i = i^*$) or line 99 (when $i = i^*$) since $lose_{i^*}(op)$ must have been decremented at least twice before C to get down to the value 0. Let att_1, att_2 be the last two such attempts.

There is no successful forward or backward CAS between the beginning of att_1 and the end of att_2; otherwise, $lose_{i^*}(op)$ would be at least 1 at C since $lose_{i^*}(op)$ is decremented at most twice between the beginning of att_1 and C (once each during att_1 and att_2) and it could not be set to 3 between the beginning of att_1 and C.

Let $nodes_1$ and $nodes_2$ be the local array $nodes$ used as the first argument to CheckInfo during att_1 and att_2 respectively. By Lemma 8.11.1, $nodes_1[i*] = nodes_2[i*]$. Let x be this Node. Since there is no successful forward or backward CAS between the beginning of att_1 and the end of att_2, $node_{i*}(op) = x$ at all configurations between the beginning of att_1 and the end of att_2 (by Lemma 8.11.2).

Since att_1 does not create an Info object, no unsuccessful flag CAS sets $lose_{i^*}(op)$ to 2 during att_1. Moreover, no attempt after the beginning of att_1 and before the beginning of att sets $lose_{i^*}(op)$ to 2, since $lose_{i^*}(op)$ would be decremented at most once by att_2 between setting $lose_{i^*}(op)$ to 2 and C. So, there is no unsuccessful flag CAS step that sets $lose_{i^*}(op)$ to 2 between the beginning of att_1 and C.

Similarly, there is no successful flag CAS on x by a *different* operation between the beginning of att_1 and the end of att_2 (otherwise, $lose_{i^*}(op)$ would be at least 1 at C). So, neither att_1 nor att_2 fail at line 100. Thus, att_1 and att_2 must both fail as a result of their reads at line 93 (with $i = i^*$). This contradicts Lemma 8.11.3.

Thus, $lose_{i^*}(op) > 0$ at C, so the step following C decrements $lose_{i^*}(op)$ and hence decreases Φ_{flag} by 1. By Corollary 8.7, this is the only step of att (excluding the call to UpdateCursor) that changes Φ. Thus, $\Delta\Phi(att) < 0$. □

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 97 reads copied or marked	0	0	-1

Table 8.5: CheckInfo returns false on line 97 (Lemma 8.13)

Lemma 8.13. *If att is an attempt whose call to* CheckInfo *returns false on line 97,* $\Delta\Phi(att) < 0$.

Proof. Consider some attempt *att* of an update operation *op* whose call to CheckInfo returns false after reading copied or marked on line 97. Let C be the configuration before this read. We show that the step following C decrements $\phi_{state}(op)$. To derive a contradiction, assume $\phi_{state}(op) = 0$ in C. Then, there are at least two earlier attempts that fail due to line 97 since $\phi_{state}(op)$ must have been decremented at least twice before C to get down to the value 0. Let att' be the last such attempt and C' be the configuration just before att' begins. Then, there is no successful forward or backward CAS between C' and C; otherwise, $\phi_{state}(op)$ would be at least 1 at C since $\phi_{state}(op)$ is decremented only once between C' and C. Similarly, no step changes the *state* of a Node between C' and C. Let *nodes* and *nodes*$'$ be the local array used as the first argument to CheckInfo during *att* and att' respectively. By Lemma 8.11.1, $nodes = nodes'$. Since the *state* of no Node is set between C' and C, for some $i^* \in \{0, 1, 2\}$, both att' and *att* reads copied or marked from $nodes[i^*].state$

on line 97. Let x be this Node and I be the Info object such that $x.state$ is set to copied or marked for the first time inside HELP(I), Then, $x = I.nodes[1]$.

By Lemma 7.21, $x.info$ is set to I before $x.state$ is set to marked or copied for the first time and I is a successful Info object. By Lemma 7.17, $I.status$ is never set to aborted. Since $x.info$ is not changed from I to another value when $I.status$ is inProgress (by Lemma 7.15), $x.info = I$ at all configurations after setting $x.state$, but not after the first forward CAS of I. Moreover, $x.info$ would remain equal to I forever after the first forward CAS of I (By Lemma 7.25). Thus, $x.info = I$ at all configurations between C' and C. Since both attempts att' and att set $oldInfo[i^*]$ to $x.info$ before the call to CHECKINFO on line 29 or 43, they both set $oldInfo[i^*]$ to the Info object I. Since att' does not read inProgress from $I.status$ on line 93, $I.status$ is changed from inProgress to committed before that. The first forward CAS of I occurs before $I.status$ is set to committed for the first time. Since there is no successful forward CAS between C' and C, the first forward CAS of I occurs before C'. By Lemma 7.28.5a and 7.32, x is not reachable at all configurations after the first forward CAS of I. So, x is not reachable at any configuration during att. By Lemma 8.9, $i^* = 0$. By Lemma 8.10, no backward CAS of I occurred before att reads the prv field on line 90. Since the first backward CAS of I succeeds (by Lemma 7.28.1), there is no backward CAS of I between C' and C. So, $I.status$ is set to committed before any backward CAS of I occurs, which is a contradiction.

Thus, $\phi_{state}(op) > 0$ at C, so the step following C decrements $\phi_{state}(op)$ and hence decreases Φ_{state} by 1. By Corollary 8.7, this is the only step of att that changes Φ. Thus, $\Delta\Phi(att) < 0$. □

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 106 fails to flag $I_{att}.nodes[0]$	0	-1	0
line 118 sets $I_{att}.status$ to aborted	0	0	0

Table 8.6: The attempt att fails because it fails to flag $I_{att}.nodes[0]$ (Lemma 8.14)

Lemma 8.14. *Let I_{att} be an Info object created by an attempt att of an update operation op. If the first flag CAS of I_{att} on $I_{att}.nodes[0]$ fails, $\Delta\Phi(att) < 0$.*

Proof. By Lemma 8.2, the only steps belonging to att (excluding the call to UpdateCursor) that could change Φ are line 106 and 118 and they only affect Φ_{flag}. Let C be the configuration before the first flag CAS of I_{att} on $I_{att}.nodes[0]$ fails. We show that $lose_0(op)$ is 3 at C. If there is a successful forward or backward CAS during att before C, $lose_0(op) = 3$ at C. Suppose there is no successful forward or backward CAS during att before C. Let x be $I_{att}.nodes[0]$. Since the first flag CAS of I_{att} on x fails, some other operation flags x between when att reads $x.info$ on line 28 or 42 and C. Let C' be the configuration after one such successful flag CAS on x. If $node_0(op) = x$ at C', $lose_0(op) = 3$ at C' and hence at C. Suppose $node_0(op) \neq x$

at C'. Since there is no forward CAS during att, $node_0(op)$ is not changed during att and $node_0(op) \neq x$ at all configurations during att. By Lemma 8.10, for some Info object I with $I.nodes[1] = x$, the first forward CAS of I occurs before att reads the prv field on line 90. So, $x.state$ is set to marked or copied before att reads the prv field on line 90. By Observation 7.3, $x.state$ is marked or copied when att reads $x.state$ on line 97. So, att's call to CheckInfo on line 29 or 43 returns false, contradicting the fact that att creates I_{att} on line 33 or 45.

Thus, $lose_0(op) = 3$ at C, so the step following C decreases $lose_0(op)$ by 1 and hence decreases Φ_{flag} by 1. Since att's call to Help(I_{att}) returns false, $I_{att}.status$ is set to aborted on line 118 and no call to Help(I_{att}) can set $I_{att}.status$ to committed on line 116 (by Corollary 7.18). Since the $info$ field of no Node is ever set to I_{att}, the step that changes $I_{att}.status$ from inProgress to aborted on line 118 does not change Φ. So, by Corollary 8.8, the first flag CAS of I_{att} on x is the only step belonging to att that changes Φ. Thus, $\Delta\Phi(att) < 0$. □

Consider an attempt att that creates an Info object I on line 33 or 45 and sets $I.nodes[0].info$ to I successfully. Lemma 8.16, below, shows what the value of $node_i(op)$ is, for some i, at some configurations during att. Then, Lemma 8.16 will be used to analyze the changes to Φ_{flag} by steps belonging to att. (Recall that I is successful if, for each $0 \leq i \leq 2$, the first flag CAS of I on $I.nodes[i]$ succeeds.) The following lemma is used to prove Lemma 8.16.

Lemma 8.15. *Suppose an attempt att creates an Info object I_{att} on line 33 or 45 and the first flag CAS of I_{att} on $I_{att}.nodes[0]$ succeeds. Let i^* be the greatest index such that the first flag CAS of I_{att} on $I_{att}.nodes[i^*]$ succeeds. Let C be a configuration after the last step of att's call to* UpdateCursor*, but not after an execution of line 118 or a forward CAS of I_{att}. Then,*

1. *For each $0 \leq k \leq i^*$, $I_{att}.nodes[k]$ is reachable in C.*

2. *For each $0 \leq k \leq \min(i^*, 1)$, $I_{att}.nodes[k].nxt$ is $I_{att}.nodes[k+1]$ in C.*

Proof. **Statement 1** Suppose $0 \leq k \leq i^*$. Let y be $I_{att}.nodes[k]$. Then, the first flag CAS of I_{att} on y succeeds. If I_{att} is not a successful Info object, no step that belongs to *att* executes line 110–116 inside Help(I_{att}). Since there is no execution of line 118 before C, $I_{att}.status$ is inProgress in C. Otherwise, I_{att} is a successful Info object, so that $I_{att}.status$ is never set to aborted (by Lemma 7.17). Since there is no forward CAS of I_{att} before C, $I_{att}.status$ is not set to committed before C. So, in either case, $I_{att}.status$ is inProgress in C. Since $I_{att}.status$ is inProgress at all configurations before C (by Observation 7.2), $y.info$ is still equal to I_{att} in C (by Lemma 7.15). Let I be an Info object other than I_{att} with $I.nodes[1] = y$. If the first forward CAS of I occurs before C, $y.info = I$ at C (by Lemma 7.25). So, no forward CAS of I succeeds before C. By Lemma 7.28.12, y was reachable at some configuration before C. So, y is still reachable in C (by Lemma 7.28.5b).

Statement 2 First, we show that $I_{att}.nodes[0].nxt = I_{att}.nodes[1]$ at some configuration during att's call to UpdateCursor. By Lemma 7.28.13, $I_{att}.nodes[1]$ was reachable in the configuration before att's last execution of line 82. Since att's call to CheckInfo on line 29 or 43 returns true, $I_{att}.nodes[1].state$ is ordinary at an execution of line 97 inside att's call to CheckInfo. Since $I_{att}.nodes[1].state$ is not set to marked or copied before that (by Observation 7.3), $I_{att}.nodes[1]$ does not become unreachable before att's call to CheckInfo on line 29 or 43 (by Lemma 7.28.6). So, $I_{att}.nodes[1]$ is still reachable when att reads the prv field of that Node on line 90. By Invariant 7.6.10, $I_{att}.nodes[1] \neq Head$. Since $I_{att}.nodes[1].prv = I_{att}.nodes[0]$ when att reads the prv field of $I_{att}.nodes[1]$ on line 90, $I_{att}.nodes[0].nxt = I_{att}.nodes[1]$ when that read occurs (by Lemma 7.28.10). Since $I_{att}.nodes[1].nxt = I_{att}.nodes[2]$ when att reads the nxt field of $I_{att}.nodes[1]$ on line 90, for each $0 \leq k \leq \min(i^*, 1)$, $I_{att}.nodes[k].nxt = I_{att}.nodes[k+1]$ at some configuration during att's call to UpdateCursor.

Let x and x' be $I_{att}.nodes[k]$ and $I_{att}.nodes[k+1]$ respectively. To derive a contradiction, suppose a forward CAS of an Info object $I' \neq I_{att}$ changes $x.nxt$ from x' to another value before C. Since $x'.state$ is set to copied or marked before the forward CAS of I', $x'.state$ is not ordinary at all configurations after the forward CAS of I' (by Observation 7.3). Since $x'.state$ is ordinary when att reads $x'.state$ on line 97, the forward CAS of I' occurs after att reads $x'.state$ on line 97. Let

old be the Info object that *att* reads from $x.info$ on line 28, 42 or 89. Next, we show that $old = I'$. Since the first flag CAS of I_{att} on x succeeds, by Lemma 7.13, $x.info = old$ at all configurations between when *att* reads $x.info$ on line 28, 42 or 89 and the first flag CAS of I_{att} on x. By Lemma 7.16, $x.info = I_{att}$ at all configurations after the first flag CAS of I_{att} on x, but not after $I_{att}.status$ is changed from inProgress to another value. We showed above that $I_{att}.status$ is inProgress at C. So, $I_{att}.status$ is inProgress at all configurations before C (by Observation 7.2) and $x.info$ is not changed from I_{att} to another value before C. So, $x.info$ is *old* or I_{att} at all configurations between when *att* reads $x.info$ on line 28, 42 or 89 and C. Since the forward CAS of I' succeeds during that period and $x.info = I' \neq I_{att}$ at the configuration before the forward CAS of I' (by Lemma 7.21), $I' = old$. Since *att* does not read inProgress at any execution of line 93, $old.status$ is not inProgress at all configurations after *att* reads $old.status$ on line 93 (by Observation 7.2). Since the forward CAS of I' occurs after *att*'s line 97, $I'.status$ is not inProgress at the configuration before the forward CAS of I', contradicting Lemma 7.20. □

Lemma 8.16. *Suppose an attempt att creates an Info object I_{att} on line 33 or 45 and the first flag CAS of I_{att} on $I_{att}.nodes[0]$ succeeds. Let i^* be the greatest index such that the first flag CAS of I_{att} on $I_{att}.nodes[i^*]$ succeeds.*

Let C be a configuration after the last step of att's call to UpdateCursor*, but not after an execution of line 118 or a forward CAS of I_{att}. Then, for all*

$0 \leq k \leq \min(i^* + 1, 2)$, $I_{att}.nodes[k] = node_k(op)$ *in* C.

Proof. Let U be *att*'s call to UpdateCursor on line 25 or 39. Since the first flag CAS of I_{att} on $I_{att}.nodes[0]$ succeeds, $I_{att}.nodes[0]$ is reachable in C (by Lemma 8.15.1) and $I_{att}.nodes[0].nxt = I_{att}.nodes[1]$ in C (by Lemma 8.15.2). So, $I_{att}.nodes[1]$ is reachable in C. By Lemma 8.9, $node_1(op) = I_{att}.nodes[1]$ at some configuration during U. If $node_1(op)$ is changed from $I_{att}.nodes[1]$ to another value, $I_{att}.nodes[1]$ is unreachable at all configurations after that (by Lemma 7.32). Since $I_{att}.nodes[1]$ is reachable at C, $node_1(op)$ is still equal to $I_{att}.nodes[1]$ in C. Thus, the claim holds for $k = 1$.

Since $I_{att}.nodes[0]$ is reachable in C and $I_{att}.nodes[0].nxt$ is $I_{att}.nodes[1]$ in C and $node_1(op)$ is $I_{att}.nodes[1]$ in C, $node_0(op) = I_{att}.nodes[0]$ in C. Thus, the claim holds for $k = 0$.

If $i^* \geq 1$, then the first flag CAS of I_{att} on $I_{att}.nodes[1]$ succeeds and, by Lemma 8.15.2, $I_{att}.nodes[1].nxt = I.nodes[2]$ in C. Since $I_{att}.nodes[1].nxt$ is $I_{att}.nodes[2]$ in C and $node_1(op)$ is $I_{att}.nodes[1]$ in C, $node_2(op) = I_{att}.nodes[2]$ in C. □

Lemma 8.17. *If s is a successful flag CAS, then s decreases Φ by at least 3.*

Proof. By Lemma 8.2, s can only change Φ_{flag}. Let *att* be the attempt of an update operation *op* that s belongs to and I_{att} be the Info object that *att* creates on line 33 or 45. By Lemma 7.14, s is the first flag CAS of I_{att} on $I_{att}.nodes[i^*]$ for some

$i^* \in \{0, 1, 2\}$. Let x be $I_{att}.nodes[i^*]$. By Lemma 7.28.5b, x becomes unreachable only after a successful forward CAS of an Info object I with $I.nodes[1] = x$ and, by Lemma 7.25, $x.info$ cannot be changed after the forward CAS of I. Since x was reachable earlier than s (by Lemma 7.28.12) and s changes $x.info$, x is still reachable when s occurs. Let u and v be Nodes. As a result of s, the (abort(u) - flag(u)) term

- decreases by 1 for $u = x$ (since $x.info.status \neq$ inProgress at the configuration before s by Lemma 7.15)
- either stays the same or increases by 1 if $u.nxt = x$
- stays the same for all other u.

So, as a result of s, $\phi_{flag}(v)$

- decreases by 1 for $v = x$
- either stays the same or decreases by 1 if v is before x in the list
- stays the same for v strictly after x in the list.

By Lemma 8.16, $I_{att}.nodes[i^*] = node_{i^*}(op)$ at the configuration before s. So, as a result of s, $\phi_{flag}(node_{i^*}(op))$ decreases by 1 and $\sum_{i=0}^{2} \phi_{flag}(node_i(op))$ decreases by at least 1 (which contributes at least -3 to Φ_{flag}) and $lose_i(op)$ is not affected.

