A CONCURRENT MOVE TO FRONT LIST WITH AN ADAPTIVE BOUND

SHALOM ASBELL

A Thesis submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of Master of Science

Graduate Program in Electrical Engineering
and Computer Science
York University, Toronto, Ontario

December 2025

Abstract

Few concurrent data structures adapt dynamically to access patterns, and those that do lack formal performance guarantees. This thesis introduces the first *self-adjusting* concurrent data structure with an adaptive bound analogous to those of optimal self-adjusting sequential structures. We present a lock-free *move to front* (MTF) list supporting a dynamic set of keys, where an operation *op* on a key k runs in an amortized number of steps proportional to the size of its *working set* plus a contention term. The working set of *op* is the set of keys accessed since the last operation on key k, and contention is the number of operations that run concurrently with *op*. We further show that the list performs at most twice as much work as the best possible fixed list for a set of searches, up to contention. Finally, we prove that the number of nodes that can be reached from shared memory is bounded by the number of keys in the set plus contention.

Acknowledgements

I would first like to express my sincere gratitude to my supervisor, Professor Eric Ruppert. This work would not have been possible without his time, patience, and steady mentorship. I am deeply grateful for the depth of computer science knowledge he has shared, and for the clarity of thought, rigor, and productivity he helped me develop throughout my research.

I also thank Professor Shahin Kamali for his encouragement, for his guidance to our research group, and for his thoughtful comments on this thesis. I am likewise grateful to Professor Neal Madras for serving on my oral examination committee and for his insightful feedback, which strengthened the final manuscript.

Thank you to my parents for their unwavering support throughout my undergraduate and graduate studies. To my four siblings, thank you for the energizing breaks you provided, and to my cousins, Eitan and Aliza, thank you for being a source of comforting friendship over this journey. I also extend my thanks to Carmel for her constant encouragement and support.

Thank you to my labmates in the Theory Group, and to my fellow graduate students, for the laughter, motivation, and stimulating challenges that have shaped me over the past few years. Finally, thank you to Argos+ for being with me at every step of this computer science journey.

Contents

List of Tables

List of Figures

Author's Declaration

Large Language Models were used solely to improve grammar and clarity, to provide occasional LaTeX formatting assistance, and to help locate reference materials.

Chapter 1

Introduction

Self-adjusting (*adaptive*) data structures dynamically adapt to access patterns to improve efficiency. They restructure themselves after each access, typically using heuristics that shorten the access path to the accessed object, so that future accesses to that object are faster. Over time, repeated restructuring ensures that frequently accessed objects have shorter access paths than those accessed infrequently. Moreover, because access sequences often access related items in close succession, such restructuring improves efficiency for near-future accesses. In real life, access sequences are rarely uniform, and many exhibit temporal locality [17].

Classic examples of sequential self-adjusting structures include self-adjusting trees like the *splay tree* [39] and self-organizing lists like the *move-to-front* (MTF) list [38], which have runtime bounds that capture the performance benefits of self-adjustment (see [29]). In the concurrent setting, making a structure adaptive is difficult because restructuring can disrupt correctness under interleaving operations, a challenge not present in the sequential case. As a result, concurrent self-adjusting structures remain limited: many use locks, preventing any step-complexity analysis, while existing lock-free designs do not have worst-case bounds that reflect adaptivity. This gap between the mature theory of sequential self-adjusting structures and the underdeveloped theory for their concurrent counterparts motivates this thesis.

Specifically, we ask:

> Can we design concurrent self-adjusting data structures for which we can prove upper bounds analogous to those of optimal sequential self-adjusting data structures?

This thesis takes an initial step toward addressing this challenge by introducing a concurrent *move-to-front* (MTF) list whose performance bounds closely approximate those of the sequential MTF list.

In the sequential setting, accessing a key k in the set takes $O(|w|)$ steps, where w is the set of distinct keys accessed since the previous access to k. These keys are called the working set for the access to k. In the concurrent setting, overlapping operations make this notion of a working set ambiguous; we therefore define the *concurrent working set*.

We assign each operation a specific moment during its execution, called its *linearization point*, at which it is considered to take place, and we define the working set using these moments together with the execution intervals of operations. We denote the working set of a successful access *op* by $w(op)$ (see Section 5.1 for a formal definition).

In our lock-free MTF list, an operation *op* on key k with linearization point ℓ completes in amortized $O(|w(op)| + \dot{c}(op))$ steps if k is present at ℓ, where $\dot{c}(op)$ denotes the point contention of *op* (see Section 1.1.2). We further show that our list performs at most twice as many steps as the optimal fixed list serving the same set of searches, even under adversarial scheduling, up to a contention term. Finally, we show that in any configuration, at most $O(n + \dot{c})$ nodes can be reached from shared memory, where n denotes the size of the set represented by the list, and $\dot{c}$ is the maximum point contention over an execution (see Section 1.1.2). This ensures that the total shared-memory footprint grows linearly with the number of keys plus contention.

1.1 Model and Preliminaries

We assume a classical asynchronous shared-memory model with $m \geq 2$ processes operating on a shared list. Processes communicate by performing *read*, *write*, and *compare-and-swap* (CAS) instructions (defined in Section 1.1.3) on shared variables. The system is assumed to be crash-prone: at any time, a subset of the m processes may fail, while the remaining processes continue to execute.

A configuration is the complete global state: the values of all shared variables together with the local state of every process.

Definition 1. *An execution e is a finite or infinite alternating sequence of configurations and steps,*

$$e = (C_0, s_1, C_1, s_2, C_1, \ldots)$$

where C_0 is the initial configuration and C_i (for $i \geq 1$) is the configuration that results after the i^{th} step s_i has been applied.

1.1.1 Lock-Freedom and Amortized Step Complexity

Lock-based concurrent algorithms can suffer from system-wide starvation if a process fails to release a lock. *Lock-free* algorithms avoid this problem by guaranteeing system-wide progress, even if some processes are halted or delayed. Despite their name, "lock-free" does not mean free of locks; it describes a progress guarantee in which at least one operation always completes within a finite number of steps. Lock-free algorithms allow us to bound amortized step complexity by attributing the wasted steps of processes that fail to make progress to operations that do. The technique of charging inexpensive operations extra steps to cover the cost of more expensive ones is known as *amortized analysis*, first introduced by Tarjan et al. for sequential data structures [41]. For a discussion of amortized analysis in the study of lock-free data structures, see [37].

1.1.2 Contention

Contention refers to the number of operations that are running concurrently with an operation. It is formalized by two related measures: *point contention* and *interval contention*. The point contention of an operation op, denoted $\dot{c}(op)$, is the maximum number of operations that are active at any single time during the execution of op. We write $\dot{c}$ for the maximum point contention over an execution. The interval contention of op, denoted $\bar{c}(op)$, is the total number of operations that run during the execution of op.

Over any execution, the sum of all operations' interval contention is at most twice the sum of all operations' point contention [22]. Hence, if each operation op incurs a cost of $O(\bar{c}(op))$, then amortizing the total interval-contention cost across all operations yields a per-operation amortized cost of $O(\dot{c}(op))$.

1.1.3 Compare-and-Swap (CAS)

Atomic operations are machine instructions that read from or write to memory in a single, indivisible step, without interference from other instructions. Lock-free algorithms typically rely on *read–modify–write* (RMW) primitives: atomic operations that read a memory location and conditionally update it in the same indivisible step. *Compare-and-swap* (CAS) is a RMW primitive widely used for implementing lock-free data structures (see [16, 20, 24, 34]). It is supported by the x86 architecture via the `CMPXCHG` (compare-and-exchange) instruction and by the Arm architecture via the `CAS` operation introduced in the Armv8.1-A instruction set (see [27, 32]).

The ubiquity of CAS stems from its ability to achieve consensus among an arbitrary number of processes [25]. In a CAS-based lock-free algorithm, a process updates a shared memory word X by reading its current value into a local variable *old*, computing a desired replacement value *new*, and then executing $X.\text{CAS}(old, new)$. The CAS instruction atomically installs *new* if the current value of X equals *old*; otherwise, the CAS instruction does not modify X. The latter may occur if the value in X was modified by a contending process between the read and the CAS. In all cases, CAS returns the value it read from X before attempting an exchange.

1.1.4 Sequential Specification of a Set

A *sequential specification* defines the abstract state and behavior of a data object. It specifies, for each operation, and abstract state of an object, the new abstract state that the operation moves the object into and the return value produced by the operation. In proofs of correctness for concurrent objects, sequential specifications are used to define the sequential behavior a correct concurrent implementation must simulate.

The notion of a sequential specification is best illustrated by example. Consider a *set* object that maintains a dynamic set of keys.

Definition 2. *Let S denote the abstract state of a set before an operation is applied, and let S' denote the state after the operation completes. The sequential specification of the set is defined as follows:*

- SEARCH*(k) returns true if and only if k is in S. The resulting state S' is the same as the initial state; that is, $S' = S$.*
- INSERT*(k) returns true if and only if k is not in S. The resulting state S' is obtained by adding k to S; that is, $S' = S \cup \{k\}$*
- DELETE*(k) returns true if and only if k is in S. The resulting state S' is obtained by removing k from S; that is, $S' = S \setminus \{k\}$.*

1.1.5 Linearizability

To ensure the correctness of our MTF list, we adopt *linearizability*, a strong correctness condition for shared objects [26]. At a high level, a shared object is *linearizable* if its operations appear to take effect in a legal sequential order, even when they overlap in time.

Formally, to prove a concurrent execution of operations on a shared object is linearizable, each operation that has completed, and possibly some that have not completed, can be assigned a distinct point in its execution interval–its *linearization point*–at which it is considered to take effect atomically. The execution is *linearizable* if its operations return results consistent with the object's sequential specification, as if the operations had been executed sequentially in the order defined by their linearization points.

In Chapter 4, we prove every execution on our MTF list satisfies linearizability with respect to the sequential specification of a set (as defined by Definition 2).

Chapter 2

Related Work

2.1 Online Algorithms

Online algorithms process requests as they arrive, without knowledge of future inputs [5]. Each decision made to handle a request is irrevocable, and made based only on the information available at the time of the request, with the goal of maintaining long-term efficiency as new requests appear. This framework captures the essence of real-time reactive computation, where performance depends on adaptability under uncertainty.

The efficiency of an online algorithm is typically evaluated using *competitive analysis*, which compares its performance on a sequence of requests against that of an optimal *offline* algorithm with full foresight of the sequence. More formally, let $Cost_\sigma(\mathrm{ALG})$ denote the total cost incurred by an online algorithm ALG on an input sequence σ, and let $Cost_\sigma(\mathrm{OPT})$ denote the minimal possible cost achievable by an optimal offline algorithm on σ. The algorithm ALG is said to be *C-competitive* if there exists a constant $\alpha \geq 0$ such that

$$Cost_\sigma(\mathrm{ALG}) \leq C \cdot Cost_\sigma(\mathrm{OPT}) + \alpha \quad \forall \text{ input sequences } \sigma.$$

The ratio C quantifies how well the online algorithm performs relative to the offline optimum, providing a formal and robust measure of adaptability despite the lack of future knowledge. The lower the ratio, the better the relative performance.

Sequential self-adjusting lists, such as the MTF list, are classic online algorithms: they respond to each access and restructure themselves dynamically to improve the efficiency of future accesses.

2.2 Sequential Self-adjusting Lists

Self-adjusting lists have been extensively studied in the sequential setting. A variety of restructuring heuristics have been proposed and analyzed, including *Move-to-Front* (MTF) [31], *Transpose* [36], *Frequency Count* [36], *TimeStamp* [4], and *Move-to-Recent-Item* (MRI) [19], each named after the restructuring rule they apply. Many of these heuristics have been shown to perform within a constant factor of the optimal offline self-adjusting list, making them suitable candidates for evaluation in concurrent environments (see [29] for a comprehensive survey).

While many of these heuristics would be worth exploring in the concurrent setting, our list is based on MTF. We chose MTF because of its strong theoretical guarantees and because it is easier to implement under concurrency: each access moves the key to the front, avoiding complex changes to the middle of the list.

Sleator and Tarjan proved that MTF is 2-competitive with the optimal offline list for any sequence of searches, under a cost model in which moving an accessed key directly to the front of the list is free (a "free exchange"), while swapping adjacent keys incurs unit cost (a "paid exchange") [38]. Since MTF is 2-competitive with the optimal offline list, it is also 2-competitive with the optimal offline *static* list, that is, a fixed list whose keys are ordered in decreasing order of access frequency before serving the sequence. In Chapter 6, we show that our lock-free MTF list achieves the same 2-competitiveness against an optimal offline static list for an adversarially scheduled set of searches, up to a contention term.

MTF has been shown to perform particularly well on access sequences with high temporal locality [8]. The time to access to a key k in a MTF list is upper bounded by the access's *working set* size–the number of distinct keys accessed since the last access of k [18]. This inherently captures the shortened access path for recently accessed keys. Albers et al. [6] introduced a function to formalize locality of reference in an access sequence σ in the context of paging. Specifically, for any contiguous subsequence (window) of length τ in σ, the number of unique keys accessed in that window cannot exceed $f(\tau)$. Angelopoulos et al. [9] proved that MTF is uniquely optimal under bijective analysis for access sequences where $f(\tau)$ is concave. Albers and Lauer [7] reinforced this result using an alternative measure of locality based on the number of *runs* in a sequence, where a run is defined as the number of times a key k is accessed consecutively in a subsequence σ_{kl} obtained from an access sequence σ by retaining only accesses to the key k and l. They showed that for sequences containing a high ratio of "long runs" (multiple consecutive accesses) to "short runs" (single appearances), the

MTF algorithm achieves a competitive ratio of 1. This theoretical backing positions MTF as particularly attractive for applications in which access sequences are known to possess temporal locality.

2.3 Online Analysis in the Shared-Memory Setting

In the sequential setting, online analysis is simple and symmetric: both the algorithm under study and the optimal offline algorithm (OPT) process the same access sequence in lockstep. Each access begins and completes before the next one starts, ensuring that both algorithms execute identical workloads under identical conditions. This symmetry enables a fair and meaningful comparison between their performances.

Concurrency fundamentally disrupts this fairness. In shared-memory systems, operations may overlap, stall, or fail to complete due to contention, crashes, or preemption. If both algorithms are forced to execute a strictly ordered access sequence, concurrency would effectively vanish, collapsing the analysis to the sequential case. Conversely, allowing operations to overlap arbitrarily introduces nondeterminism: identical logical access sequences may yield distinct interleavings and outcomes, undermining fairness and the notion of "identical workloads."

Early efforts to extend online analysis to distributed systems avoided the complexity of a concurrent optimal by comparing distributed algorithms to optimal sequential ones [12, 13]. Ajtai et al. [2] were the first to formalize a competitive framework for distributed algorithms in which the adversary could schedule the optimal algorithm concurrently. To address nondeterministic scheduling, they defined an algorithm as k-competitive with the optimal offline algorithm if, for every possible schedule of requests, the algorithm takes at most $O(k)$ times more steps than the optimal offline algorithm under the same schedule.