For $op' \neq op$, $lose_i(op')$

- increases by ≤ 3 if $node_i(op') = x$,
- does not change if $node_i(op') \neq x$.

Then, $\phi_{flag}(node_i(op'))$

- decreases by 1 if $node_i(op') = x$,
- does not increase if $node_i(op') \neq x$.

So, the sum $3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op')) + \sum_{i=0}^{2} lose_i(op')$ does not increase. Thus, s decreases Φ_{flag} by at least 3. □

Lemma 8.18. *Let I_{att} be an Info object created by an attempt att of an update operation op. If, for $i^* \in \{1, 2\}$, the first flag CAS of I_{att} on $I_{att}.nodes[i^*]$ fails, $\Delta\Phi(att) < 0$.*

Proof. By Lemma 8.2, the only steps belonging to *att* (excluding the call to UpdateCursor) that can change Φ are line 106 and 118 and these steps can only change Φ_{flag}. Since no flag CAS of I_{att} on $I_{att}.nodes[i^*]$ succeeds (by Lemma 7.14), no invocation of Help(I_{att}) executes line 110–116. So, *att*'s first execution of line 118 changes $I_{att}.status$ from inProgress to aborted. So, only the first execution of line 118 belonging to I_{att} and the first flag CAS of I_{att} on each Node in $I_{att}.nodes$ can change Φ_{flag}. We show how these steps change Φ.

Line 106 succeeds Since the first flag CAS of I_{att} on $I_{att}.nodes[i^*]$ occurs, for each $0 \leq i < i^*$, the first flag CAS of I_{att} on $I_{att}.nodes[i]$ succeeds. By Lemma 8.17 and definition of Φ, a successful flag CAS decreases Φ_{flag} by at least 3.

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 106 flags $I_{att}.nodes[0]$	0	≤ -3	0
line 106 fails to flag $I_{att}.nodes[1]$	0	-1	0
line 118 sets $I_{att}.status$ to aborted	0	0	0

Table 8.7: The attempt *att* fails because it fails to flag $I_{att}.nodes[1]$ (Lemma 8.18)

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 106 flags $I_{att}.nodes[0]$	0	≤ -3	0
line 106 flags $I_{att}.nodes[1]$	0	≤ -3	0
line 106 fails to flag $I_{att}.nodes[2]$	0	-1	0
line 118 sets $I_{att}.status$ to aborted	0	0	0

Table 8.8: The attempt *att* fails because it fails to flag $I_{att}.nodes[2]$ (Lemma 8.18)

Line 106 fails to flag $I_{att}.nodes[i^*]$ Let y be $I_{att}.nodes[i^*]$ and C_l be the configuration after the first flag CAS of I_{att} on y fails. Then, $lose_{i^*}(op)$ is set to 2 at C_l. We show that $lose_{i^*}(op)$ is 3 at C_{l-1}. Since the first flag CAS of I_{att} on y fails, some other operation changes $y.info$ to an Info object other than I_{att} between when *att* reads $y.info$ on line 99 and C_{l-1}. Let C_m be the configuration after the first time that y is flagged between when *att* reads $y.info$ on line 99 and C_l. By Lemma 8.16, $y = node_{i^*}(op)$ at C_m, so $lose_{i^*}(op) = 3$ at C_m. For each $0 \leq i < i^*$, $I_{att}.status$ is

inProgress when the first flag CAS of I_{att} on $I_{att}.nodes[i]$ succeeds (by Lemma 7.16). Since the first execution of line 118 inside any call to Help(I_{att}) changes $I_{att}.status$ from inProgress to aborted, $I_{att}.status$ is still inProgress when the first flag CAS of I_{att} on y fails. Since att's call to Help(I_{att}) on line 34 or 46 does not return true, so before att's call to Help(I_{att}) terminates, $I_{att}.status$ is set to aborted. Thus, the first flag CAS of I_{att} on y occurs during att, so C_m and C_l are also during att. So, $lose_{i^*}(op)$ is not decremented between C_m and C_l and $lose_{i^*}(op)$ is still 3 at C_{l-1}. Thus, $lose_{i^*}(op)$ decreases by 1 at C_l.

Line 118 There might be an execution of line 118 that changes $I_{att}.status$ from inProgress to aborted. If so, let C_k be the configuration after this step. Let $0 \leq i < i^*$ and z be $I_{att}.nodes[i]$. Next, we show that abort(z) is 1 at C_{k-1}. We consider two cases.

Case 1: $i = 0$ and $i^* = 2$. Then, the first flag CAS of I_{att} on $I_{att}.nodes[1]$ succeeds. By Lemma 7.16, $I_{att}.status$ is not set to aborted before the first flag CAS of I_{att} on $I_{att}.nodes[1]$. By Lemma 8.15.2, $z.nxt = I.nodes[1]$ when the first flag CAS of I_{att} on $I_{att}.nodes[1]$ succeeds. By Lemma 8.16, $z = node_0(op)$ when the first flag CAS of I_{att} on $I_{att}.nodes[1]$ succeeds. So, abort(z) is set to 1 by that step. By Lemma 8.16, $node_0(op)$ is not changed from z to another value before C_k. So, the termination of another operation cannot set abort(z) to 0 between the first flag

CAS of I_{att} on $I_{att}.nodes[1]$ and C_k. By Lemma 8.15.2, $z.nxt$ is not changed from $I.nodes[1]$ to another value before C_k. Since, at all configurations between the first flag CAS of I_{att} on $I_{att}.nodes[1]$ and C_k, $I_{att}.status$ is inProgress and $z.info = I_{att}$ (by Lemma 7.15), abort(z) is still 1 at C_{k-1}.

Case 2: $i = i^* - 1$. Then, the first flag CAS of I_{att} on $I_{att}.nodes[i+1]$ fails. So, some other operation flags $I_{att}.nodes[i+1]$ between when att reads $I_{att}.nodes[i+1].info$ on line 99 and the first flag CAS of I_{att} on $I_{att}.nodes[i+1]$. Let C_m be the configuration after the first such flag CAS on $I_{att}.nodes[i+1]$. By Lemma 8.15.2, $z.nxt = I.nodes[i+1]$ at C_m. By Lemma 8.16, $z = node_i(op)$ at C_m. So, abort(z) is set to 1 at C_m. By Lemma 8.16, $node_i(op)$ is not changed from z to another value before C_k. So, the termination of another operation cannot set abort(z) to 0 between C_m and C_k. By Lemma 8.15.2, $z.nxt$ is not changed from $I.nodes[i+1]$ to another value before C_k. Next, we show that $z.info.status$ is not changed from inProgress to committed or aborted between C_m and C_{k-1}. Note C_m is between reading $I_{att}.nodes[i+1].info$ on line 99 and C_k. Let old be the value of $z.info$ that att reads on line 28, 42 or 89. Since the first flag CAS of I_{att} on z succeeds, $z.info$ is old or I_{att} at all configurations between when att reads $z.info$ on line 28, 42 or 89 and C_k (by Lemma 7.13). Since $old.status$ is not inProgress when att reads $old.status$ on line 93, $old.status$ is not changed from inProgress to committed or aborted between C_m and C_k (by Observation 7.2). Since $I_{att}.status$ is not changed

from inProgress to aborted before C_k and abort(z) is set to 1 at C_m, abort(z) is still 1 at C_{k-1}.

The only Nodes whose *info* fields are set to I_{att} are $I_{att}.nodes[i]$ for $0 \leq i < i^*$. Since $I_{att}.status$ is not inProgress for the first time in C_k, $I_{att}.nodes[i].info$ is still I_{att} in C_{k-1} (by Lemma 7.15). For each $0 \leq i < i^*$, both abort($I_{att}.nodes[i]$) and flag($I_{att}.nodes[i]$) are changed from 1 to 0 at C_k. So, the (abort(u) - flag(u)) term does not change for any reachable Node u at C_k. Thus, $\phi_{flag}(v)$ does not change for any reachable Node v at C_k. Hence, Φ_{flag} does not change at C_k.

So, the only steps belonging to *att* that change Φ are the first flag CAS of I_{att} on $I_{att}.nodes[i]$, for each $0 \leq i \leq i^*$. Thus, $\Delta\Phi(att) < 0$. □

Lemma 8.19. *If an attempt att of update operation op is not the last attempt of op,* $\Delta\Phi(att) < 0$.

Proof. So, *att* fails due to the call to CheckInfo on line 29 or 43 or the call to Help on line 34 or 46. If the call to CheckInfo on line 29 or 43 returns false, by Lemma 8.12 and 8.13, $\Delta\Phi(att) < 0$. Suppose the call to Help on line 34 or 46 returns false. Let I_{att} be the Info object that *att* creates on line 33 or 45. Then, *att*'s call to Help(I_{att}) returns false on line 34 or 46. So, $I_{att}.status$ is set to aborted on line 118 before *att*'s call to Help(I_{att}) returns. So, all calls to Help(I_{att}) set *doPtrCAS* to false at an execution of line 107 (by Lemma 7.17). So, the first flag CAS of I_{att} on $I_{att}.nodes[i]$ for some i fails. By Lemma 8.14 and

Step	$\Delta\Phi_{cursor}$	$\Delta\Phi_{flag}$	$\Delta\Phi_{state}$
line 106 flags $I_{att}.nodes[0]$	0	≤ -3	0
line 106 flags $I_{att}.nodes[1]$	0	≤ -3	0
line 106 flags $I_{att}.nodes[2]$	0	≤ -3	0
line 110 or 113 changes $I_{att}.nodes[1].state$ from ordinary to marked or copied	0	0	$\leq 2 \cdot \dot{c}(op)$
the forward CAS of I_{att} on line 114 succeeds	$\leq \dot{c}(op)$	$\leq 54 \cdot \dot{c}(op)$	$\leq 2 \cdot \dot{c}(op)$
the backward CAS of I_{att} on line 115 succeeds	0	$\leq 9 \cdot \dot{c}(op)$	$\leq 2 \cdot \dot{c}(op)$
line 116 changes $I_{att}.status$ from ordinary to committed	0	$\leq 27 \cdot \dot{c}(op)$	0
att's line 35 or 47	-1	0	0

Table 8.9: The attempt *att*'s call to HELP(I_{att}) on line 34 or 46 returns true (Lemma 8.20)

8.18, $\Delta\Phi(att) < 0$. □

Lemma 8.20. *If the attempt att of update operation op is the last attempt of op,* $\Delta\Phi(att) \leq 97 \cdot \dot{c}(op)$.

Proof. Suppose *op* does not crash during *att*. If *att* returns invalidCursor on line 26 or 40, $\Delta\Phi(att) = 0$. If *att* returns false on line 44, since *att*'s call to CHECKINFO returns true, no step belonging to *att* changes Φ, so $\Delta\Phi(att) = 0$. It remains to

consider the case where *att*'s call to HELP on line 34 or 46 returns true. Let I_{att} be the Info object that *att* creates on line 33 or 45. Then, $I_{att}.status$ is set to committed on line 116 before *att*'s call to HELP(I_{att}) on line 34 or 46 returns true. So, the first executions of line 110–116 occur before *att*'s call to HELP returns. By Lemma 8.2, the only steps belonging to *att* that can affect Φ are the executions of line 106 on each Node in $I_{att}.nodes$, line 110 or 113, line 114, 115 and 116 and line 35 or 47. (By Corollary 7.18, no step belonging to *att* executes line 118.) We account for the changes in Φ for each of these steps s separately. (See Table 8.9.)

A successful flag CAS on Line 106 By Lemma 8.17 and definition of Φ, this step decreases Φ_{flag} by at least 3 and does not affect Φ_{cursor} or Φ_{state}.

Line 110 or 113 By Lemma 7.23, only the first execution of line 110 or 113 belonging to *att* changes the *state* of a Node from ordinary to marked or copied. Let s be the first execution of line 110 or 113 that belongs to *att*. By Lemma 8.2, s can only change Φ_{state}. For each update operation op' running when s occurs, s increases $\phi_{state}(op')$ by at most 2, so s increases Φ_{state} by at most $2 \cdot \dot{c}(op)$ (since the number of update operations op' running when s occurs is at most $\dot{c}(op)$).

A successful forward or backward CAS on Line 114 or 115 Let s be the successful forward or backward CAS of I_{att} and op' be an update operation running

when s occurs. By Lemma 7.28.1, s is the first execution of line 114 or 115 inside any call to HELP(I_{att}). Then, s increases $\phi_{state}(op')$ by at most 2 and increases $lose_i(op')$ for each i by at most 3. So, s increases Φ_{state} and $\sum_{op}\sum_{i=0}^{2} lose_i(op)$ by at most $2 \cdot \dot{c}(op)$ and $9 \cdot \dot{c}(op)$ respectively. These are the only changes that can be caused by a backward CAS.

For the remainder of this case, we consider a successful forward CAS s. By Lemma 7.28.5b, $I_{att}.nodes[1]$ is the only Node that becomes unreachable by s. Since $I_{att}.nodes[1].state$ is set to marked or copied on line 110 or 113 before s occurs, $I_{att}.nodes[1].state$ is copied or marked when s occurs (by Observation 7.3). Let c' be an active Cursor when s occurs. Suppose $realNode(c'.node) = I_{att}.nodes[1]$ when s occurs. If I_{att} was created by a DELETE, $I_{att}.nodes[1].nxt = I_{att}.nodes[2]$ when s occurs (by Lemma 7.28.7a and 7.28.7c). By Lemma 7.28.5a, $I_{att}.nodes[2]$ is reachable at the configuration after s. If I_{att} was created by an INSERTBEFORE, $I.nodes[1].copy = I.newPrv$ when s occurs (by Lemma 7.26). By Lemma 7.28.8, $I.newPrv$ is reachable at the configuration after s. So, in either case, s increases $\phi_{cursor}(c')$ by 1. Since s changes the nxt field of $I.nodes[0]$ and $I.nodes[0]$ is reachable in the configuration before s and in the configuration after s (by Lemma 7.28.5a), s does not change ϕ_{cursor} of any other Cursor. Thus, s increases Φ_{cursor} by at most $\dot{c}(op)$.

Next, we show how s changes Φ_{flag}. Let x, y and z be $I_{att}.nodes[0]$, $I_{att}.nodes[1]$

and $I_{att}.nodes[2]$ respectively and n_1 and n_2 be $I_{att}.newNxt$ and $I_{att}.newPrv$ respectively. We will use the following two basic facts.

Fact 1: For any Node v, the sum $\phi_{flag}(v) = \sum_{w \text{ is after } v \text{ in the list}} (\text{abort}(w) - \text{flag}(w))$ loses at most one term (for y by Lemma 7.28.5b) and gains at most 2 terms (for n_1 and n_2 in case of an insertion by Lemma 7.28.8). Moreover, no individual (abort(w) - flag(w)) term increases as a result of s. Since the absolute value of each term is at most 1, s increases the sum $\phi_{flag}(v)$ by at most 3.

Fact 2: In any configuration, if u is reachable and $u.nxt = v$, then $|\phi_{flag}(u) - \phi_{flag}(v)| = |\text{abort}(u) - \text{flag}(u)| \leq 1$.

Let C_m be the configuration after s and v be a Node. Let $\phi(v)$ be the value of $\phi_{flag}(v)$ at C_m and $\phi'(v)$ be the value of $\phi_{flag}(v)$ at C_{m-1}. Suppose op' is an update operation running when s occurs. If $node_1(op') \neq y$ at C_{m-1}, s does not change $node_1(op')$ (by Lemma 7.28.5b), so, by Fact 1, $\Delta\phi_{flag}(node_1(op')) \leq 3$. If $node_1(op') = y$ at C_{m-1}, s changes $node_1(op')$ to z or n_2 (since, at C_m, $y.nxt = z$ by Lemma 7.28.7a and 7.28.7c in case of deletion, and $y.copy = n_2$ in case of insertion by Lemma 7.26). We consider two cases.

Case 1: s changes $node_1(op')$ from y to z. Then,

$$\begin{aligned} \Delta\phi_{flag}(node_1(op')) &= \phi(z) - \phi'(y) \\ &= \phi(z) - \phi'(z) + \phi'(z) - \phi'(y) \leq 4. \end{aligned}$$

(By Fact 1, $\phi(z) - \phi'(z) \leq 3$ and by Fact 2, $\phi'(z) - \phi'(y) \leq 1$ since $y.nxt = z$ at C_{m-1}.)

Case 2: s changes $node_1(op')$ from y to n_2. Then,

$$\begin{aligned}\Delta\phi_{flag}(node_1(op')) &= \phi(n_2) - \phi'(y) \\ &= \phi(n_2) - \phi(z) + \phi(z) - \phi'(z) + \phi'(z) - \phi'(y) \leq 5.\end{aligned}$$

(By Fact 2, $\phi(n_2) - \phi(z) \leq 1$ (since $n_2.nxt = z$ at C_m by Lemma 7.28.3a), by Fact 1, $\phi(z) - \phi'(z) \leq 3$ and by Fact 2, $\phi'(z) - \phi'(y) \leq 1$ since $y.nxt = z$ at C_{m-1}.)