More recently, Boyar, Ellen, and Larsen [15] proposed the *fully adversarial model*, in which each process in the system is assigned its own sequence of accesses, and the adversary may schedule both the algorithm and the optimal offline arbitrarily, subject only to the constraint that each process executes its local sequence in order.

To avoid the ambiguities that concurrency introduces into online analysis, we compare our list to a static offline optimal algorithm. Both algorithms execute the same global set of searches on a fixed key set, but only OPT may initialize its data structure in the configuration most favorable to the upcoming access pattern. After initialization, the structure of OPT's list

cannot be modified. We permit the adversary to interleave the steps arbitrarily, for both our list and the static optimal list. We can do so since the cost of the static optimal is invariant to scheduling or contention. This model of comparison decouples our list's performance from timing events, providing a clean analytical foundation that extends classical online analysis to shared-memory systems. Our adversarial model, which we call the *global adversarial model*, is stronger than the fully adversarial model of Boyar, Ellen, and Larsen [15], as it allows the adversary to schedule a single global set of operations arbitrarily, rather than merely interleaving predetermined per-process schedules.

2.4 Concurrent Self-Adjusting Data Structures

Afek et al. [1] introduced the first concurrent self-adjusting tree. They began by designing a new sequential self-adjusting tree, called the CB-tree, which achieves the same asymptotic performance bounds as the splay tree [39] while limiting each operation to a constant number of rotations. This design aimed to reduce contention by minimizing the likelihood of multiple operations performing rotations at the same location. Building on this, they developed a concurrent version of the CB-tree. However, their approach relies on locks, leading to unbounded step complexity, and requires each node to maintain a counter tracking how often it is accessed. To improve throughput, Afek et al. update these counters using writes, but this strategy can negate the performance benefits of self-adjustment because it allows for a stale update to overwrite a node's access count with a much smaller value.

More recently, Aksenov et al. [3] addressed an open question of Afek et al. [1] by extending self-adjustment for concurrent data structures to skip-lists. Like the concurrent CB-tree, their approach also relies on locks, resulting in unbounded step complexity.

Tan et al. [40] made the first attempt at a lock-free self-adjusting list, but their algorithm has issues with efficiency, and they provide no proof of correctness. They maintain two copies of their list at all times, one of which is self-adjusting, and the other a non-self-adjusting list whose nodes are sorted by key. A lookup traverses through the self-adjusting list first, and if no node containing the key is found, either due to its absence from the set, or a concurrent adjustment that resulted in a miss, the non-self adjusting-list is traversed as well. In the worst case, this requires iterating through all of the keys in the set twice. Furthermore, the removal scheme of their list is based on the removal scheme of Michael [33], which can require repeated re-traversals through the list in the case of contention on a removal, resulting in poor amortized complexity (see [20] for a detailed analysis of Michael's list).

Fu and Zhang [21] made another attempt at a lock-free self-adjusting list, by appending node representing operations to their list in an attempt to store a partial history of the list, but their design has core issues, stemming from using writes to update edges directly, that result in space and time bounds that depend on the total number of operations on the list. A worst-case sequence for their list is shown in Appendix A.

Boyar, Ellen, and Larsen [15] proposed a lock-free move-to-front (MTF) list for a fixed set of keys (i.e., only SEARCH operations are supported), intended mainly to enable competitive analysis in the concurrent setting. Each process has a dedicated slot in a shared *announce array*, where it records the key it is searching for before traversing the list. When an operation on key k reaches the node with key k, it scans the announce array for processes also searching for key k, writes the result of the lookup into their slots, and then moves the node with key k to the front. This ensures that concurrent operations looking for key k learn the correct result even if the node with key k is moved to the front while they are still traversing.

Because their algorithm only supports a fixed key set, it cannot be used for general cases that require a dynamic key set. They also do not provide a worst-case bound per operation. In Appendix B, we show that the amortized step complexity of each SEARCH on their list is $\Omega\big(|w(op)| + p\big)$, where p is the number of processes. Since $\dot{c} \leq p$, our list asymptotically outperforms theirs while also supporting a dynamic key set.

2.5 Lock-Free Singly-linked Lists

There are two primary challenges that lock-free lists need to overcome to support concurrent removals, both of which result from concurrent updates to the incoming and outgoing edges of a node. The first of these challenges, is ensuring the predecessor of a node being removed is not removed concurrently (see Figure 2.1), which would result in the node remaining in the list. The second of these challenges is ensuring the predecessor of a node being inserted is not concurrently removed from the list, which would result in the inserted node becoming invisible to future accesses (see Figure 2.2).

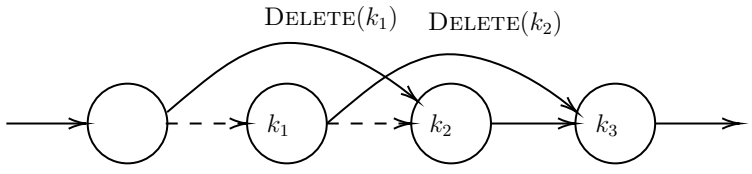

Figure 2.1: Concurrent DELETE operations performed on a list resulting in k_2 remaining in the list.

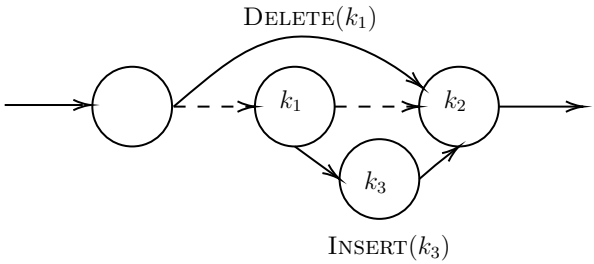

Figure 2.2: Concurrent INSERT and DELETE operations performed on a list, resulting in k_3 becoming unreachable by future lookups.

Harris [23] addressed these challenges by storing a *mark* bit in the *next* pointer of each node in the list that represents if it is marked for removal or not. To update the *next* pointer of a node, a CAS is performed with the old *next* pointer, and the *mark* bit set to zero. The CAS fails if the *next* field of the node was changed, or the node has been marked for removal. This ensures the outgoing edge of a node is never updated concurrently with its removal from the list. A removal or insertion of a node x that fails at updating the *next* pointer of x's predecessor re-traverses from the head of the list to find the predecessor of x. Michael [33] extended Harris's list to make it compatible with safe memory reclamation. Fomitchev and Ruppert [20] showed that this re-traversal can result in operations on Harris's list taking $\Omega(\max(n) \cdot \dot{c})$ steps, where $\max(n)$ is the maximum number of elements in the list during an execution, and $\dot{c}$ is the maximum point contention.

Valois [42] introduced backlinks–pointers to a node's predecessor–to facilitate node removal. Backlinks allow an operation to backtrack to find an unmarked predecessor of a node whose predecessor has been marked by following the backlink, rather than restarting from the head of the list. A chain of backlinks can be followed from a node x under removal to find the first node that precedes x that is not under removal. Fomitchev and Ruppert [20] showed that a chain of backlinks can be re-traversed multiple times by the same process. This occurs when the backlink of a marked node x is set to a node in a chain of backlinks that has already been traversed by an operation attempting to update the outgoing edge of x, causing the operation to retraverse the entire chain. As a result, each operation can take $\Omega(m)$ steps, where m is the total number of operations in the sequence.

Fomitchev and Ruppert resolved this issue by flagging the incoming edge of a node x slated for removal, ensuring the outgoing edge of x's predecessor remains fixed until the removal completes. A node cannot be both flagged and marked; thus, if an operation reads the predecessor of a node being removed and finds it flagged, it helps complete the removal before proceeding. This approach ensures safe concurrent updates and achieves an amortized $O(\dot{c})$

overhead for handling removals.

2.6 Version Lists

Version lists are concurrent linked lists that maintain timestamped versions of an object, enabling queries to obtain atomic snapshots while remaining concurrent with updates [43]. Both version lists and our MTF list insert new nodes exclusively at the head and perform deletions anywhere. Since insertions occur only at the head, removals can cooperatively unlink entire chains of nodes.

Our removal mechanism (see Section 3.3.3) resembles that of Blelloch et al. [43] for their version list. Their algorithm uses a doubly-linked list in which each node maintains a forward pointer (*right*) to its successor and a backward pointer (*left*) to its predecessor. These symmetric links let a remove operation identify both neighbors of a marked node without further traversal. A node to be removed is first marked to indicate logical deletion, after which a scan is performed in the forward and backwards directions–following *left* and *right* pointers–to find the nearest unmarked predecessor and successor. It then performs two CAS operations: one updates the *right* neighbor's *left* pointer and the other the *left* neighbor's *right* pointer to splice out the deleted region. If either CAS fails or a neighbor becomes marked, the operation retries outward until it succeeds.

In contrast, our MTF list remains singly linked and adds backward pointers, or *backlinks*, only as needed. When a node x is removed, the remover sets $x.backlink$ to its predecessor during the forward traversal. This lazy-backlink design postpones backlink assignment until removal and uses only a constant number of additional CAS steps to ensure backlinks are never shortened. As a result, it provides the same lock-free safety and progress guarantees as Blelloch et al.'s version list [43], but with simpler structural maintenance.

2.7 Lower Bounds

Attiya and Fouren [11] showed that each operation *op* on a lock-free implementation of a dynamic set using *read-modify-write* (RMW) primitives (e.g., CAS or Test&Set) must incur a cost of $\min\big(\dot{c}(op), \log\log p\big)$ where p is the number of processes and $\dot{c}(op)$ is the point contention of *op*. This implies that when contention is small relative to the number of processes (i.e., $\dot{c} \in O(\log\log p)$), the $\dot{c}(op)$ term in our amortized step–complexity bound is unavoidable.

Chapter 3

Lock-Free Move-To-Front (MTF) List

3.1 High-Level Overview

Performing a *move-to-front* (MTF) operation on a concurrent list poses several challenges. The core issue, touched upon by [15] and [21], is that MTF operations must appear atomic with respect to concurrent searches and deletions to maintain correctness. Without such atomicity, two problems can occur: a lookup might miss a key moved by another concurrent lookup to the front of the list (see Figure 3.1), and a key removed by an operation may be wrongly reinserted by a concurrent MTF (see Figure 3.2).

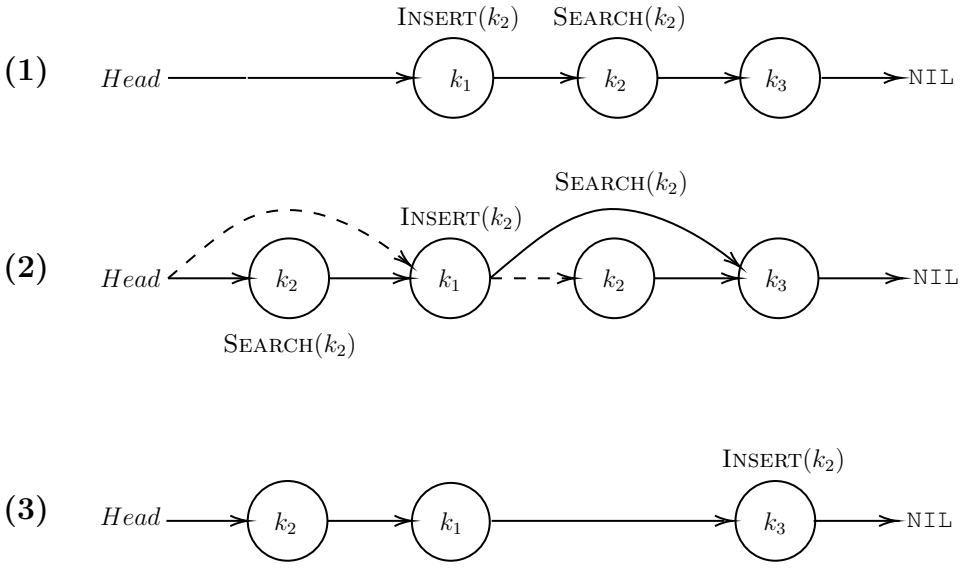

Figure 3.1: A key k_2 becoming invisible to a concurrent lookup. In **(1)**, INSERT(k_2) dereferences the head and pauses while SEARCH(k_2) locates k_2. In **(2)**, upon finding k_2, SEARCH(k_2) performs a move-to-front (MTF) update (dashed arrows indicate stale edges) and completes. Finally, in **(3)**, INSERT(k_2) resumes and reaches the node with key k_3, missing k_2.

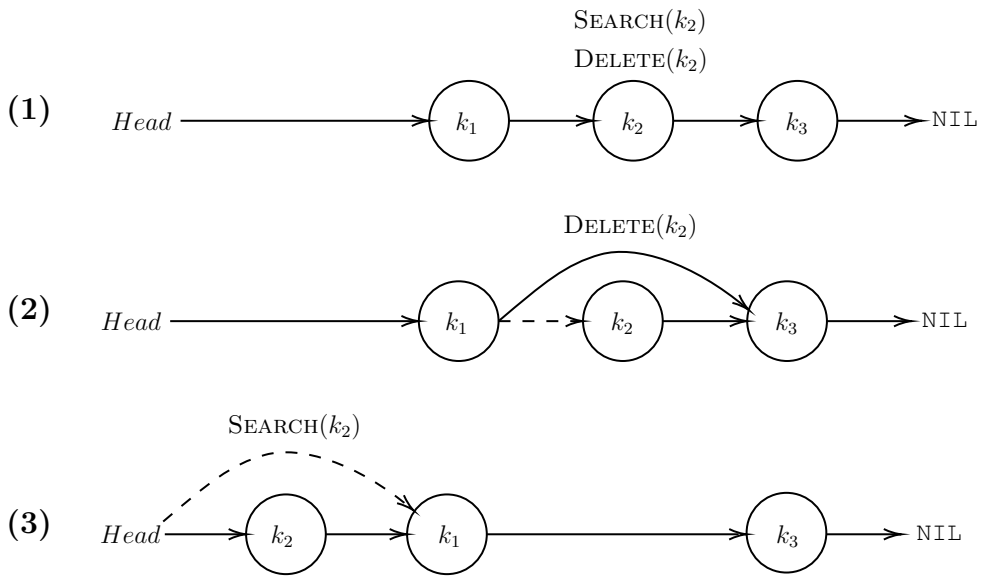

Figure 3.2: A key k_2 incorrectly re-inserted after deletion. In **(1)**, SEARCH(k_2) and DELETE(k_2) reach the node with k_2. In **(2)**, DELETE(k_2) removes k_2 and returns `true`. Finally, in **(3)**, SEARCH(k_2) resumes and performs its move-to-front (MTF) step, appending a new node with k_2 at the head, incorrectly re-inserting k_2.