Thus, in all cases, $\phi_{flag}(node_1(op')) \leq 5$.

Next, we show a bound on $\Delta\phi_{flag}(node_2(op'))$ by considering four cases.

Case 1: If $node_1(op') = x$ at C_{m-1}, since s changes $x.nxt$ from y to either z or n_1, and x is reachable at C_m (by Lemma 7.28.5a), it follows that s does not change $node_1(op')$ but changes $node_2(op')$ from y to either z or n_1. As proved above, if s changes $node_2(op')$ from y to z, $\Delta\phi_{flag}(node_2(op')) = \phi(z) - \phi'(y) \leq 4$. Otherwise, s changes $node_2(op')$ from y to n_1. Then,

$$\begin{aligned}\Delta\phi_{flag}(node_2(op')) &= \phi(n_1) - \phi'(y) \\ &= \phi(n_1) - \phi(n_2) + \phi(n_2) - \phi'(y) \leq 6.\end{aligned}$$

(By Fact 2, $\phi(n_1) - \phi(n_2) \leq 1$ (since $n_1.nxt = n_2$ at C_m by Lemma 7.28.3a) and, as proved above, $\phi(n_2) - \phi'(y) \leq 5$.)

Case 2: Suppose $node_1(op') = y$ at C_{m-1} and I_{att} is created by a DELETE operation. Then, y becomes unreachable at C_m (by Lemma 7.28.5a). Since $y.state$ is set to marked before C_m, $y.state$ = marked at C_m (by Lemma 7.23). Since $y.nxt = z$ at C_{m-1} and C_m (by Lemma 7.28.7a and 7.28.7c), s changes $node_1(op')$ to z (since z is reachable at C_m by lemma 7.28.5a) and changes $node_2(op')$ from z to $z.nxt$. Then,

$$\begin{aligned} \Delta\phi_{flag}(node_2(op')) &= \phi(z.nxt) - \phi'(z) \\ &= \phi(z.nxt) - \phi(z) + \phi(z) - \phi'(z) \leq 4. \end{aligned}$$

(By Fact 2, $\phi(z.nxt) - \phi(z) \leq 1$ and by Fact 1, $\phi(z) - \phi'(z) \leq 3$.)

Case 3: Suppose $node_1(op') = y$ at C_{m-1} and I_{att} is created by an INSERT-BEFORE operation. Then, y becomes unreachable at C_m (by Lemma 7.28.5a). Since $y.state$ is set to copied before C_m, $y.state$ = copied at C_m (by Lemma 7.23). Since $y.copy$ is set to n_2 before C_m, $y.copy = n_2$ at C_m (by Lemma 7.26). So, s changes $node_1(op')$ from y to n_2 (since n_2 is reachable at C_m by Lemma 7.28.8). Since $y.nxt = z$ at C_{m-1} (by Lemma 7.28.7a and 7.28.7c) and $n_2.nxt = z$ at C_m (by Lemma 7.28.3a), s does not change $node_2(op')$. So, by Fact 1, $\Delta\phi_{flag}(node_2(op')) \leq 3$.

Case 4: Otherwise, s does not change $node_2(op')$, so, by Fact 1, $\Delta\phi_{flag}(node_2(op')) \leq 3$.

Thus, in all cases, $\Delta\phi_{flag}(node_2(op')) \leq 6$.

Next, we show a bound on $\Delta\phi_{flag}(node_0(op'))$. If $node_1(op') = y$ at C_{m-1} and I_{att} is created by an InsertBefore operation, s changes $node_1(op')$ from y to n_2 as proved above. Since $x.nxt = y$ at C_{m-1} (by Lemma 7.28.7a and 7.28.7c) and x is reachable at C_{m-1} (by Lemma 7.28.5a), $node_0(op') = x$ at C_{m-1}. Since $n_1.nxt = n_2$ at C_m (by Lemma 7.28.3a) and n_1 is reachable at C_m (by Lemma 7.28.8), s changes $node_0(op')$ from x to n_1. Then,

$$\begin{aligned}\Delta\phi_{flag}(node_0(op')) &= \phi(n_1) - \phi'(x) \\ &= \phi(n_1) - \phi(x) + \phi(x) - \phi'(x) \leq 4.\end{aligned}$$

(By Fact 2, $\phi(n_1) - \phi(x) \leq 1$ (since $x.nxt = n_1$ at C_m by Lemma 7.28.1) and by Fact 1, $\phi(x) - \phi'(x) \leq 3$.)

If $node_1(op') = y$ at C_{m-1} and I_{att} is created by a Delete operation, s changes $node_1(op')$ from y to z as proved above. Since $x.nxt = y$ at C_{m-1} (by Lemma 7.28.1) and x is reachable at C_{m-1} (by Lemma 7.28.5a), $node_0(op') = x$ at C_{m-1}. Since s sets $x.nxt$ to z, s does not change $node_0(op')$. So, by Fact 1, $\Delta\phi_{flag}(node_0(op')) \leq 3$.

Otherwise, s does not change $node_0(op')$, so, by Fact 1, $\Delta\phi_{flag}(node_0(op')) \leq 3$.

Thus, the changes to $\sum_{op'} 3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op')) \leq 3 \cdot \dot{c}(op) \cdot (4+5+6) = 45 \cdot \dot{c}(op)$. In addition, s increases $\sum_{op} \sum_{i=0}^{2} lose_i(op)$ by at most $9 \cdot \dot{c}(op)$. The total changes in Φ_{flag} as a result of s is at most $54 \cdot \dot{c}(op)$.

Line 116 By Observation 7.2 and Corollary 7.18, only the first execution of line 116 belongs to att changes $I_{att}.status$ from inProgress to committed. Let s be that step. By Lemma 7.15, the *info* fields of three Nodes in $I_{att}.nodes$ are I_{att} at the configuration before s. So, s decreases flag(u) by one for three Nodes u and cannot increase abort(u). So, $\phi_{flag}(v)$ increases by at most 3 for every Node v. Thus, Φ_{flag} increases by at most $27 \cdot \dot{c}(op)$.

Line 35 or 47 Let s be the execution of line 35 or 47 belongs to att and C_p be the configuration after s occurs. Since $c.node = I_{att}.nodes[1]$ at the last execution of line 82 inside att's call to UpdateCursor, $c.node = I_{att}.nodes[1]$ at C_{p-1}. Since att's call to Help(I_{att}) on line 34 or 46 returns true, $I_{att}.status$ is set to committed on line 116 inside a call to Help(I_{att}) earlier. So, the first executions of line 110–116 inside any call to Help(I_{att}) occur earlier than s. By Lemma 7.28.5a and 7.32, $I_{att}.nodes[1]$ is unreachable at all configurations after the first forward CAS of I_{att}.

If I_{att} is created by an InsertBefore operation, $I_{att}.nodes[1].state$ is copied at all configurations after the first execution of line 113 belongs to att (by Lemma 7.23). Let $yCopy$ be the Node created on att's line 31. Then, $I_{att}.nodes[1].copy$ is $yCopy$ at all configurations after the first execution of line 112 belongs to att (by Lemma 7.26). Since $I_{att}.nodes[1]$ is unreachable at C_{p-1}, $realNode(c.node) = realNode(c.node.copy) = realNode(yCopy)$ at C_{p-1}. Since s sets $c.node$ to $yCopy$,

$length(c.node)$ decreases by 1 at C_p.

If I_{att} is created by a DELETE operation, $I_{att}.nodes[1].state$ is marked at all configurations after the first execution of line 110 belongs to att (by Lemma 7.23). By Lemma 7.28.11, $I_{att}.nodes[1].nxt = I_{att}.nodes[2]$ at all configurations after the successful forward CAS of I_{att}. Since $I_{att}.nodes[1]$ is unreachable at C_{p-1}, $realNode(c.node) = realNode(c.node.nxt) = realNode(I_{att}.nodes[2])$ at C_{p-1}. Since s sets $c.node$ to $I_{att}.nodes[2]$, $length(c.node)$ decreases by 1 at C_p.

Thus, in both cases, s decreases Φ_{cursor} by 1. Since s does not change realNode($c.node$), s does not change Φ_{flag}.

Thus, if att is a complete attempt of op, $\Delta\Phi(att) \leq 97 \cdot \dot{c}(op)$. Otherwise, op crashes while running att. As we showed above and in Lemma 8.12, 8.13, 8.14 and 8.18, the steps belonging to att increase Φ by at most $97 \cdot \dot{c}(op)$. Thus, $\Delta\Phi(att) \leq 97 \cdot \dot{c}(op)$. □

Theorem 8.21. *The amortized complexity of each* INSERTBEFORE *or* DELETE *operation op is at most* $205 \cdot \dot{c}(op)$.

Proof. Let s be the invocation of an update operation. Then, s increases Φ_{state} by at most 2. The invocation of an update operation op adds the term $3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op))$ to Φ_{flag}. Let u be a reachable Node when op is invoked. If abort(u) is set to 1 by a step s' earlier, then, when s' occurs, u was equal to

$node_i(op')$ for some i and some update operation op'. If $node_i(op')$ is changed from u to another value, either u becomes unreachable or $u.nxt$ is changed. If $u.nxt$ is changed, abort(u) is set to 0. If, when op' terminates, $u = node_i(op')$ and there is no other operation op'' such that $node_j(op'') = u$ for some j, then abort(u) is set to 0. Thus, if abort(u) = 1 when op is invoked, then there is an active update operation op' such that $node_i(op') = u$ for some i when op is invoked. So, the number of reachable Nodes whose aborts are 1 at a configuration is at most $3 \cdot \dot{c}(op)$. For each reachable Node v, |abort(v) - flag(v)| ≤ 1 at any configuration. So, when op is invoked, the new term $3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op))$ increases Φ_{flag} by at most $27 \cdot \dot{c}(op)$. When op is invoked, the new term $\sum_{i=0}^{2} lose_i(op)$ which is initially 9 is also added to Φ_{flag}. Since s increases $\dot{u}$ by 1, s also increases Φ_{flag} by $27(\dot{u}^2 - (\dot{u} - 1)^2) = 54 \cdot \dot{u} - 27$. (Since each update is called with a local Cursor, $\dot{u} \leq \dot{c}(op)$ during operation op.) So, the invocation of op increases Φ by at most $2 + 27 \cdot \dot{c}(op) + 9 + 54 \cdot \dot{u} - 27 \leq 81 \cdot \dot{c}(op) - 16$.

When op terminates, it decreases $\dot{u}$ and might set the abort variable of at most three Nodes to 0. This can only decrease Φ_{flag}. In addition, the termination of op removes the term $3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op)) + \sum_{i=0}^{2} lose_i(op)$ from Φ_{flag}. The term $\sum_{i=0}^{2} lose_i(op)$ is non-negative, but the term $3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op))$ can be negative. Since at most three Nodes might be flagged by an Info object created by an active operation and this contributes $-3 \cdot \dot{u}$ to each $\phi_{flag}(v)$ for a Node v, hence $3 \cdot$

$\sum_{i=0}^{2} \phi_{flag}(node_i(op))$ is at least $-27 \cdot \dot{u}$. So, removing the term $3 \cdot \sum_{i=0}^{2} \phi_{flag}(node_i(op))$ from Φ_{flag} can increase Φ_{flag} by at most $27 \cdot \dot{c}(op)$.

By Corollary 8.3 and Lemma 8.4, 8.19 and 8.20, the steps belonging to *op* increases Φ by at most $97 \cdot \dot{c}(op) + 81 \cdot \dot{c}(op) + 27 \cdot \dot{c}(op) = 205 \cdot \dot{c}(op)$ and the steps belonging to each unresolved attempt of *op* and each iteration of line 82–88 of *op* decrease Φ by at least 1. Since, for simplicity, we assume each loop iteration of 82–88 inside UpdateCursor and each attempt of an update operation (excluding UpdateCursor) takes one unit of time, the amortized complexity of *op* is $205 \cdot \dot{c}(op)$. □

8.1.5 Summing Up

We first bound the total number of steps taken during any finite execution of the implementation and, then we prove our doubly-linked list is non-blocking.

Theorem 8.22. *Let n_o be the number of operations other than updates invoked during a finite execution α. Then, the total number of steps taken by all operations during the execution α is at most* $\sum_{op \text{ is an update invoked in } \alpha} O(\dot{c}(op)) + n_o$.

Proof. When InitializeCursor(c) is called, since $c.node$ is initialized to $Head.nxt$ on line 68, $length(c.node) = 0$. When $c.node$ is set to $Head.nxt$ on line 73 inside a ResetCursor operation, $length(c.node) = 0$. So, line 68 and 73 do not increase

Φ. By Lemma 8.4, no step belonging to a GET operation increases Φ. The claim follows from Theorem 8.6 and 8.21. □

Theorem 8.23. *The implementation of a doubly-linked list is non-blocking.*

Proof. To derive a contradiction, assume there is an infinite execution α where only finitely many operations are completed. Then, some operation *op* performs an infinite number of executions of a loop (in UPDATECURSOR, INSERTBEFORE or DELETE). Let n_o be the number of operations other than updates invoked during α and let $T = \sum_{op \text{ is an update invoked in } \alpha} 205 \cdot \dot{c}(op) + n_o$, which is finite. By Theorem 8.21, there are at most a total of T iterations of loops in α and this contradicts the fact that *op* performs infinitely many loop iteration. □

8.2 The Results of Empirical Evaluation of Doubly-linked List

Here, we give a simple empirical evaluation of our implementation on a multicore system to show our implementation is scalable and practical. We evaluated our implementation (NBDLL) on a Sun SPARC Enterprise T5240 with 32GB RAM and two UltraSPARC T2+ processors, each with eight 1.2GHz cores, for a total of 128 hardware threads. The experiments were run in Java. The Sun JVM version 1.7.0_3 was run in server mode. The heap size was set to 4G to ensure that the

garbage collector was invoked regularly, but not too often, so that the measurements reflect the running time of the algorithms themselves. (We found the suitable heap size experimentally by monitoring the number of calls to the garbage collector when the heap size is set to different values.)

The only other handcrafted implementation of doubly-linked list using CAS is the one presented in [44]. However, it has not been proved correct and, as we discovered using the Java PathFinder model checker [50], their implementation has at least minor problems. As Table 3.2 shows, the implementations of doubly-linked lists using 2-CAS do not fully support cursors: the implementation in [19] does not support them at all, and the implementation in [3] only supports update operations, without allowing cursors to be moved. The latter implementation also does not have any scheme to recover the location of a cursor when its item is removed from the list by another process. Since other provably correct list implementations do not support the same functionality for cursors as ours, we focus on testing the scalability of our list rather than comparing to other implementations. We do compare NBDLL to a doubly-linked list using the Java implementation of transactional memory of [30] (STMDLL) that provides the same functionality as NBDLL. Since such a general technique usually has more overhead than a handcrafted implementation, we expected that NBDLL would outperform STMDLL.

We evaluated NBDLL in two different scenarios whose results are shown in

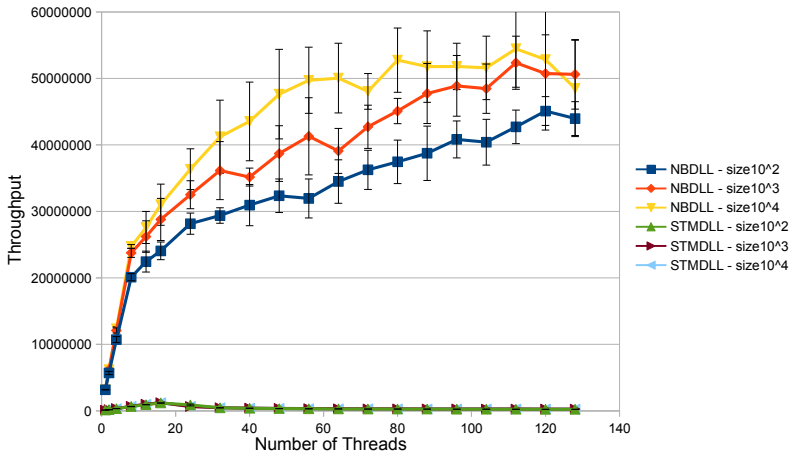


Figure 8.1: Ratio: i5-d5-m90

Figure 8.1 and 8.2 and Figure 8.3. In each graph, the x-axis is the number of threads (from 1 to 128) and each data point in our graphs is the average of fifteen 4-second trials. Since Java optimizes running code, we ran two warm-up trials before each experiment, but the results of these trials are discarded. Error bars in graphs show standard deviations. We increase the number of threads from 1 to 128 to show the scalability of NBDLL. In our experiments, each thread has its own local cursor to call update and move operations with. Each thread's cursor had a random starting location.

In the first scenario, we ran NBDLL and STMDLL with operation ratios of 5% InsertBefores, 5% Deletes and 90% moves (i5-d5-m90) and 30% InsertBefores, 30% Deletes and 40% moves (i30-d30-m40). (See Figure 8.1 and 8.2.) We ran the experiments with three different list sizes: 10^2, 10^3 and 10^4 to measure per-

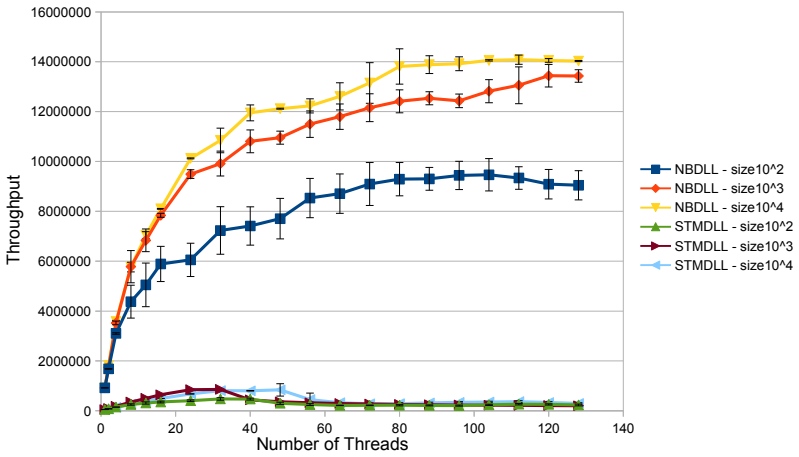


Figure 8.2: Ratio: i30-d30-m40

formance under high, medium and low contention. The y-axis in Figure 8.1 and 8.2 gives throughput (the number of operations terminated per second). To increase the contention consistently when the number of threads are increased, each thread alternated between InsertBefore and Delete operation to keep the number of elements in the list consistent. In addition, we try to keep the distribution of the cursors consistent through the experiments. We chose fractions of MoveLefts and MoveRights so that the cursors remained approximately evenly distributed across the list: According to the sequential specification, when a thread removes an element from the list, the thread moves its cursor to the next element in the list. When there are no other concurrent updates to the list, the deletion does not change the number of elements to the left of the cursor. When a thread inserts an element into the list, when there are no other concurrent updates to the list,

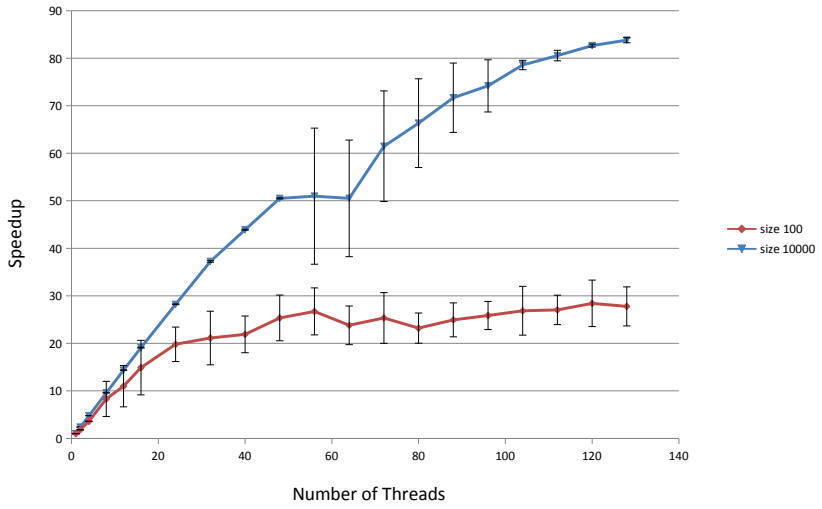


Figure 8.3: Sorted List

the insertion increases the number of elements to the left of the cursor by 1. So, to avoid shifting cursors to the right end of the list by InsertBefore operations, we added some MoveLeft operations to keep the cursors approximately evenly distributed through the list.

Before starting each experiment, to create the initial list with k elements, first, a thread inserts k elements into the list. Then, the same number of threads as the experiment run the same ratio of operations as the experiment until the number of InsertBefore and also Delete operations are equal to or more than k. (This pre-filling ensures that the initial list is not created on only one page of the memory.)

Our results show that NBDLL scales much better than STMDLL. NBDLL scales well for different ratios of operations and scales best for up to 16 threads (since the

machine has 16 cores). For the list with 10^2 elements, throughput scales more slowly since contention becomes very high.

In the second scenario, we implemented a sorted list using update and move operations. (See Fig. 8.3.) Each thread continuously chooses a random key and then tries to insert or delete that key. The thread first searches for the key using move operations and then the thread might perform an update operation (if a DELETE finds the key or if an INSERTBEFORE does not find it is already in the list). The y-axis in Figure 8.3 gives speedup, which is the throughput of key insertions and deletions (each of which consists of many move operations and zero or one update) divided by the throughput when one thread runs by itself. Threads insert or delete random keys from the ranges $[0, 2 \cdot 10^2]$ and $[0, 2 \cdot 10^4]$ and the list is initialized to be half-full. Since the number of move operations called to find the location for insertion and deletion depends on the size of the list, it is not fair to compare the throughput of lists with different sizes. Since speedup compares the number of updates performed by all threads to one thread, we show speedup of lists with different sizes in Figure 8.3 instead of throughput.

To create the initial list with k elements, first, a thread inserts odd values between 0 and $2k$ to the list. Then, the same number of the threads as the experiment try to add or remove a random value less than $2k$ from the list until the number of INSERTBEFORE and also DELETE operations are equal to or more than k.

For shorter lists, less time is required to find the correct location, but contention is high. As our results show, our implementation scales well and longer lists scale better because of lower contention.

9 Patricia Trie Implementation

A Patricia trie stores a set of keys, which are encoded as ℓ-bit binary strings. (See Figure 9.1.) The name Patricia trie is an acronym for Practical Algorithm To Retrieve Information Coded in Alphanumeric, and trie comes from the word reTRIEve. Each node in the trie is either an internal node or a leaf node. The keys are stored as labels of the leaves of the trie. Each internal node has exactly two children. Each internal node has a label that is the longest common prefix of its children's labels. If an internal node's label has length $k-1$, then the kth bit of the node's left and right child's labels are 0 and 1, respectively. Since keys are ℓ-bit binary strings and the path from the root to a key is determined by the sequence of characters in the key, the height of the trie is at most ℓ. Thus, if key strings are short, the height of the trie remains small without requiring any complicated balancing. In addition to the basic operations that insert, delete and find keys stored in the trie, our implementation also provides a replace operation that atomically deletes one key and inserts another.

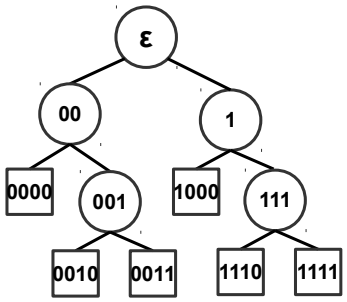


Figure 9.1: An example of a Patricia trie. Leaves are represented by squares and internal nodes are represented by circles.

Patricia tries are widely used in practice. For instance, in data mining, Patricia tries can be used to store items of a database as keys, together with auxiliary information about the keys such as a counter that shows the frequency of the items in the database. We employ our approach to implement a non-blocking linearizable Patricia trie. In this chapter, we describe our Patricia trie implementation.

A Patricia trie can also be used to store a set of points in $\mathbb{R}^d$. For example, a point in $\mathbb{R}^2$ whose coordinates are (x, y) can be represented by a key formed by interleaving the bits of x and y. (This yields a data structure very similar to a quadtree.) The Replace operation can then be used to move a point from one location to another atomically. This operation has applications in Geographic Information Systems [18]. The Replace operation would also be useful if the Patricia trie were adapted to implement a priority queue, so that one could change the priority of an element in the queue.

Our trie is the first non-blocking handcrafted Patricia trie implementation. However, other varieties of non-blocking search tree implementations have been proposed. Searching for a key in our Patricia trie implementation is wait-free, unlike the searches in binary search trees of [14], because the length of a search path in a Patricia trie is bounded by the length of the key. There are several novel features of this work. In [14], modifications were only made at the bottom of the search tree. Our new Patricia trie implementation also copes with modifications that can occur anywhere in the trie. This requires proving that changes in the middle of the trie do not cause concurrent searches passing through the modified nodes to go down the wrong branch. Howley and Jones [27] introduced changes in the middle of a search tree but only to keys stored in internal nodes, not to the structure of the tree itself. Subsequent to our work on Patricia trie, Brown et al. [9] described a framework for making a single change anywhere in a tree. The Replace operation of our Patricia trie implementation makes two changes to the trie atomically using single-word CAS. Cederman and Tsigas [11] proposed a non-blocking Replace operation for a tree-based data structure, but they require 2-CAS, which modifies two non-adjacent locations.

Sequential Specification The Patricia trie stores a set D of keys from a finite universe U. It supports four operations, Find, Insert, Delete and Replace.

The initial state of the set is $\emptyset$. We assume keys are encoded as ℓ-bit binary strings. (At the end of Section 9.2, we shall describe how our implementation can handle unbounded length keys.) Below, we describe the state transitions and responses for each type of operation on a set in state D.

- If $v \in D$, FIND(v) returns true; otherwise, it returns false. In either case, FIND(v) does not change D.

- If $v \notin D$, INSERT(v) changes D to $D \cup \{v\}$ and returns true; otherwise, it returns false.

- If $v \in D$, DELETE(v) changes D to $D - \{v\}$ and returns true; otherwise, it returns false.

- If $v \in D$ and $v' \notin D$, REPLACE(v, v') changes D to $D - \{v\} \cup \{v'\}$ and returns true; otherwise, it returns false.

Update Operations The INSERT and DELETE operations are handled by changing one pointer in the data structure using a CAS step. However, the REPLACE operation changes either one or two pointers of the data structure using CAS steps. Next, we explain how our update operations are handled by some examples.

Consider Trie 1 in Figure 9.2. To remove the key 1100 from the trie, the internal node with label 110 and the leaf node with label 1100 are removed from the trie as shown in Trie 2 of Figure 9.2. So, to perform a DELETE(v) operation, the pointer

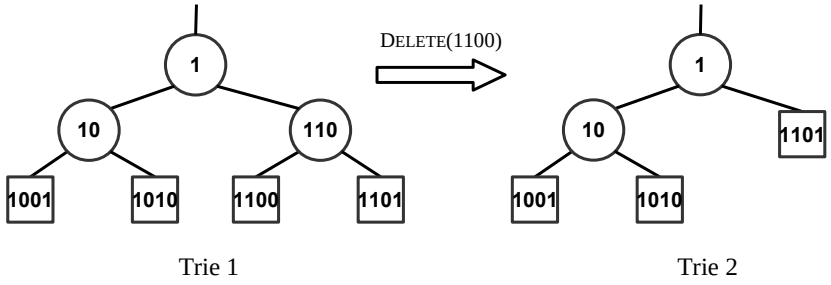


Figure 9.2: Removing the key 1100 from the trie

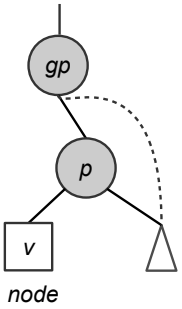


Figure 9.3: DELETE(v) Triangles are either a leaf node or a subtree. The grey circles are flagged nodes. The dotted lines are the new child pointers that replace the old child pointers (solid lines).

of the grandparent of the leaf whose key is v is changed to the sibling of that leaf as shown in Figure 9.3.

Consider Trie 1 in Figure 9.4. To insert the key 010, a new internal node with label 01 must be added between the internal node with label 0 and the leaf node with label 011 as shown in Trie 2 of Figure 9.4. However, the deletion of the key 010 just after the insertion of this key would change the right child of the internal node with label 0 back to the leaf node with label 011 again as in Trie 1 of Figure 9.4. This would cause an ABA problem. To avoid this, the insertion of 010 replaces

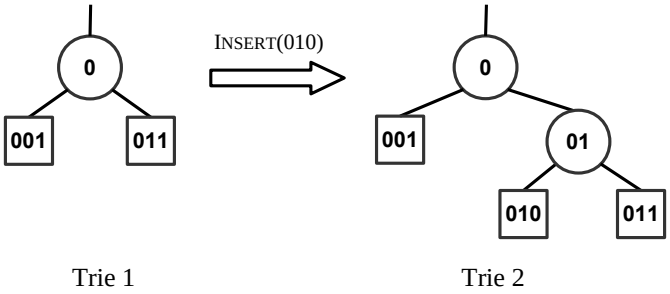


Figure 9.4: Inserting the key 010 into the trie

the leaf node whose label is 011 with a new *copy* of that node. This is analogous to the way inserts in the doubly-linked list replaces one node with a new copy.

The insertion of a key might sometimes change nodes in the middle of the trie. For instance, consider Trie 1 in Figure 9.5. To insert the key 1110, a new internal node with label 11 must be added between the internal nodes with labels 1 and 110 as shown in Trie 2 of Figure 9.5. Since the deletion of the key 1110 might cause an ABA problem, the insertion of 1110 again replaces the internal node whose label is 110 with a new copy.

To add a new key to the trie, there are two different cases as shown in Figure 9.6: To perform an INSERT(v) operation, the operation first searches for a location in the trie to insert the new leaf node with key v. If the INSERT reaches a leaf node, the appropriate child pointer in the parent of the leaf is changed to a new internal node whose child is the new leaf with key v as shown in Case 1 of Figure 9.6. Otherwise, the INSERT reaches an internal node whose label is not a prefix of v

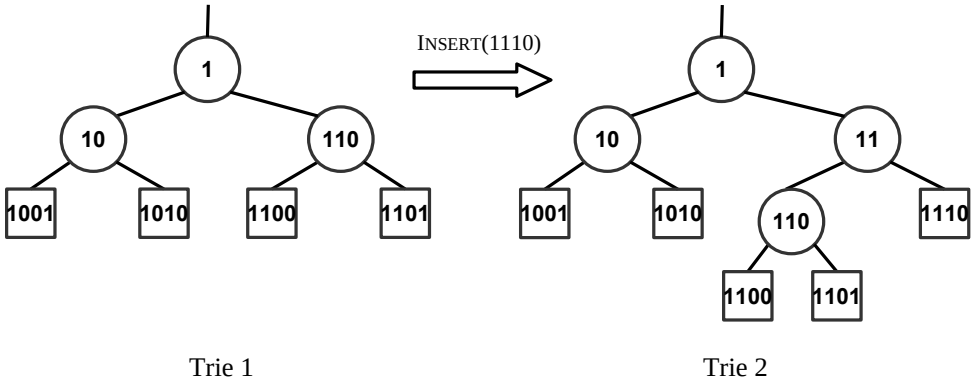


Figure 9.5: Inserting the key 1110 into the trie

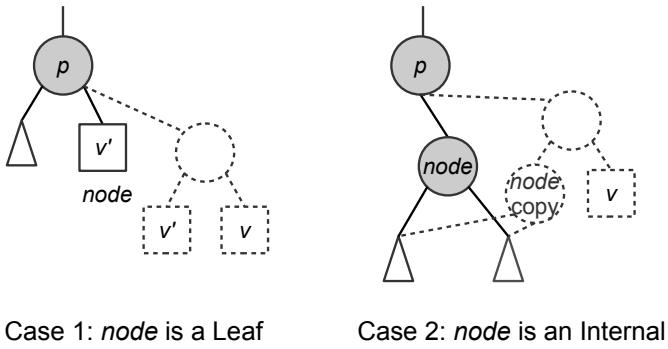


Figure 9.6: Different cases of INSERT(v). The dotted circles and squares are newly created nodes.

during its search. Then, the appropriate child pointer of the parent of the internal node is changed to a new internal node whose child is the new leaf with key v as shown in Case 2 of Figure 9.6. In both cases, the other child of the newly created internal node is a copy of the original child of the parent.

Next, we explain how the REPLACE(v_d, v_i) operation is handled. If INSERT(v_i) and DELETE(v_d), as described in Figure 9.3 and 9.6, would not overlap, REPLACE(v_d, v_i) is done by performing two CAS steps: one for the INSERT(v_i)

and one for the DELETE(v_d). This is called the general case of the REPLACE operation. The general case of REPLACE(v_d, v_i) is linearized at the first CAS step.

Cases when the insertion and deletion would occur in overlapping portions of the trie are handled as special cases. In the special cases, the REPLACE operation changes the trie with only one CAS step. There are four special cases of REPLACE(v_d, v_i) where the changes required by the insertion and deletion are on overlapping portions of the trie. More precisely, a special case occurs when some node would have to be flagged for both changes. The four possible ways this could happen are shown in Figure 9.7. In Figure 9.7, $node_d$, p_d and gp_d are the leaf with key v_d, the parent and the grandparent of that leaf, respectively. Similarly, $node_i$ and p_i are the node that would be replaced to insert v_i and the parent of $node_i$, respectively. Special case 1 occurs when the label of p_d is a prefix of v_i and $node_i$ is the leaf with key v_d. Then, the pointer of p_d is set to the new leaf with key v_i. Special case 2 or 3 occurs when p_d must be replaced by a new internal node whose child is the new leaf with key v_i. Special case 4 occurs when both p_d and gp_d must be replaced by a new internal node whose child is the new leaf with key v_i. We now present an example for each special case of the REPLACE operation.