To address these challenges, we store a partial history of operations in our list by adding nodes for ongoing operations to the list in addition to nodes that represent inserted keys. Our approach has some similarity to the approach of Fu and Zhang [21] (as described in Section 2.4), but they do not limit the additional operation nodes to ongoing operations, resulting in worst-case space and time bounds proportional to the total number of operations (see appendix A). Our approach is also simpler than Fu and Zhang's: we add only one node per operation at the head of the list, rather than one to three.

Each operation on our list adds a node with its operation type and key to the head of the list. Consequently, nodes can be of three different types: insert, search, and delete. For now, assume nodes are not removed, so that the list retains a complete history, and we linearize operations when they add their node. To determine key membership, an operation on the key k traverses forward from the node it added to find nodes with a matching key: if it encounters an insert node with key k before a delete node with key k, then k is in the set; otherwise it is absent. Search nodes can be ignored during the traversal, but they play an important role in our MTF behaviour. Specifically, if a SEARCH(k) operation determines that the key k is present when it added its node, it promotes the search node it added to an insert node, effectively moving k towards the front of the list, and shortening the access path for future accesses of k.

By appending operation nodes to the list, and never removing any, we eliminate the issues shown in Figure 3.1 and Figure 3.2. The problem in Figure 3.1 is solved because a successful lookup no longer removes nodes when moving a key to the front; instead, it simply promotes the node it appended to an insert node. The problem in Figure 3.2 is also solved: if a concurrent SEARCH(k) or INSERT(k) adds its node before a concurrent DELETE(k), it will promote its node to an insert node rather than adding a new one at the front of the list. Conversely, if it appends its node after the DELETE(k), it will encounter a delete node with the key k during its traversal and correctly conclude that k is not in the set.

While storing a total history of operations in our list resolves these issues, it introduces a harmful problem: the list grows with every operation, potentially leading to much longer traversals. In the worst case, a lookup for a key accessed only once before at the very beginning of an execution must traverse $\Theta(m)$ nodes, where m is the total number of operations on the list. When $m \gg n$, where n is the number of keys present at the access's linearization point, $\Theta(m)$ is worse than the upper bounds of non-self-adjusting concurrent lists. For example, the list of Fomitchev and Ruppert achieves an amortized lookup time $O(n + \dot{c})$, where $\dot{c}$ is the maximum point contention [20].

Thus, an efficient concurrent self-adjusting list *must* remove nodes from the list, to keep its length proportional to the number of keys in the set. Therefore, each operation on our list that does not insert a new key either removes some nodes or tells a concurrent operation to remove them on its behalf. Operations remove only nodes that no longer matter to ongoing or future work. If removing a node could change the affect the result of an ongoing operation *op*, the remover first informs *op* of its result by setting a flag in the node added by *op*.

A failed SEARCH(k) operation removes the search node x it added to the list because operations ignore search nodes in their traversals. A successful SEARCH(k) or INSERT(k) operation removes the first search or insert node w with the key k reached from the node x it added, provided the operation that added w has determined its result. Otherwise, it tells the operation that added w to remove w. Before removing w, or telling the operation that added w to remove it, the successful SEARCH(k) or INSERT(k) promotes x to an insert node.

Although this may seem to reintroduce the problem shown in Figure 3.1, where a successful SEARCH(k) or INSERT(k) removes a node representing the key k from the traversal path of an operation for k, it does not (see Figure 3.3). If an operation on key k begins after a successful SEARCH(k) or INSERT(k) has removed an insert node with key k from its traversal path (and before the next DELETE(k)), it will still encounter an insert node. This is because

SEARCH(k) promotes its own node x to an insert node before performing the removal, and the node added by INSERT(k) is already an insert node. An operation on k that runs concurrently with a SEARCH(k) (and begins before the next DELETE(k)) may traverse past x while it is temporarily a search node, but it can later refer back to x to check its type: once x is observed as an insert node, the operation concludes that k is present and returns `true`. We will show that node removals performed by successful SEARCH and INSERT operations never cause a key to become invisible to concurrent or future lookups.

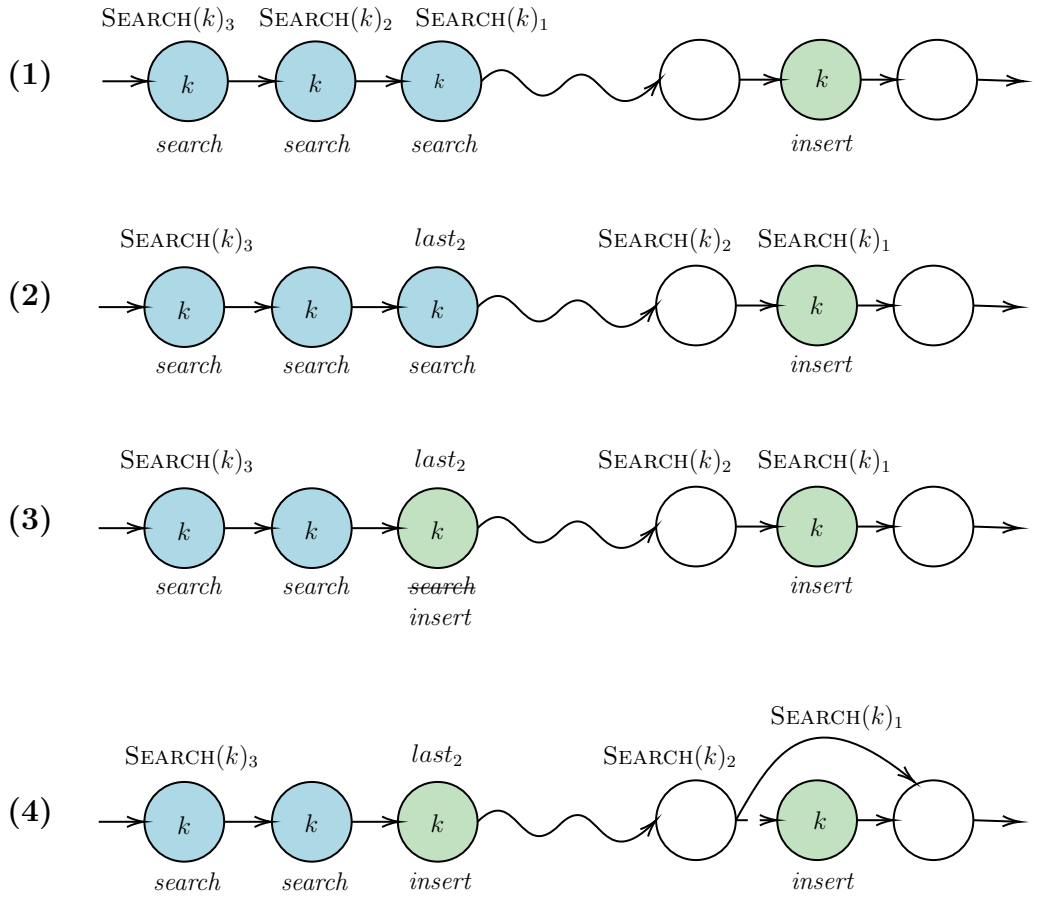

Figure 3.3: Three concurrent Search(k) operations on our list. All nodes with key k are shown and the squiggly arrow represents a chain of nodes. In **(1)**, all three operations append their nodes and begin traversing. In **(2)**, SEARCH(k)$_1$ reaches a matching insert node, while SEARCH(k)$_2$ reaches its predecessor and saves the last node with k in a local variable $last_2$. In **(3)**, SEARCH(k)$_1$ promotes its node to an insert node. In **(4)**, SEARCH(k)$_1$ removes the insert node it encountered and returns `true`; SEARCH(k)$_2$ then checks the type of $last_2$ and also returns `true`. SEARCH(k)$_3$ can reach the new insert node added by SEARCH(k)$_1$.