Consider Trie 1 in Figure 9.8. To perform REPLACE(011, 010), since the label of the parent of the leaf with key 011 is a prefix of 010, a CAS step replaces the leaf with key 011 by a new leaf with key 010 as shown in Trie 2 of Figure 9.8.

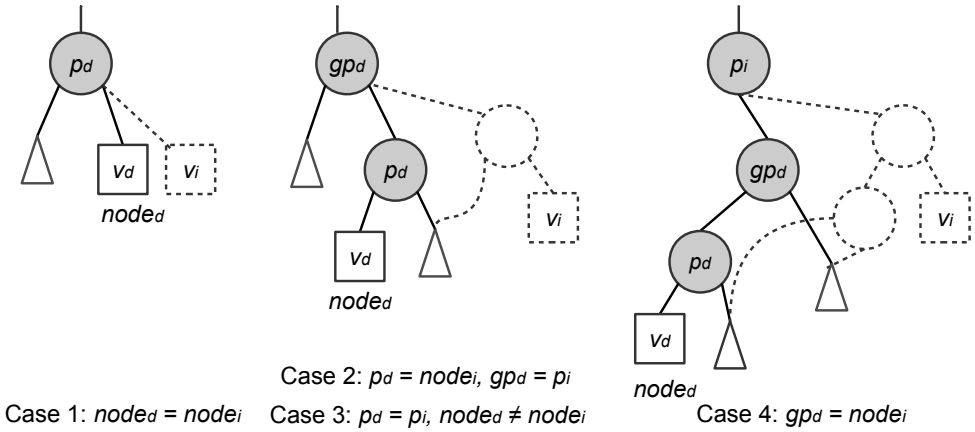


Figure 9.7: Special cases of REPLACE(v_d, v_i)

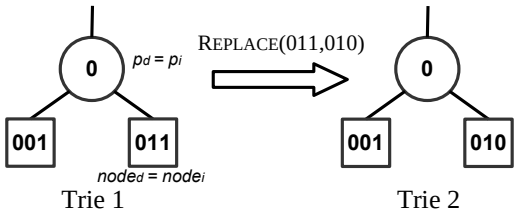


Figure 9.8: Replacing the key 011 with the key 010 (Special case 1)

As another example, consider Trie 1 in Figure 9.9. To replace the key 1010 with the key 1000, a CAS step replaces the internal node whose label is 10 with an internal node whose label is 100 and whose children are leaf nodes with labels 1000 and 1001 as shown in Trie 2 of Figure 9.9.

Consider Trie 1 in Figure 9.10. To replace the key 1100 with the key 1111, the children of the internal node with label 11 are set to the leaf node with label 1101 and the internal node whose label is 111 and whose children are leaf nodes with keys 1110 and 1111 as shown in Trie 2 of Figure 9.10. To avoid an ABA problem,

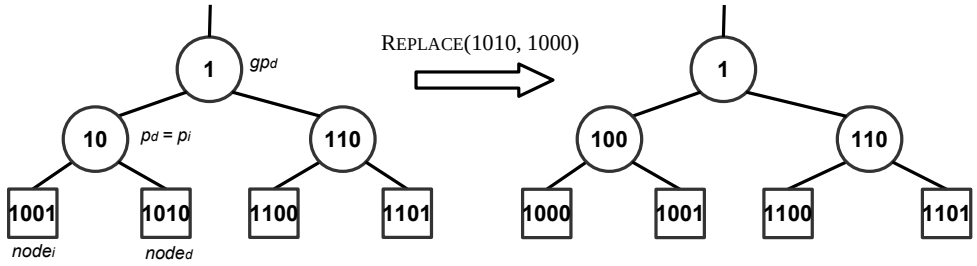


Figure 9.9: Replacing the key 1010 with the key 1000 (Special case 3)

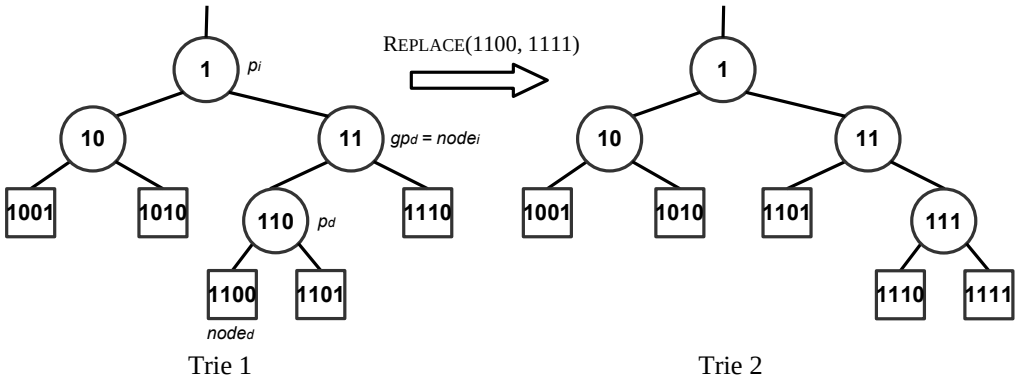


Figure 9.10: Replacing the key 1100 with the key 1111 (Special case 4)

the internal node with label 11 is also replaced by a new copy. So, to perform REPLACE(1100, 1111), a CAS step replaces the internal node whose label is 11 with the new copy of that internal node.

To apply our methodology, any node whose pointer will be changed or removed from the trie by a CAS step must be flagged before the CAS step. Consider an update that is done by performing one CAS step which removes a leaf node from the trie. Since no pointer in the data structure points to the leaf after the CAS

step is performed and a leaf node does not have any children to change, it is not necessary to flag the leaf node.

Consider a REPLACE(v_d, v_i) operation that requires to perform two CAS steps. Since the second CAS removes the leaf with key v_d from the trie, a pointer in the trie still points to the leaf between the first and second CAS step. Recall that the REPLACE operation is linearized at the first CAS step. So, in this case, we also flag the leaf node with label v_d before the CAS steps are performed. Any other process that reaches the leaf between two CAS steps can see the leaf node is flagged, so that using the information of the descriptor of the REPLACE operation, the process can check whether that the REPLACE is already linearized. If so, the process behaves as if the leaf node is not in the trie.

9.1 Representation of the Trie in Memory

The objects that are used in our Patricia trie implementation are shown in Figure 9.11. The Patricia trie is represented using Leaf and Internal objects which are subtypes of Node objects. Each Node object has a *label* field, which is never changed after initialization and stores the Node's label. An Internal object has an array *child* of two Node objects that stores pointers to the children of the Node.

For simplicity, our trie initially contains an Internal Node *Root* whose *label* is the empty string and whose children are two dummy Leaf Nodes with *label*s 0^ℓ and

```
1.  type Leaf (subtype of Node)                          ▷ represent a key
2.      String label                                     ▷ the value of the key
3.      Info info                                        ▷ descriptor of update

4.  type Internal (subtype of Node)
5.      String label          ▷ the longest common prefix of its children's labels
6.      Node[2] child                                    ▷ left and right child
7.      Info info                                        ▷ descriptor of update

8.  type Flag (subtype of Info)
9.      Internal[4] flag                                 ▷ Nodes to be flagged
10.     Info[4] oldInfo                  ▷ expected values of CASs that flag
11.     Internal[2] unflag                               ▷ Nodes to be unflagged
12.     Internal[2] par                  ▷ Nodes whose children to be changed
13.     Node[2] old                      ▷ expected children of Nodes par
14.     Node[2] new                      ▷ children of par to be changed to
15.     Leaf rmvLeaf                                     ▷ Leaf to be flagged
16.     Boolean flagDone                 ▷ set to true if flagging is successful

17. type Unflag (subtype of Info)                        ▷ has no fields
```

Figure 9.11: Object types used to implement Patricia trie

1^ℓ as shown in Figure 9.12. For simplicity, we assume the dummy keys 0^ℓ and 1^ℓ cannot be elements of D. This ensures that the trie always has at least two Leaf Nodes and avoids special cases of update operations that would occur if the *Root* were a Leaf.

Each Node object also has an *info* field that stores a pointer to an Info object that serves as the descriptor of the update operation that is in progress at the Node (if any). Instead of using a *status* field as in our doubly-linked list implementation, we use a slightly different approach for the Info objects. Info objects have two subtypes: Flag and Unflag. An Unflag object is used to indicate that no update is

▷ Initialization

18. *lChild* ← new Leaf(0^ℓ, new Unflag())
19. *rChild* ← new Leaf(1^ℓ, new Unflag())
20. *Root* ← new Internal(ε, [*lChild*, *rChild*], new Unflag())

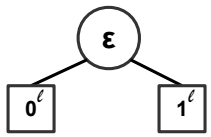


Figure 9.12: Initialization of the Patricia trie

in progress at a Node. We say a Node is *flagged* or *unflagged*, depending on whether its *info* field stores a Flag or Unflag object. Initially, the *info* field of each Node is an Unflag object. Unflag objects have no field and they are used instead of null pointers to avoid the ABA problem in the *info* field of a Node. The *info* and *child* fields of Internal Nodes are changed using CAS steps. However, a Leaf Node gets flagged by writing a Flag object into its *info* field. (A Leaf Node is only flagged before it is removed from the trie. We show that after a Leaf Node is flagged, its *info* field will never be changed again.)

A Flag object has a number of fields. The *flag* field stores the Nodes to be flagged before the trie is changed and the *unflag* field stores the Nodes to be unflagged after the trie is changed. Before creating a Flag object, an update reads the *info* field of each Node that will be affected by the update before reading that Node's *child* field. This value read from the *info* field is stored in the Flag's *oldInfo* field, and is used as the expected value by the CAS that flags the Node. This ensures that

if the Node is successfully flagged, it has not changed since its children were read. The boolean *flagDone* field is set to true when the flagging for the update has been completed. In the case of a REPLACE operation, the $rmvLeaf$ field points to the Leaf to be removed by the update after flagging is complete. The actual changes to the trie to be made are described in three more array fields of the Flag object: par, old and new. For each i, the update should CAS the appropriate *child* pointer of $par[i]$ from $old[i]$ to $new[i]$. As we shall see, once all Nodes are successfully flagged, the CAS on each *child* pointer will be guaranteed to succeed because that pointer cannot have changed since the old value was read from it.

9.2 Algorithm Descriptions

Pseudo-code for our Patricia trie implementation is given on page 267 to 270. A CAS step on an element of the *child* field of a Node is called a *child CAS*. A CAS step that attempts to set the *info* field of a Node to a Flag object is called a *flag CAS*.

If any child CAS step is executed for an update, the update operation is successful and it is linearized at the first such child CAS. If a REPLACE operation performs two different child CAS steps, the REPLACE operation also flags the Leaf Node of the old key before the first child CAS step. We say the Leaf is *logically removed* from the trie at the first child CAS step. Any operation that later reaches

21. INSERT($v \in U$): Boolean
22. while(true)
23. $I \leftarrow$ null
24. $\langle -, p, node, -, pI, keyInTrie \rangle \leftarrow$ SEARCH(v) ▷ search for a location to insert key
25. if $keyInTrie$ then return false ▷ key is already in the trie
26. $nodeI \leftarrow node.info$
27. $copy \leftarrow$ new copy of $node$
28. $new \leftarrow$ CREATENODE($copy$, new Leaf(v, new Unflag()), $nodeI$)
29. if $new \neq$ null then ▷ if Node new is successfully created
30. if $node$ is Internal then
31. $I \leftarrow$ NEWFLAG($[p, node]$, $[pI, nodeI]$, $[p]$, $[p]$, $[node]$, $[new]$, null)
▷ if no interference detected, create insertion's descriptor
32. else $I \leftarrow$ NEWFLAG($[p]$, $[pI]$, $[p]$, $[p]$, $[node]$, $[new]$, null)
▷ if no interference detected, create insertion's descriptor
33. if $I \neq$ null and HELP(I) then return true ▷ try to perform the insertion

34. DELETE($v \in U$): Boolean
35. while(true)
36. $I \leftarrow$ null
37. $\langle gp, p, node, gpI, pI, keyInTrie \rangle \leftarrow$ SEARCH(v) ▷ search for key v to be deleted
38. if $\neg keyInTrie$ then return false ▷ v is not in the trie
39. $sibling \leftarrow p.child[1 - (|p.label| + 1)$th bit of $v]$
▷ read the sibling of the Node with key v
40. if $gp \neq$ null then
41. $I \leftarrow$ NEWFLAG($[gp, p]$, $[gpI, pI]$, $[gp]$, $[gp]$, $[p]$, $[sibling]$, null)
▷ in no interference detected, create deletion's descriptor
42. if $I \neq$ null and HELP(I) then return true ▷ try to perform the deletion

43. REPLACE($v_d \in U$, $v_i \in U$): Boolean
44. while(true)
45. $I \leftarrow$ null
46. $\langle gp_d, p_d, node_d, gpI_d, pI_d, keyInTrie_d \rangle \leftarrow$ SEARCH(v_d) ▷ search for key v_d to be deleted
47. if $\neg keyInTrie_d$ then return false ▷ v_d is not in the trie
48. $\langle -, p_i, node_i, -, pI_i, keyInTrie_i \rangle \leftarrow$ SEARCH(v_i) ▷ search for a location to insert v_i
49. if $keyInTrie_i$ then return false ▷ v_i is already in the trie
50. $nodeI_i \leftarrow node_i.info$
51. $sibling_d \leftarrow p_d.child[1 - (|p_d.label| + 1)$th bit of $v_d]$ ▷ read sibling of Node with v_d
52. if $gp_d \neq$ null and $node_i \notin \{ node_d, p_d, gp_d \}$ and $p_i \neq p_d$ then ▷ **the general case**
53. $copy_i \leftarrow$ new copy of $node_i$
54. $new_i \leftarrow$ CREATENODE($copy_i$, new Leaf(v_i, new Unflag()), $nodeI_i$)
55. if $new_i \neq$ null and $node_i$ is Internal then ▷ if Node new_i is successfully created
56. $I \leftarrow$ NEWFLAG($[gp_d, p_d, p_i, node_i]$, $[gpI_d, pI_d, pI_i, nodeI_i]$, $[gp_d, p_i]$, $[p_i, gp_d]$, $[node_i, p_d]$, $[new_i, sibling_d]$, $node_d$) ▷ if no interference, create update's descritor
57. else if $new_i \neq$ null and $node_i$ is Leaf then ▷ if Node new_i is successfully created
58. $I \leftarrow$ NEWFLAG($[gp_d, p_d, p_i]$, $[gpI_d, pI_d, pI_i]$, $[gp_d, p_i]$, $[p_i, gp_d]$, $[node_i, p_d]$, $[new_i, sibling_d]$, $node_d$) ▷ if no interference, create update's descritor
59. else if $node_i = node_d$ then
60. $I \leftarrow$ NEWFLAG($[p_d]$, $[pI_d]$, $[p_d]$, $[p_d]$, $[node_i]$, [new Leaf(v_i, new Unflag())], null) ▷ if no interference, create update's descritor - **special case 1**
61. else if ($node_i = p_d$ and $p_i = gp_d$) or ($gp_d \neq$ null and $p_i = p_d$) then
62. $new_i \leftarrow$ CREATENODE($sibling_d$, new Leaf(v_i, new Unflag()), $sibling_d.info$)
63. if $new_i \neq$ null then ▷ if Node new_i is successfully created
64. $I \leftarrow$ NEWFLAG($[gp_d, p_d]$, $[gpI_d, pI_d]$, $[gp_d]$, $[gp_d]$, $[p_d]$, $[new_i]$, null) ▷ if no interference, create update's descritor - **special case 2 and 3**
65. else if $node_i = gp_d$ then
66. $pSib_d \leftarrow gp_d.child[1 - (|gp_d.label| + 1)$th bit of $v_d]$
67. $child_i \leftarrow$ CREATENODE($sibling_d$, $pSib_d$, -)
68. if $child_i \neq$ null then ▷ if Node $child_i$ is successfully created
69. $new_i \leftarrow$ CREATENODE($child_i$, new Leaf(v_i, new Unflag()), -)
70. if $new_i \neq$ null then ▷ if Node new_i is successfully created
71. $I \leftarrow$ NEWFLAG($[p_i, gp_d, p_d]$, $[pI_i, gpI_d, pI_d]$, $[p_i]$, $[p_i]$, $[node_i]$, $[new_i]$, null) ▷ if no interference, create update's descritor-**special case 4**
72. if $I \neq$ null and HELP(I) then return true ▷ try to perform the update

73. FIND($v \in U$): Boolean

74. $\langle$ -, -, -, -, -, $keyInTrie\rangle \leftarrow$ SEARCH(v) ▷ search for key v in the trie

75. return $keyInTrie$

76. SEARCH($v \in U$): $\langle$ Node, Node, Node, Info, Info, Boolean$\rangle$
Post-conditions:
If Search(v) returns $\langle gp, p, node, gpI, pI, keyInTrie\rangle$, it satisfies the following postconditions:
(1) At some configuration during the SEARCH, $gp.info$ was gpI (if gp is not null).
(2) Later during the SEARCH, p was a child of gp (if gp is not null).
(3) Later during the SEARCH, $p.info$ was pI.
(4) Later during the SEARCH, $p.child[i] = node$ for some i, and $(p.label) \cdot i$ is a prefix of v.
(5) If $node$ is an Internal Node, $node.label$ is not a prefix of v.
(6) If $keyInTrie$ is true, the Node whose $label$ is v is logically in the trie at some configuration during SEARCH(v).
(7) If $keyInTrie$ is false, then at some configuration during the SEARCH, no Node containing v is logically in the trie.