A DELETE(k) removes all *insert* and *search* nodes with key k that it reaches from the node x

it added, up to (but not including) the first node it encounters that was added by another DELETE(k) operation, or until it reaches the end of the list if no such node exists, and then removes x. This ensures that any operation on key k that adds a node after the DELETE(k), but before the next INSERT(k), does not encounter an *insert* node with key k after x has been removed (see Figure 3.4). While x remains in the list, such operations instead encounter x and return `false`.

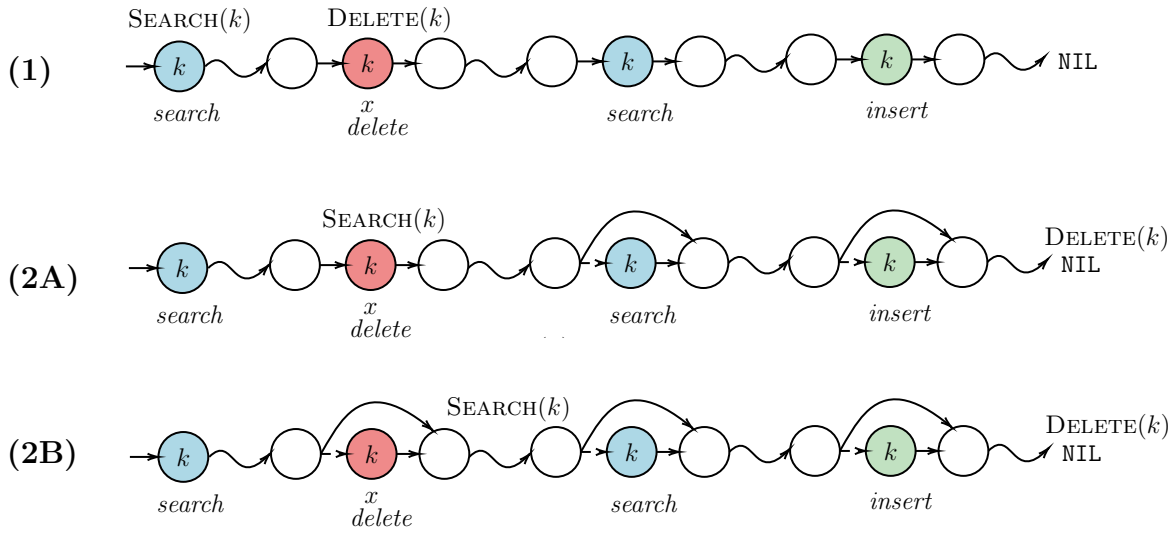

Figure 3.4: A Delete(k) removing all reachable nodes with key k. This example illustrates how a DELETE(k)'s removal procedure preserves correctness for a SEARCH(k) that adds its node after the DELETE(k) but before the next INSERT(k). The initial state is shown in **(1)**. From this state, two outcomes are possible: in **(2A)**, the SEARCH(k) encounters x and correctly returns `false`; in **(2B)**, after the DELETE(k) removes all matching nodes and then x, the SEARCH(k) reaches the end of the list and also returns `false`.

Consider three operations: INSERT(k) adds node z, SEARCH(k) adds node y, and DELETE(k) adds node x, in that order (with no other nodes for k added). If the DELETE(k) removes z, the SEARCH(k) may incorrectly conclude that k was not in the set. Thus, prior to removing z, the successful DELETE(k) informs the operation that added y of its result by setting an indicator field in y (see fig. 3.5).

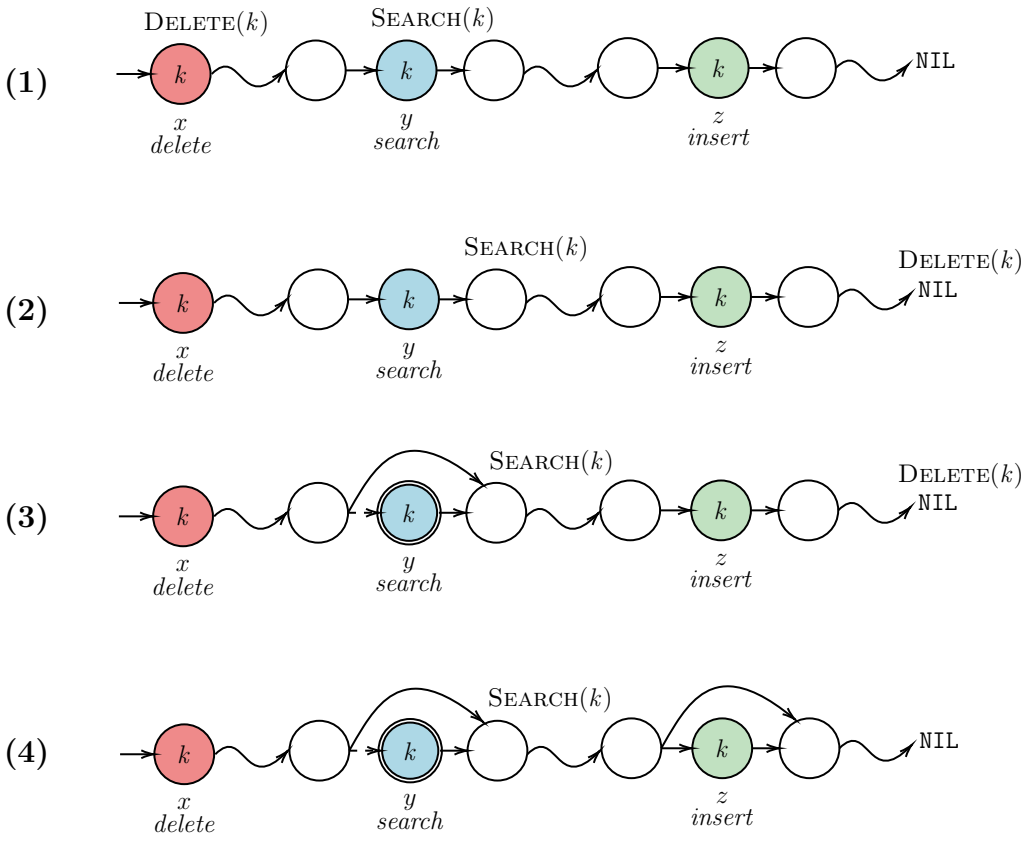

Figure 3.5: A successful Delete(k) informing a concurrent Search(k) before removing its node. The initial state is shown in **(1)**, with the DELETE(k) visiting the node x it claimed, and the SEARCH(k) visiting the node y it claimed. All nodes for the key k are shown. In **(2)**, the DELETE(k) traverses to the end of the list, reading the insert node z with key k, while the SEARCH(k) pauses before reaching z. In **(3)**, the DELETE(k) sets an indicator in y (depicted by the node with a double outline) before removing y and completes. In **(4)**, the SEARCH(k) wakes up and reads its node and correctly determines that k is in the set when it added y.