77. $\langle p, pI\rangle \leftarrow \langle$null, null$\rangle$

78. $node \leftarrow Root$ ▷ start from $Root$

79. while ($node$ is Internal and $node.label$ is a prefix of v) ▷ v might be in subtree whose root is $node$

80. $\langle gp, gpI\rangle \leftarrow \langle p, pI\rangle$

81. $\langle p, pI\rangle \leftarrow \langle node, node.info\rangle$ ▷ read the $info$ field before reading the $child$ pointer

82. $node \leftarrow p.child[(|p.label| + 1)$th bit of $v]$

83. if $node$ is Leaf then ▷ detect if Leaf is already replaced by another key

84. $rmvd \leftarrow$ LOGICALLYREMOVED($node.info$)

85. $keyInTrie \leftarrow (node.label = v$ and $\neg rmvd)$

86. else $keyInTrie \leftarrow$ false

87. return $\langle gp, p, node, gpI, pI, keyInTrie\rangle$

88. NEWFLAG($flag$, $oldInfo$, $unflag$, par, old, new, $rmvLeaf$): {null, Info}

89. for $i \leftarrow 0$ to $(|oldInfo| - 1)$, ▷ detect if other updates in progress

90. if $oldInfo[i]$ is Flag then

91. HELP($oldInfo[i]$) ▷ help an update to complete

92. return null ▷ retry my update

93. if $flag$ has duplicates with different values in $oldInfo$ then return null ▷ retry my update

94. else remove duplicates in $flag$ and $unflag$ (and corresponding entries of $oldInfo$)

95. sort elements of $flag$ by their $label$s and permute elements of $oldInfo$ accordingly

96. return new Info($flag$, $oldInfo$, $unflag$, par, old, new, $rmvLeaf$, false) ▷ create the update's descriptor

```
97.  CreateNode(node1: Node, node2: Node, info: Info): {null, Node}
98.    if node1.label is a prefix of node2.label or node2.label is a prefix of node1.label then
99.        if info is Flag then Help(info)      ▷ if another update in progress detected, help it
100.       return null                                                    ▷ retry my update
101.   else return new Internal whose children are node1 and node2
                                                          ▷ create a new Internal Node
102. LogicallyRemoved(I: Info): Boolean
103.   if I is Unflag then return false
104.   return (I.old[0] not in I.par[0].child) ▷ detect the Replace decribed by I is linearized

105. Help(I: Flag): Boolean
106.   i ← 0
107.   doChildCAS ← true
108.   while (i < |I.flag| and doChildCAS)
109.       CAS(I.flag[i].info, I.oldInfo[i], I)                              ▷ flag CAS
110.       doChildCAS ← (I.flag[i].info = I)
111.       i ← i + 1
112.   if doChildCAS then                  ▷ if all Nodes in I.flag are successfully flagged
113.       I.flagDone ← true
114.       if I.rmvLeaf ≠ null then I.rmvLeaf.info ← I                   ▷ flag the Leaf Node
115.       for i ← 0 to (|I.par| − 1)
116.           k ← (|I.par[i].label| + 1)th bit of I.new[i].label
117.           CAS(I.par[i].child[k], I.old[i], I.new[i])                ▷ child CAS
118.   if I.flagDone then
119.       for i ← (|I.unflag| − 1) down to 0
120.           CAS(I.unflag[i].info, I, new Unflag())                    ▷ unflag CAS
121.       return true                                          ▷ the update is completed
122.   else
123.       for i ← (|I.flag| − 1) down to 0
124.           CAS(I.flag[i].info, I, new Unflag())                      ▷ backtrack CAS
125.       return false                                            ▷ the update failed
```

that Leaf Node can determine that the key is already removed. We say a Node is *reachable* at configuration C if there is a path from the *Root* to the Node at C. We say a Leaf Node is *logically in the trie* at configuration C if the Node is reachable and not logically removed at C. We maintain the invariant that the Leaf Nodes that are logically in the trie (excluding the two dummy Leaf Nodes) contain exactly the set of keys in the set D.

First, we explain the routines that operations call. A SEARCH(v) is used by updates and the FIND operation to locate key v within the trie. The SEARCH(v) starts from the *Root* Node and traverses down the trie. At each step of the traversal, SEARCH(v) chooses the child according to the appropriate bit of v (line 82). The SEARCH(v) stops if it reaches an Internal Node whose *label* is not a prefix of v. We show that any Node visited by the SEARCH *was* reachable at some time during the SEARCH. Hence, if the SEARCH(v) does not return a Leaf containing v, there was a configuration during the SEARCH when no Leaf containing v was reachable. Moreover, the Node that is returned is the location where an INSERT would have to put v. If SEARCH(v) reaches a Leaf Node and the Leaf Node is logically in the trie, SEARCH(v) sets $keyInTrie$ to true (line 85).

As we described earlier, update operations must change the *child* pointers of the parent or grandparent of the Node returned by the SEARCH. The SEARCH operation returns *gp*, *p* and *node*, the last three Nodes reached (where *p* stands

for parent and *gp* stands for grandparent). A SEARCH also returns the values *gpI* and *pI* that it read from the *info* fields of *gp* and *p* before reading their *child* pointers. More formally, the SEARCH satisfies the postconditions mentioned in the pseudo-code (line 76).

To create a new Internal Node, the INSERT and REPLACE operations call CREATENODE with the two children Nodes as arguments. Sometimes, the operation also passes the *info* of one of the children Nodes to the CREATENODE. If there is an interference with other updates, the operation might pass wrong children to CREATENODE. In this case, the CREATENODE calls the HELP routine if the Info passed to it is a Flag object and then returns null to make the INSERT or REPLACE operation start from scratch (line 98–100).

After performing SEARCH, an update calls NEWFLAG to create a Flag object. For each Node that the update must flag, a value read from the *info* field of the Node during SEARCH is passed to NEWFLAG as the old value to be used in the flag CAS step. The NEWFLAG routine checks if all old values for *info* fields are Unflag objects (line 90). If some *info* field is not an Unflag object, then there is some other incomplete update operating on that Node. In this case, the NEWFLAG routine tries to complete the incomplete update by calling the HELP routine (line 91), and then returns null, which causes the update to restart. In some cases, a REPLACE operation must change two parts of the trie, and if those parts overlap, the list of

Nodes to be flagged by the REPLACE might contain duplicates. If the duplicate elements do not have the same old values, their *child* fields might have changed since the operation read them, so NEWFLAG returns null and the operation starts over (line 93). Otherwise, only one copy of each duplicate element is kept (line 94). The NEWFLAG routine sorts the Nodes to be flagged by their *label*s (to ensure non-blocking progress property) and returns the new Flag object (line 95–96).

After an update u creates a Flag object I, it calls HELP(I) to attempt to complete the update. A step that is performed inside a call to HELP(I) is called a *step of* I. First, it uses flag CAS steps to store the Flag object I in the *info* fields of the Nodes in I.*flag* (line 109). If all Nodes are flagged successfully, the *flagDone* field of the Flag object is set to true (line 113). The value of the *flagDone* field is used to coordinate processes that help the update. Suppose a process p is executing HELP(I). After p performs a flag CAS on a Node x, if it sees a value different from I in x's *info* field, there are two possible cases. The first case is when all Nodes were already successfully flagged for I by other processes running HELP(I), and then x was unflagged before p tries to flag x. (Prior to this unflagging, some process performed the child CAS steps of I successfully.) The second case is when no process flags x successfully for I. Since the *flagDone* field of I is set to true only after all Nodes are flagged successfully, p checks the value of the *flagDone* field on line 118 to determine which case of these two cases occurred. If *flagDone* is true,

the modifications to the trie for update u have been made. If *flagDone* is false, the update operation cannot be successfully completed, so all Internal Nodes that got flagged earlier are unflagged by *backtrack CAS* steps at line 123–124 and the update u will have to start over.

After flagging all Nodes successfully and setting $I.flagDone$, if $I.rmvLeaf$ is non-null, the *info* field of that Leaf is set to I (line 114). Only the two-step Replace operations flag a Leaf. Then, Help(I) changes the *child* fields of Nodes in $I.par$ using child CAS steps (line 115–117). Finally, Help(I) uses *unflag CAS* steps to unflag the Nodes in $I.unflag$ and returns true (line 118–121). $I.unflag$ includes the Nodes that were flagged earlier, except those that were removed from the trie. (Any Node deleted by the update remains flagged forever.)

An Insert(v) operation first calls Search(v). Let $\langle$-, p, $node$, -, -, $keyInTrie\rangle$ be the result returned by Search(v). If $keyInTrie$ is true, Insert(v) returns false since the trie already contains v (line 25). Otherwise, the insertion attempts to replace $node$ with a Node created at line 28, whose children are a new Leaf Node containing v and a new copy of $node$. (See Figure 9.6.) Thus, the parent p of $node$ must be flagged. A new copy of $node$ is used to avoid the ABA problem. If $node$ is an Internal Node, since $node$ is replaced by a new copy, Insert(v) must flag $node$ permanently (line 31).

A Delete(v) operation first calls Search(v). Let $\langle gp, p, node$, -, -, $keyInTrie\rangle$

be the result returned by the SEARCH(v). If $keyInTrie$ is false, DELETE(v) returns false since the trie does not contain v (line 38). Then, DELETE(v) replaces p by the sibling of $node$. (See Figure 9.3.) So, DELETE(v) must flag the grandparent gp and the parent p of $node$ (line 41). Since p is removed from the trie, only gp must be unflagged after the deletion is completed.

A REPLACE(v_d, v_i) operation first calls SEARCH(v_d) and SEARCH(v_i), which return $\langle gp_d, p_d, node_d, \text{-}, \text{-}, keyInTrie_d\rangle$ and $\langle \text{-}, p_i, node_i, \text{-}, \text{-}, keyInTrie_i\rangle$. The REPLACE checks that v_d is in the trie and v_i is not, as in the INSERT and DELETE operations (line 46–49). If either test fails, the REPLACE operation returns false.

In the general case of the REPLACE operation (line 52–58), we create a Flag object which instructs the HELP routine to perform the following actions. The REPLACE flags the same Nodes that an INSERT(v_i) and a DELETE(v_d) would flag. After flagging these Nodes, the Leaf $node_d$ also gets flagged. Then, v_i is added to the trie, as in INSERT(v_i). When the new Leaf Node containing v_i is added to the trie, the Leaf $node_d$, which contains v_d, becomes logically removed, but not physically removed yet. Then, $node_d$ is physically deleted as in DELETE(v_d). After $node_d$ is flagged, any SEARCH that reaches $node_d$ checks if p_i is a parent of $node_i$ using $node_d.info$. If it is not, it means the new Leaf containing v_i is already inserted and the operation behaves as if v_d is already removed.

There are four special cases of REPLACE(v_d, v_i) that requires only one child

CAS step. Although the code for these cases looks somewhat complicated, it simply implements the actions described in Figure 9.7 by creating a Flag object and calling HELP. The insertion of v_i replaces $node_i$ by a new Node. The cases when the deletion must remove $node_i$ or change $node_i.child$ are handled as special cases. So, the case that $node_d = node_i$ is one special case (line 59–60). (See Figure 9.8 for an example.) In the deletion, p_d is removed, so the case that $p_d = node_i$ or $p_d = p_i$ are also handled as a special case (line 61–64). (See Figure 9.9 for an example.) In the deletion, $gp_d.child$ is changed. So, the last special case is when $gp_d = node_i$ (line 65–71). (See Figure 9.10 for an example.) Here, we explain one special case in detail. The others are handled in a similar way. In case 2, $p_d = node_i$ and $gp_d = p_i$ (line 61). So, REPLACE(v_d, v_i) creates an Info object that contains instructions to flag gp_d and $node_i$, replace $node_i$ with a new Internal Node whose non-empty children are a new Leaf containing v_i and the sibling of $node_d$, and then unflag gp_d (line 61–64).

Our Patricia trie implementation can also be used to store unbounded length strings. One approach would be to append \$ to the end of each string. To encode a binary string, 0, 1 and \$ can be represented by 01, 10 and 11. Then, every encoded key is greater than 00 and smaller than 111, so 00 and 111 can be used as keys of the two dummy Leaf Nodes (instead of 0^ℓ and 1^ℓ). Moreover, since *label*s of Nodes never change, they need not fit in a single word.

9.3 Sketch of Correctness Proof of Patricia Trie

A detailed proof of correctness is provided in [40]. It is very similar to the correctness proof our doubly-linked list implementation, so we only provide a sketch here. First, we explain how linearization points are chosen for each operation. Let $\langle$ -, -, *node*, -, -, $keyInTrie\rangle$ be the result returned by SEARCH(v). If $keyInTrie$ is true, postcondition (6) of the SEARCH says there is a configuration during the SEARCH when *node* is logically in the trie and the SEARCH is linearized at that configuration. Otherwise, postcondition (7) of the SEARCH ensures there is a configuration during the SEARCH when no Leaf containing v is logically in the trie and the SEARCH is linearized at that configuration. The FIND operation is linearized at the linearization point of its call to SEARCH. If an update returns false, it is linearized at the linearization point of the SEARCH that caused the update to fail. Let I be a Flag object created by an update. If a child CAS performed by any call to HELP(I) is executed, the update is linearized at the *first* such child CAS. Next, we sketch the correctness proof in five parts. The first two parts prove that the successful CAS steps performed by all calls to HELP(I) proceed in the expected order. (See Figure 9.13.)

Part 1: Flagging Consider a Flag object I created by some update operation. In Part 1, we prove all Nodes in $I.flag$ are flagged for I when the first child CAS of I on each Node in $I.par$ occurs. More precisely, we prove that only the first flag CAS (by any of the helpers of I) on each Node can succeed and the flag CAS steps succeeded on Nodes in $I.flag$, ordered according to the Nodes' *label*s. Here, we also prove that if all Nodes in $I.flag$ are successfully flagged, none of these Nodes is unflagged before the first child CAS of I on each Node in $I.par$ occurs. The proofs of most lemmas in this part follow from the structure of the HELP routine.

To do this, we first show that the SEARCH operation satisfies its postconditions (1) to (5), described in the pseudo-code on page 269. The proof of the following lemma follows from the pseudo-code.

Lemma 9.1. *Assume* SEARCH(v) *returns* $\langle gp, p, node, gpI, pI, rmvd\rangle$. *Then, the following statements are true.*

1. *If gp is not null, then, at some configuration during the* SEARCH, *$gp.info$ was gpI, and at some later configuration during the* SEARCH, *p was a child of gp.*

2. *Then, at some later configuration during the* SEARCH, *$p.info$ was pI, and at some later configuration during the* SEARCH, *$p.child[i] = node$ for some i.*

3. *$(p.label) \cdot i$ is a prefix of v.*

4. *If $node$ is an Internal Node, $node.label$ is not a prefix of v.*

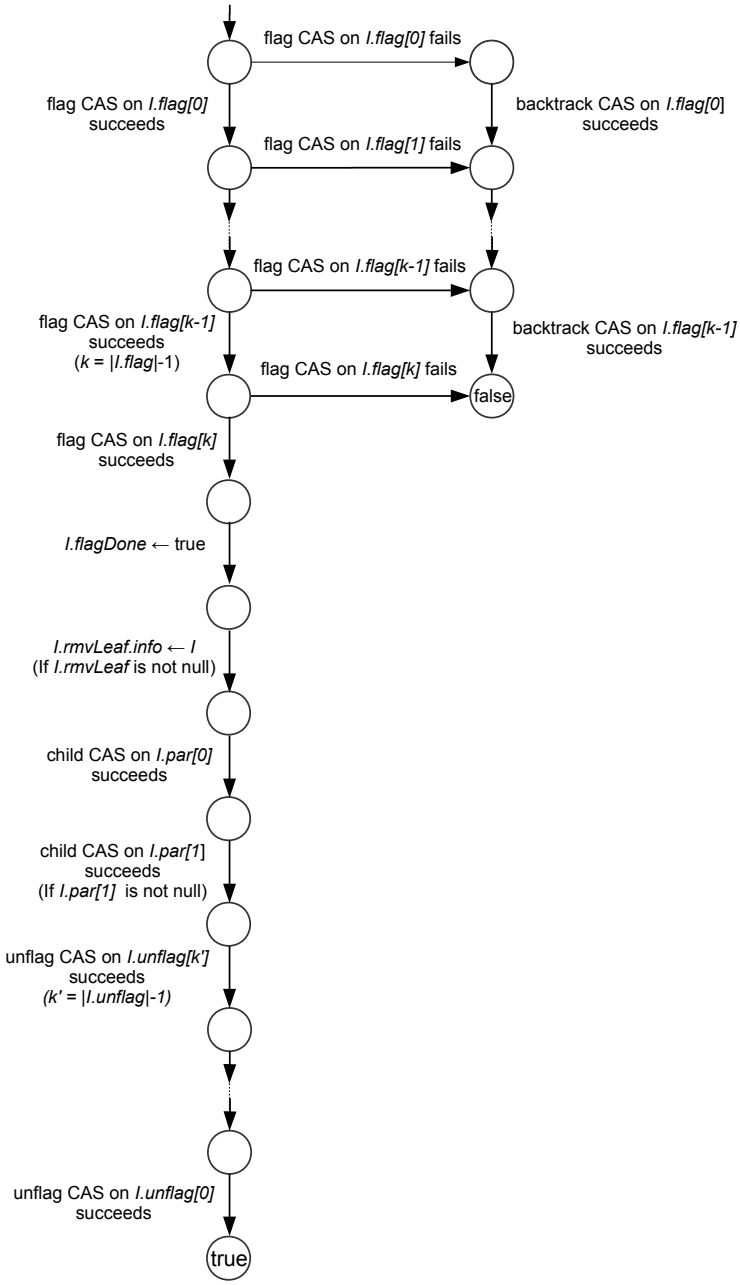


Figure 9.13: The correct order of steps inside HELP(I) for each Flag object I. (Steps can be performed by different calls to HELP(I).)