The removal routine of DELETE(k) operations explains why SEARCH(k) and INSERT(k) only remove nodes added by operations that have determined their results. Consider four operations on key k: an INSERT(k) adds z, a second INSERT(k) adds y_1, a SEARCH(k) adds y_2, and a DELETE(k) adds x, in that order, with no other nodes for k added. If the SEARCH(k) removes y_1 while the DELETE(k) is still in progress, the DELETE may never encounter y_1 and therefore be unable to update y_1's indicator field. If the DELETE misses y_1 and then removes all nodes with key k it reached, it can remove z before it is seen by the INSERT(k) that added y_1, causing that INSERT to incorrectly return `false` (see Figure 3.6).

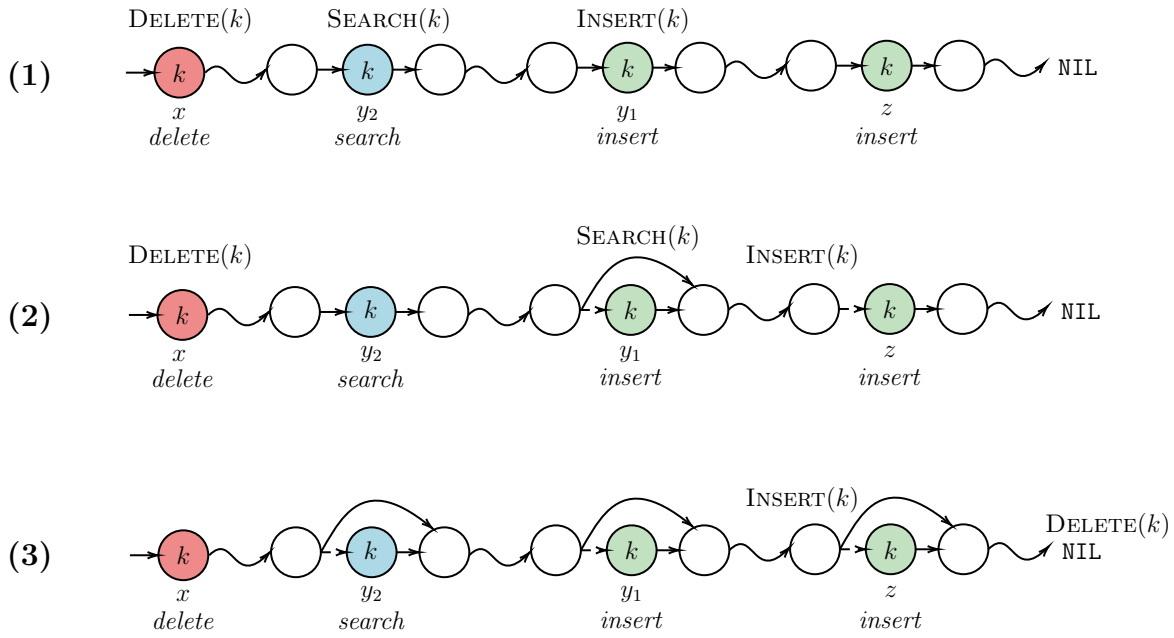

Figure 3.6: Search(k) removing the node of an ongoing operation. This example illustrates the hazard of a SEARCH(k) removing a node added by an operation that has not determined its result. In **(1)**, active DELETE(k), SEARCH(k), and INSERT(k) operations appear above the nodes they added; all nodes for the key k are shown. In **(2)**, the INSERT pauses after reaching the predecessor of z, while the SEARCH removes y_1 and the DELETE waits at x. In **(3)**, the DELETE resumes and traverses the list, missing y_1, and removes all reachable nodes with k, including z which should have been visible to the INSERT, which now traverses to the tail and incorrectly returns `false`.

To address the challenges of concurrent node removals in a linked-list (as described in Section 2.5), we combine Harris's edge-marking technique with Valois's backlinks [23, 42]. In Harris's marking technique, the outgoing edge of a node is marked before the node is removed, preventing concurrent updates to the same edge. Valois's backlink method complements this by storing, in each marked node, a pointer to a node previously observed to be its predecessor. This makes it easier to locate an unmarked predecessor if the predecessor of a node to be removed has itself become marked for removal.

We allow chains of marked nodes to be removed in a single operation, rather than one node at a time similar to Blelloch et al. [43] (see Section 2.5). To avoid repeatedly traversing the same sequence of backlinks, a process that skips over a chain in the forward direction creates a new backlink that skips over the chain in the backward direction. This new backlink serves as a shortcut to an unmarked node for future removal operations.

3.2 Structure and Evolution of the List

3.2.1 List Structure

Our MTF list consists of NODE objects, each of which stores various fields that collectively define the state of the list, and provide information for operations to determine their results and perform safe removals (see Figure 3.7).

```
 1. construct MTFList
 2. |  Node Head          ▷ points to the head of the list

 3. construct OpData
 4. |  Key key            ▷ key associated with the operation
 5. |  enum Type {search, insert, delete} type   ▷ the operation type

 6. construct Link        ▷ both fields stored in a single memory word
 7. |  Node next          ▷ pointer to the next node in the list
 8. |  Boolean mark       ▷ deletion flag

 9. construct Node (init: data←NIL, backlink←NIL, end←−1, remove←false)
10. |  OpData data        ▷ operation data
11. |  int serial         ▷ serial number
12. |  Link link          ▷ stores the node's mark status and its next node
13. |  Node backlink      ▷ pointer to previously observed predecessor
14. |  int end            ▷ serial number of a matching node (used in helping)
15. |  Boolean remove     ▷ flag indicating whether the node should be removed
```

Figure 3.7: The data objects of the MTF list.

The list is accessed through a global variable *Head*, which stores the *sentinel* node representing the head of the list.

Definition 3. *The node in Head is called the* ***sentinel****.*

Each node maintains a word-sized LINK object that stores the pointer to its successor. A node x *points* to another node via the *next* field of its LINK object $x.link$.

Definition 4. *A node x* ***points*** *to a node y in configuration C if $x.link.next = y$.*

A node y is said to be *reachable* from x if it can be obtained by repeatedly following *next* pointers starting at x (see Figure 3.8).

Definition 5. *Let $R(x, C) = (x_0, x_1, \ldots, x_r)$ be a sequence of nodes such that $x_0 = x$ and, for all $0 \leq i < r$, $x_i.link.next = x_{i+1}$. A node y is **reachable** from x in configuration C if $y \in R(x, C)$.*

The nodes reachable from $Head$ in a configuration are the nodes *in the list* in that configuration.

Definition 6. *A node x is **in the list** in configuration C if $x \in R(Head, C)$.*

Each LINK object also packs a Boolean flag, *mark*, together with its *next* pointer within its word. Nodes are created *unmarked* ($link.mark =$ `false`); setting $mark =$ `true` marks a node for removal. Storing *next* and *mark* in a single word enables atomic CAS operations on the pair $(next, mark)$. Because both fields are updated atomically, the *next* field of a marked node cannot be modified, ensuring that the next pointer of a node marked for removal remains unchanged (Figure 2.1 and Figure 2.2 show the problematic scenarios this avoids).

The least significant bit of LINK encodes the *mark* flag, while the remaining bits encode *next*. Since pointers on modern architectures are aligned to with word-sized (even) memory addresses, their least significant bit is always 0; hence, the remaining bits of the word are enough to represent *next*.

Important Note. For brevity, we write ***x.key***, ***x.type***, ***x.next***, and ***x.mark*** as shorthand for $x.data.key$, $x.data.type$, $x.link.next$, and $x.link.mark$, respectively.

To facilitate node removals, each node x has a *backlink* field that references a previously known unmarked predecessor of x. Backlinks help removals locate an unmarked predecessor of a node x marked for removal when its current predecessor becomes marked. Figure 3.8 shows a chain of links and backlinks in the list.