Lemma 9.1 is used to prove that each update preserves the following invariant, ensuring the structure is a trie.

Invariant 9.2. *If $x.child[i] = y$, then $(x.label) \cdot i$ is a prefix of $y.label$.*

Proof sketch. We show that when an Internal Node is created and when the *child* field of an Internal Node is changed, the invariant is preserved if the invariant was true at all earlier configurations. It is trivial to show that when the *Root* Node is created on line 20 and a new copy of a Node is made on line 27 or 53 and a new Internal Node is created on line 101, the invariant is preserved.

Next, we need to show when a child CAS of I succeeds, the invariant is preserved. The child CAS changes $I.par[j].child[i]$ from $I.old[j]$ to $I.new[j]$ (for some j) where i is the $(|I.par[j].label| + 1)$th bit of $I.new[j].label$. Thus, it suffices to show that $I.par[j].label$ is a proper prefix of $I.new[j].label$. I can be created on line 31, 32, 41, 56, 58, 60, 64 or 71. We consider different cases according to what line creates I.

Case 1: I is created at line 31 or 32. Let $\langle$ -, p, *node*, -, -, -$\rangle$ be the result returned by the call to SEARCH(v) on line 24 that precedes the creation of I. Then, *new* is the new Node that is created at line 28 and whose children are a new copy of *node* and a new Leaf Node whose *label* is v. In this case, $I.par[0] = p$ and $I.new[0] = new$. By Lemma 9.1, $p.child[k] = node$ for some k at an earlier configuration. Since the invariant is true at all earlier configurations, $(p.label) \cdot k$ is

a prefix of $node.label$. By Lemma 9.1, $(p.label) \cdot k$ is a prefix of v. Since $new.label$ is the longest common prefix of v and $node.label$, $(p.label) \cdot k$ is a prefix of $new.label$.

Case 2: I is created at line 41. Let $\langle gp, p, node, \text{-}, \text{-}, \text{-} \rangle$ be the result returned by the call to SEARCH(v) on line 37 that precedes the creation of I. Let $sibling$ be the child of p that is read at line 39. Since the invariant is true at all earlier configuarions, $p.label$ is a prefix of $sibling.label$. In this case, $I.par[0] = gp$, $I.old[0] = p$, and $I.new[0] = sibling$. Since the child CAS of I succeeds, $gp.child[i]$ was p just before the child CAS. Since the invariant is true at all earlier configurations, $(gp.label) \cdot i$ is a prefix of $p.label$. Since $p.label$ is a prefix of $sibling.label$, $(gp.label) \cdot i$ is a prefix of $sibling.label$.

The other cases can be proved by similar arguments. □

The following lemma describes how the *info* field of a Node is changed. It follows easily from the pseudo-code.

Lemma 9.3. *Let x be an Internal Node. Then, $x.info$ is initially set to a new Unflag object and the only changes to $x.info$ that can occur are (1) a flag CAS at line 109 that changes $x.info$ from an Unflag object to a Flag object, or (2) an unflag CAS at line 120 or a backtrack CAS at line 124 that changes $x.info$ from a Flag object to a newly created Unflag object.*

By Lemma 9.3, we can prove that the ABA problem on the *info* fields is avoided because whenever an *info* field is changed, it is set to a newly created Flag or Unflag

object.

Lemma 9.4. *The info field of a Node is never set to a value that has been stored there previously.*

Thus, if several helpers of an Info object try to perform a flag CAS, backtrack CAS or unflag CAS on a Node, only the first one can succeed. So, the next lemma follows from the code and shows that these successful flag CAS steps proceed in the order shown in Figure 9.13.

Lemma 9.5. *Let I be a Flag object. For $0 \leq i < |I.flag| - 1$, a flag CAS of I does not successfully flag $I.flag[i+1]$ unless $I.flag[i]$ is flagged earlier by I.*

If *doChildCAS* is true at line 112, then the *info* field of all Nodes in $I.flag$ is set to I earlier. So, we have the following lemma.

Lemma 9.6. *Let I be a Flag object. A child CAS step of I is preceded by flagging all Nodes in $I.flag$ using flag CAS steps of I.*

The following lemma shows the *info* of no Node in $I.flag$ is changed from I to an Unflag object before the first child CAS step of I on each Node in $I.par$.

Lemma 9.7. *If there is any child CAS of an Info object I, then there is no unflag or backtrack CAS of I before the first child CAS step of I on each Node in $I.par$.*

Proof. To derive a contradiction, assume, for some j, an unflag or backtrack CAS of I occurs before the first child CAS of I on $I.par[j]$. Let S be that child CAS and S' be the first unflag or backtrack CAS of I that occurs before S and H be an invocation of HELP that executes S'. So, H does not execute any child CAS of I on $I.par[j]$ before S and the $doChildCAS$ variable is false when H performs line 112 (before S'). Thus, just after H tries to flag some Node y by a flag CAS of I, H sets the $doChildCAS$ variable to false at an execution of line 110 when $y.info \neq I$. By Lemma 9.6, $y.info$ is set to I before S. Since only the first flag CAS of I on $y.info$ succeeds and a flag CAS of I on $y.info$ is performed just before that execution of line 110, $y.info$ is set to I before H reads $y.info$ on line 110. Then, $y.info$ is changed from I to another value before H reads $y.info$ at line 110, contradicting the fact that the first unflag or backtrack CAS of I is the step before S'. □

By Lemma 9.6 and 9.7, we have the following corollary.

Corollary 9.8. *Each Node in $I.flag$ is flagged by I when the first child CAS on each Node in $I.par$ occurs.*

Part 2: Child CAS Steps Let gp, p and $node$ be the three Nodes that a call to SEARCH(v) returns. Part 2 of our proof shows that successful flagging ensures that gp, p and $node$ are three consecutive Nodes in the trie just before the first child CAS of I, and that the first child CAS of I on each Node in $I.par$ succeeds (and

no others do).

Lemma 9.9. *For a Flag object I, the following statements are true.*

1. *The first child CAS performed by a helper of I on each Node in $I.par$ succeeds.*
2. *A child field of a Node is never set to a Node that has been stored there before.*
3. *If all Nodes in $I.flag$ are successfully flagged with I and $x = I.flag[i]$ for some i, then no child CAS of any other Flag object $I' \neq I$ changes $x.child$ between setting $x.info$ to I and the first child CAS of I on $I.par[j]$ for some j.*
4. *If a Node x becomes unreachable by a successful child CAS of I, $x.info = I$ at all configurations after the child CAS.*
5. *If a Node becomes unreachable, the Node never becomes reachable after that.*

Proof sketch. Lemma 9.9 is proved by induction on the length of the execution. It requires reasoning about the way flags act as locks. By Corollary 9.8, a Node x is flagged with a pointer to I when the first child CAS of I on x is performed. Since the flag CAS of I on x succeeded, x has not been flagged by any other update between when the update that created I read $x.info$ during its SEARCH and the flag CAS of I on x (by Lemma 9.4). It follows that no other update has flagged x between that read and the child CAS, and hence the *child* field of x has not changed during that interval. We prove using Lemma 9.1 that the old value used

by the child CAS is still in $x.child$ when the child CAS occurs, so it succeeds. This proves Statement 1 and 3.

The ABA problem on the *child* fields is avoided because whenever a child pointer is changed, the old child is permanently removed from the trie. It follows that if a Node x becomes reachable by a child CAS of I, x is newly created by the update that created I. This proves Statement 2 and 5.

If x becomes unreachable by a child CAS of I, $x.info = I$ when the child CAS occurs (by Corollary 9.8). Since $I.flagDone$ is set to true before the child CAS, no backtrack CAS of I is executed after the child CAS. By the pseudo-code, x is not in $I.unflag$. So, $x.info$ cannot be changed from I to an Unflag object after the child CAS of I. This proves Statement 4. □

Part 1 and 2 of the proof are mostly focused on the structure of the Help routine. So, any new update operation that preserves the main invariants of the trie could be added with only minor changes to the correctness proof.

Part 3: Linearizability of Search Operations In this part, we first show the rest of the post-conditions of the Search operation are satisfied. Then, we show how to linearize Find operations that terminate.

Let I be a Flag object that is created at line 56 or 58. Then, the number of elements of $I.par$, $I.old$ and $I.new$ is two and $I.rmvLeaf$ is set to a Leaf Node.

The leaf $I.rmvLeaf$ is flagged at line 114 before the first child CAS of I at line 117, so at any configuration after the first child CAS of I, $I.rmvLeaf$ is logically removed. The following lemma shows that after a Leaf Node is flagged by a Flag object, the Leaf Node is not flagged by another Flag object.

Lemma 9.10. *Let I be a Flag object that is created at line 56 or 58 and x be $I.rmvLeaf$. If $x.info$ is set to I by line 114, $x.info$ is not set to another Flag object after that.*

Proof Sketch. When a Leaf Node is flagged at line 114, the Leaf Node is reachable. Let s be the first child CAS of I on $I.par[1]$. Since x becomes unreachable by s, x is unreachable at all configurations after s (by Lemma 9.9.5). So, $x.info$ cannot be set to another Flag object after s.

When a Leaf Node is flagged by an Info object at line 114, the parent of the Leaf Node is also flagged with the same Info object (by the pseudo-code, Corollary 9.8 and Lemma 9.9.3). Let Node p be the parent of x at the configuration before s. It follows from Corollary 9.8 that p is flagged by I at all configurations between setting $x.info$ to I and s. Thus, $x.info$ cannot be set to another Flag object before s. □

We prove the following lemma by induction on the number of steps that the SEARCH operation has done.

Lemma 9.11. *Each Node a* SEARCH *visits was reachable at some configuration*

during the SEARCH.

Lemma 9.11 is used to prove postconditions (6) and (7) of SEARCH (that are described in the pseudocode on page 269). Suppose a call to SEARCH(v) returns $\langle$-, -, -, -, -, $keyInTrie\rangle$. It follows from Lemma 9.9.1, 9.9.2 and 9.11 that if $keyInTrie$ is true there was a configuration during the SEARCH when a Leaf Node containing v was in the trie. It follows from Invariant 9.2 and Lemma 9.9.1, 9.9.2, 9.10 and 9.11 that if $keyInTrie$ is false, there was a configuration during SEARCH when v was not in the trie.

So, each FIND operation is linearized at the configuration that is defined by the postcondition (6) or (7) of SEARCH according to the result that its call to SEARCH returns.

Part 4: Linearizability of Update Operations Part 4 proves that update operations are linearized correctly. Suppose an update is linearized at the first successful child CAS performed by any helper of the operation. We argue, using Lemma 9.9.1, that this first child CAS has the effect of implementing precisely the change shown in Figure 9.6, 9.3 or 9.7 atomically. In the case of a REPLACE operation, the linearization point of a successful REPLACE adds a new Leaf to the trie. If another operation accesses the Leaf Node that would be deleted by the REPLACE after that and before the second child CAS, the test performed by

LogicallyRemoved ensures that it behaves as if the Leaf is not in the trie. This is used to establish an invariant that proves all operations return correct results.

Let D be an auxiliary variable of type Patricia trie. D initially contains the keys 0^ℓ and 1^ℓ. Each time an operation is linearized, the same operation is atomically applied to D according to the sequential specification.

Invariant 9.12. *The Leaf Nodes that are logically in the trie at a configuration (excluding dummy Nodes) contain exactly the keys in D.*

Each successful update is linearized at its first child CAS step performed by any of its helpers. This step is the only step of the update that changes the Leaf Nodes that are logically in the trie and it has the effect of implementing precisely the change shown in Figure 9.3, 9.6 or 9.7 atomically. So, it follows from Invariant 9.12 that our trie implementation is linearizable.

Part 5: The Progress Property Finally, part 5 of the proof establishes progress. First, we show that the Search operation is wait-free.

Lemma 9.13. *The* Search *operation is wait-free.*

Proof. Let ℓ be the length of the keys in U. By Invariant 9.2, length of *node.label* increases by at least one in each iteration of the loop in the Search routine. Since *label*s of Nodes have length at most ℓ, there are at most ℓ iterations. □

Lemma 9.14. *The implementation is non-blocking.*

Proof Sketch. To derive a contradiction, assume after a configuration C, no operation terminates or fails. Let I be a Flag object created by an update that is running after C. If a call to HELP(I) returns true, the update terminates, so after C, all calls to HELP(I) return false. Thus, all calls to HELP(I) set $doChildCAS$ to false because they fail to flag an Internal Node successfully after C. Consider the group of all calls to HELP(I). We say the group *blames* the Internal Node that is the first Node that no call to HELP(I) could flag successfully. Let g_0, ..., g_m be all these groups ordered by the *label*s of the Nodes that they blame. Since g_m blames an Internal Node x, x is flagged by some other group g_i where $0 \leq i < m$. Thus, g_i blames some other Node y whose *label* is less than x. So, g_i flags x before attempting to flag y, contradicting the fact that g_i attempts to flag Internal Nodes in order. □

9.4 Empirical Evaluation of Patricia Trie

Here, we provide the results of the experiments we conducted on a multi-core machine to evaluate our Patricia trie implementation (PAT). Our results show that PAT performs consistently well under different scenarios. A non-blocking skip list (ConcurrentSkipListMap) has been implemented in the Java class library by Doug Lea that can also be employed to represent a set. We experimentally com-

pared the performance of our implementation with non-blocking binary search trees (BST) [14], non-blocking k-ary search trees (4-ST) [10], ConcurrentSkipListMap (SL) of the Java library, lock-based AVL trees (AVL) [7] and non-blocking hash tries (Ctrie) [39]. For the k-ary search trees, we use the value $k = 4$, which was found to be optimal in [10].

The experiments were executed on a Sun SPARC Enterprise T5240 with 32GB RAM. The machine had two UltraSPARC T2+ processors, each with eight 1.2GHz cores, for a total of 128 hardware threads. The experiments were run in Java. The Sun JVM version 1.7.0_3 was run in server mode. The heap size was set to 2G. This ensures the garbage collector would not be invoked too often, so that the measurements reflect the running time of the algorithms themselves. Using a smaller heap size affects the performance of BST, 4-ST and PAT more than AVL and SL since they create more objects.

We evaluated the algorithms in different scenarios. We ran the algorithms using uniformly distributed keys in two different ranges: $(0, 10^2)$ to measure performance under high contention and $(0, 10^6)$ for low contention. In the range $(0, 10^2)$, the tree is very small and operations are more likely to access the same part of the tree. To measure the scalabilities of the implementations, we increased the number of concurrent threads from 1 to 128. We ran experiments with two different operation ratios: 5% INSERTs, 5% DELETEs and 90% FINDs (i5-d5-f90), and 50% INSERTs,

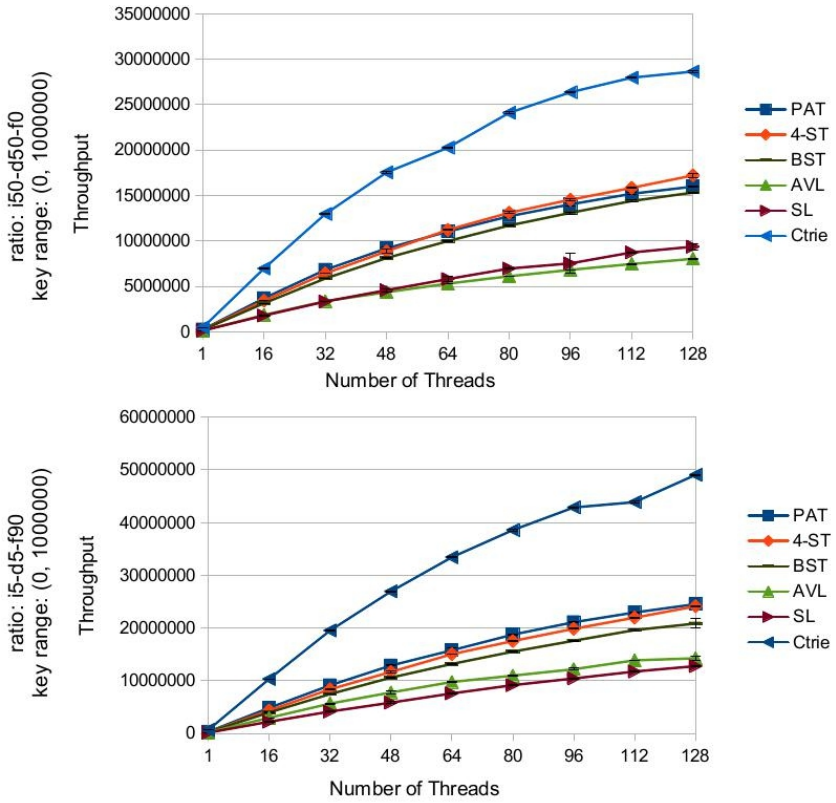


Figure 9.14: Uniformly distributed keys with key range $(0, 10^6)$

50% DELETEs and 0% FINDs (i50-d50-f0).

Since the REPLACE operation is not used in these sets of experiments, we made some minor optimizations to the pseudo-code. For example, we eliminated the *rmvd* variable in SEARCH operations.

Since the Java compiler optimizes its running code, before each experiment, we ran the code for ten seconds for each implementation. We started each experiment with a tree initialized to be half-full, created by running updates in the ratio i50-d50-f0 until the tree is approximately half-full. The average throughputs (the number

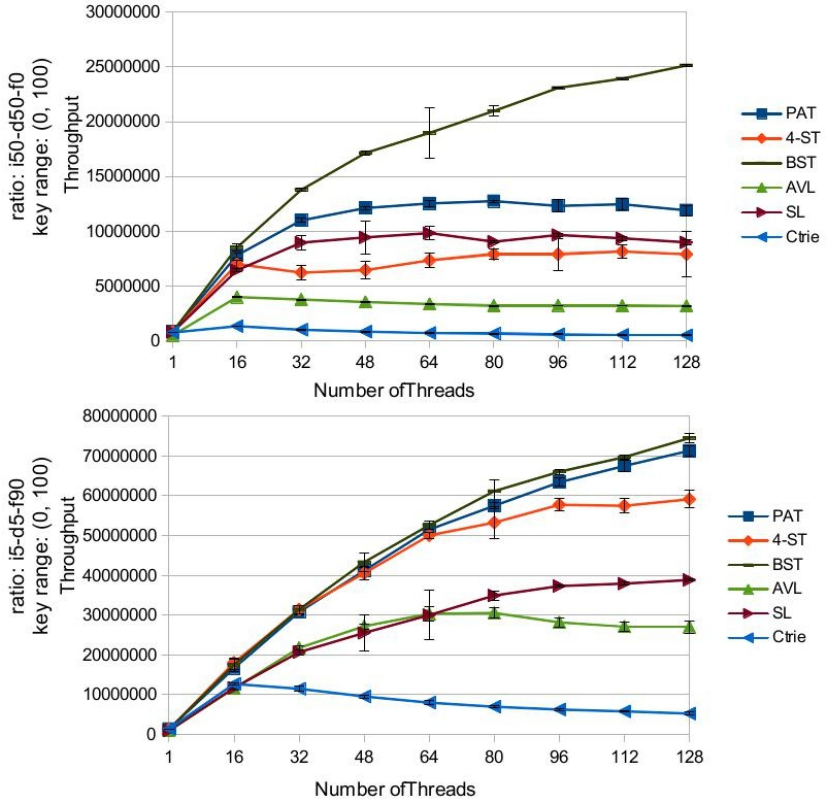


Figure 9.15: Uniformly distributed keys with key range $(0, 10^2)$

of operations terminated per second) are shown in the charts. Each data point in our graphs is the average of eight 4-second trials. (The error bars in the charts show the standard deviation.)

For uniformly distributed keys, algorithms scale well under low contention (key range of $(0, 10^6)$). (See Figure 9.14.) Under very high contention (key range of $(0, 10^2)$), most scale reasonably well when the fraction of updates is low, but experience problems when all operation are updates. (See Figure 9.15.) When the range is $(0, 10^6)$, Ctrie outperforms all others because the height of the Ctrie is kept very

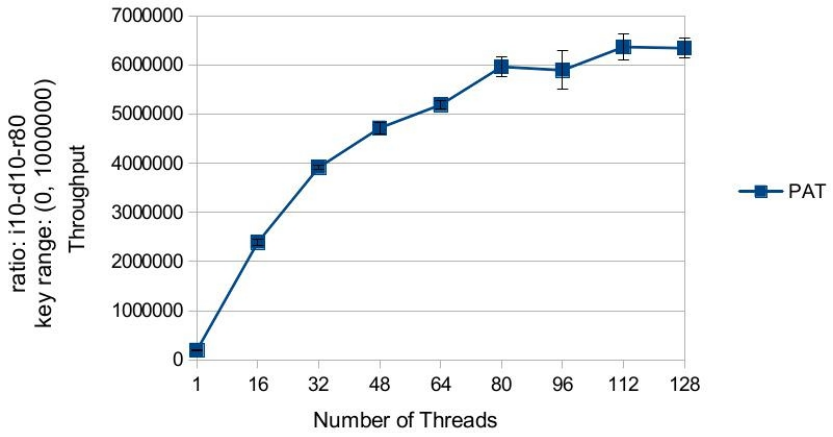


Figure 9.16: Replace operations of PAT

small by having nodes with 32 children. However when the range is $(0, 10^2)$ and the contention is very high, Ctrie does not scale. Excluding Ctrie, when the range is $(0, 10^6)$, PAT, 4-ST and BST outperform AVL and SL. Since updates are more expensive than FINDs, the throughput is greater for i5-d5-f90 than for i50-d50-f0.

To evaluate the REPLACE operations, we ran an experiment with 10% INSERTs, 10% DELETEs and 80% REPLACE operations (i10-d10-r80) and a key range of $(0, 10^6)$ on uniformly random keys. (See Figure 9.16.) We could not compare these results with other data structures since none provides atomic REPLACE operations. As the chart shows, the REPLACE operation scales well as the number of threads increases. Since the REPLACE operation is doing twice as much as work as other updates, we see by comparing Figure 9.16 to top of Figure 9.14 that the number of REPLACE operations that can be completed per second is approximately half the

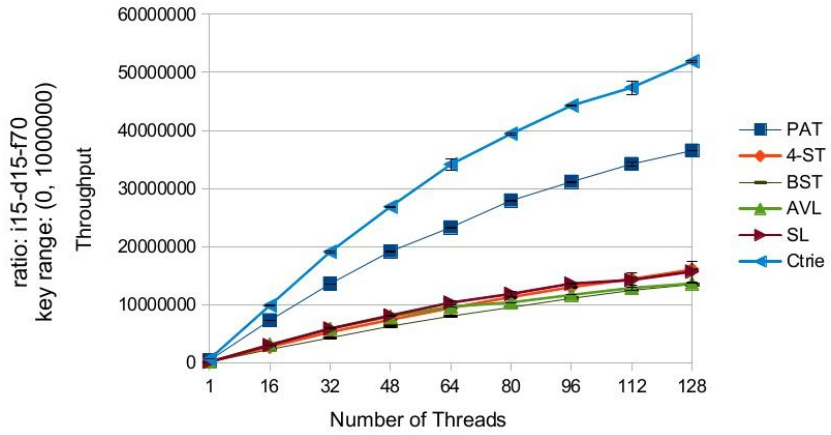


Figure 9.17: Non-uniformly distributed keys (The lines for 4-ST, BST, AVL and SL overlap.)

number of operations INSERTs and DELETEs.

We also performed some experiments on non-uniformly distributed random keys. (See Figure 9.17.) To generate non-uniform keys, processes performed operations on sequences of 50 consecutive keys from the range $(0, 10^6)$, starting from a randomly chosen key. In this experiment, since tries maintain a fixed height without doing expensive balancing operations, Ctrie outperforms all others and PAT outperforms others except Ctrie greatly. Since the results of these experiments for other operations ratios were similar, only the chart for the ratio i15-d15-f70 is presented here. Longer sequences of keys degrade the performance of BST and 4-ST even further.

10 Conclusion

In our approach, similar to [8, 9, 10, 14], an update creates Info objects and sometimes duplicates Nodes. Creating these objects introduces some overhead. Despite this, empirical evaluations in [9, 10, 35] and Section 8.2 and 9.4 confirm the practicality and scalability of this technique. However, finding more efficient memory management techniques is also an important area for future work.

One well-known memory management methodology is the use of *hazard pointers* [33]: each process has a number of shared pointers, called hazard pointers that can be written only by the owner process, but can be read by all processes. A hazard pointer is either null or points to an object that might be accessed by the process later. If a hazard pointer points to an object, the memory allocated to that object cannot be freed for reuse. In our Patricia trie, each process would require six hazard pointers to point to the latest three Nodes that are accessed during its latest call to SEARCH and their Info objects.

In our doubly-linked list, there can be a Cursor c such that $c.node$ points to a

Node that has been removed from the list. When UpdateCursor(c) is called, it may traverse an arbitrary long path of removed Nodes (on line 85 or 88) to reach a Node that is in the list. So, the hazard pointer memory management methodology cannot be applied directly to our list. One possible solution would be to assign a timestamp to a Node at the time it is removed from the list, and then the memory allocated to a Node with timestamp t cannot be freed if there is a hazard pointer to a Node whose timestamp is less than t. However, in this solution, a Cursor that is not used frequently might prevent freeing the memory of many Nodes. To avoid this, UpdateCursor on such a Cursor can be called before trying to free the memory allocated to Nodes. Future work includes applying different memory management techniques to our implementations and empirically evaluating such techniques on our implementations. Hart et al. [24] evaluated different memory management techniques. Their results show that there is no globally optimal scheme and the data structure, the workload, and the execution environment can dramatically affect memory reclamation performance.

We used two different techniques to unflag Nodes in our doubly-linked list and Patricia trie implementation. In our list, Nodes are unflagged by setting a bit in the descriptor object and, in our trie, Nodes are unflagged by setting their pointers to empty descriptor objects using CAS steps. There are trade-offs between these two approaches. The first approach unflags Nodes in one step regardless of the number

of Nodes that must be unflagged. So, the first approach is more efficient when there are many Nodes to be unflagged since the second approach creates an empty descriptor object and then performs a CAS step to unflag each Node. However, to prove that there is no ABA problem on the descriptor fields of Nodes is simpler when the second approach is used. Besides, the first approach might require a more complicated memory management technique since it might cause a chain of pointers from Nodes in the data structure to Nodes that are removed even when the operation that removed these Nodes terminated. (See an example of this on page 47.) So, the second approach is more suitable for complex data structures.

Writing detailed correctness proofs helped us to correct bugs in earlier versions and then verify the correctness of our implementations. It also helped us to simplify the pseudo-code and improve its complexity. In the process of writing detailed proofs, we found some bugs in earlier versions of the code and we reordered some lines of the code to correct the bugs. As an example of how the proof helped us to simplify the code, in our doubly-linked list implementation, we used to also check the *state* of *c.node* on line 82, but, in the process of writing the proof, we realized we can simplify the code by removing this check. We also realized that adding lines 98–100 improved the amortized complexity of the list although these lines are not necessary for the correctness of the implementation.

Future work includes generalizing and applying our coordination scheme to other

data structures. Although the proof of correctness and analysis is complex, it is modular, so it could be applied more generally to other non-blocking data structures. If the methodology is generalized for some class of data structures, then it could be applied to any data structure in that class without requiring correctness proof to be written from scratch. Brown et al. [9] have proposed a general technique for non-blocking trees that support one change to the tree atomically. Our approach can also be used for update operations that change more than one pointer atomically.

Using our approach, algorithms can be designed for more complicated non-blocking query operations on data structures, such as range queries, nearest neighbour searches and predecessor queries on Patricia tries. Such a query can be done by reading a portion of the trie repeatedly until the same set of Nodes are obtained by two consecutive traversals. One must also check that the portion of the trie has not been removed. This can be done by checking that the Nodes are not flagged permanently. Since the implementation guarantees there is no ABA problem on the *child* field, the query operation that terminates can be linearized any time between the last two traversals. Such query operations do not interfere with any update operations.

Another area of future work is the design of shared cursors for doubly linked-lists. A new sequential specification would be required for such a list. A shared

cursor could be moved when an update occurs at the node where the cursor is located. Then, an operation called with a cursor would not need to call any routine similar to UpdateCursor to retrieve the location of the cursor at the beginning. A doubly-linked list with shared cursors might need less complicated memory management compared to our doubly-linked list implementation.

Our amortized analysis techniques in Section 8.1 can also be used to analyze the amortized complexity of other non-blocking data structures. We believe employing the potential technique simplified our analysis and this technique is appropriate for analyzing the amortized complexity of other concurrent data structures. To define a potential function for an implementation, it should be determined what operation causes extra work of another operation and then make some step(s) of the first operation pay for those extra step(s). However, this must be done in such a way that it is possible to bound the number of extra steps that an operation pays for.

Bibliography

[1] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14, pages 714–723, 2014.

[2] Hagit Attiya and Eshcar Hillel. Highly-concurrent multi-word synchronization. In *Proceedings of the 9th International Conference on Distributed Computing and Networking*, pages 112–123, 2008.

[3] Hagit Attiya and Eshcar Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems*, 52(4):729–762, 2013.

[4] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 261–270, 1993.

[5] Paul Bieganski, John Riedl, John V. Carlis, and Ernest F. Retzel. Generalized suffix trees for biological sequence data: Applications and implementation. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 35–44, 1994.

[6] Anastasia Braginsky and Erez Petrank. A lock-free B+tree. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.

[7] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings ACM Symposium on Principles and Practice of Parallel Programming*, pages 257–268, 2010.

[8] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 13–22, 2013.

[9] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming*, PPOPP '14, pages 329–342, 2014.

[10] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Proceedings International Conference on Principles of Distributed Systems*, OPODIS '11, pages 207–221, 2011.

[11] Daniel Cederman and Philippas Tsigas. Supporting lock-free composition of concurrent data objects. *IEEE Transactions on Computers*, 62(9):1866–1878, September 2013.

[12] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 332–340, 2014.

[13] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing*, pages 115–124, 2012.

[14] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, 2010.

[15] Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, July 1990.

[16] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.

[17] Min Gan, Mingyi Zhang, and Shenwen Wang. Extended negative association rules and the corresponding mining algorithm. In *Proceedings of the 4th international conference on Advances in Machine Learning and Cybernetics*, pages 159–168, 2006.

[18] Michael F. Goodchild. Geographic information systems and science: today and tomorrow. *Annals of GIS*, 15(1):3–9, 2009.

[19] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, PODC '02, pages 260–269, 2002.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop Workload Characterization*, pages 3–14, 2001.

[21] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.

[22] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Symposium on Distributed Computing*, pages 300–314, 2001.

[23] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, 2002.

[24] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[25] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[26] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[27] Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012.

[28] Jikuan Hu and Weiqing Wang. Algorithm research for vector-linked list sparse matrix multiplication. In *Proceedings of the 2010 Asia-Pacific Conference on Wearable Computing Systems*, APWCS '10, pages 118–121, 2010.

[29] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.

[30] Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive software transactional memory in Java. *Transactions on High-Performance Embedded Architectures and Compilers*, 5(2), 2010.

[31] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the 15th ACM Symposium on Parallel Algorithms and Architectures*, pages 314–323, 2003.

[32] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

[33] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[34] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[35] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[36] Rotem Oshman and Nir Shavit. The skiptrie: low-depth concurrent search without rebalancing. In *Proceedings of 32nd ACM Symposium on Principles of Distributed Computing*, pages 23–32, 2013.

[37] Matthias Pfeffer, Theo Ungerer, Stephan Fuhrmann, Jochen Kreuzinger, and Uwe Brinkschulte. Real-time garbage collection for a multithreaded Java microcontroller. *Real-Time Systems*, 26(1):89–106, January 2004.

[38] Andrea Pietracaprina and Dario Zandolin. Mining frequent itemsets using Patricia tries. In *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, 2003. (Available as CEUR Workshop Proceedings Series, Vol. 90, http://ceur-ws.org/vol-90).

[39] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings ACM Symposium on Principles and Practice of Parallel Programming*, pages 151–160, 2012.

[40] Niloufar Shafiei. Non-blocking Patricia tries with replace operations. http://arxiv.org/abs/1303.3626, 2012.

[41] Niloufar Shafiei. Non-blocking Patricia tries with replace operations. In *Proceedings of the 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 216–225, 2013.

[42] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[43] Håkan Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *International Journal of Parallel Programming*, 39(6):694–716, 2011.

[44] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

[45] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Proceedings of the 16th Annual Symposium on Principles of Distributed Systems*, pages 330–344, 2012.

[46] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[47] Jyh-Jong Tsay and H.-C. Li. Lock-free concurrent tree structures for multiprocessor systems. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 544–549, 1994.

[48] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, PODS '92, pages 212–222, 1992.

[49] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, 1995.

[50] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003. See http://babelfish.arc.nasa.gov/trac/jpf.