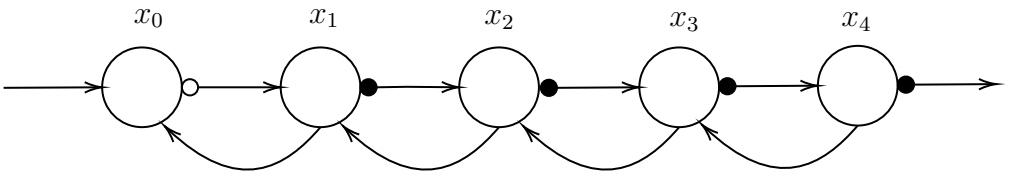

Figure 3.8: A chain of links and backlinks. Each small circle with its outgoing edge represents the LINK object of the node it is attached to; the circle is its Boolean *mark* (black = `true`, white = `false`) and the edge its *next* pointer. A node's backlink is represented by an arrow emerging from its bottom.

3.2.2 Initialization

Initially, the list is composed of two dummy nodes: one stored in *Head* and a *tail* node representing the end of the list. The dummy in *Head* has a serial number of 1, and points to the tail. The tail represents the end of the list, has serial number 0, and points to `NIL`. The *mark* fields of their links are initialized to `false`. This initial configuration is shown in Figure 3.9.

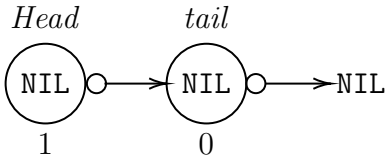

Figure 3.9: List initialization. The list starts with two dummy nodes: one stored in *Head* and a *tail* marking the end of the list. The integer below each node is its serial number, and the value inside each circle is its OpData field *data*, which is initially `NIL` (see Figure 3.7).

3.2.3 Appending and Claiming Nodes

New nodes are always appended at the head of the list by setting a new node's *next* pointer to the current sentinel and atomically updating *Head* to the new node.

Definition 7. *A node x is **appended** when Head is changed to x at line 33, and we denote this step by s^x.*

To reason about which nodes are or have been part of the list, we define the notion of *active nodes*. All nodes that have ever been appended, together with the two initial dummy nodes, are called *active*.

Definition 8. *A node x is **active** if it is one of the two dummy nodes created during initialization or has been appended at some step s^x.*

Each operation on the list begins by *claiming* the current sentinel as its node. To claim the sentinel, an operation *publishes* an OpData object into the node's *data* field via a CAS. The OpData object stores the Key field *key* of the operation and a Type field *type*, that indicates whether the operation is a SEARCH, INSERT, or DELETE operation (see Figure 3.7). Consequently, every claimed node is either an *insert*, *search*, or *delete* node.

Definition 9. *An operation op **publishes** its OpData object d when it successfully stores d into a node x at line 29. We denote this step by s^d and say that op **claims** x at s^d.*

Each active node is assigned a unique serial number. Each newly appended sentinel node receives a serial one larger than the previous sentinel's. Thus, assuming no removals, the serial numbers of nodes in the list decrease monotonically by one. We prove in Chapter 4 that serial numbers strictly increase as nodes are claimed and appended (see Lemma 21), and that the nodes in our list are sorted in decreasing order of serial number (see Invariant 16 and Item Property 2 **(ii)** of Lemma 30).

Because the serial numbers of appended nodes increase monotonically, and each append is preceded by a publish step that claims the current sentinel (as shown in Lemma 21 of Chapter 4), the serial number of an active node serves as a timestamp for its claim. This gives rise to a natural notion of newer and older nodes.

Definition 10. *For any two distinct active nodes x and y, we say that x is **newer** (**older**) than y if $x.serial > y.serial$ ($x.serial < y.serial$). Consequently, in any set of active nodes A, the **newest** (**oldest**) node is the one that is newer (older) than all other nodes in A.*

Each operation is linearized at the instant it claims a node. Thus, the result of an operation on the key k that claims a node x is determined by the newest *match* of x that was claimed by an INSERT or DELETE operation.

Definition 11. *A **match** m of an active node x is an active node with $m.serial < x.serial$ and $m.key = x.key$.*

Since we linearize operations in the order they claim their nodes, the key k is in the set when x is claimed if and only if the newest match of x claimed by an INSERT or DELETE operation was claimed by an INSERT operation.

Finally, because each appended node receives a serial number one larger than that of the previous sentinel, and removals only update the *next* pointer of an active node to an active node with a serial number smaller (proven as Property 2 **(ii)** of Lemma 30 in Chapter 4), the nodes in the list remain sorted in strictly decreasing order of serial number. This ordering allows us to formally describe when an active node is *skipped*, meaning it is bypassed by a forward link, or *skipped in the backward direction*, meaning it is bypassed by a backlink.

Definition 12. *An active node y is **skipped** if there exist active nodes x and z such that $x.next = z$ and $x.serial > y.serial > z.serial$. It is **skipped in the backward direction** if there exist active nodes x and z such that $z.backlink = x$ and $x.serial > y.serial > z.serial$ (see Figure 3.10).*

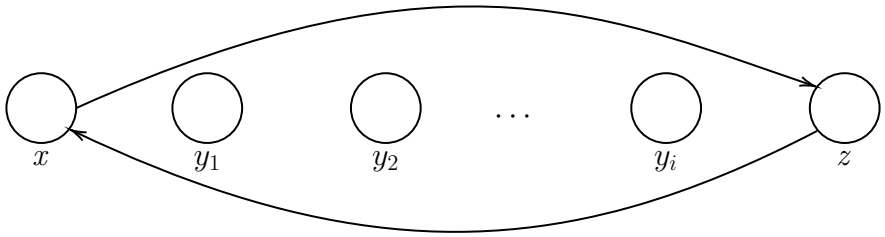

Figure 3.10: Active nodes skipped in the forward and backward directions. The nodes $y_1, y_2, \ldots, y_i$ are active nodes satisfying $x.serial > y_j.serial > z.serial$ for all $1 \leq j \leq i$. They are *skipped* because $x.next = z$, and they are *skipped in the backward direction* because $z.backlink = x$.

3.2.4 Helping

The integer field $x.end$ of a node x claimed by a SEARCH(k) or INSERT(k) operation determines where the operation's traversal should stop. Because the list is sorted in decreasing order of serial numbers, traversals proceed from nodes with larger serial values toward smaller ones. The value of $x.end$ therefore acts as a threshold: once the traversal reaches a node whose serial number is less than or equal to $x.end$, it can safely terminate.

Initially, $x.end = -1$ (see Figure 3.7), indicating that the traversal should continue until it reaches the *tail* node, whose serial number is 0. This field may later be increased to a non-negative value, either before or during the traversal, by an operation (either the one that claimed x or a concurrent one) that determines the key k was in the set when x was claimed. In particular, $x.end$ is set to a value of $w.serial$, where w is the first matching node of x–the node that the SEARCH(k) or INSERT(k) operation that claimed x is responsible for

removing. Consequently, the operation that claimed x either reaches w and removes it or stops upon encountering a node with serial number strictly less than $x.end$, knowing that w has already been removed (since w cannot be reached from a node with a serial number less than than $w.serial$). Thus, $x.end > 0$ both records that the operation should return `true` and defines a serial threshold beyond which the traversal by the operation that claimed x can safely terminate.

Finally, $x.end$ also indicates whether the SEARCH(k) or INSERT(k) that claimed it has determined its result. Upon completing its traversal, the operation attempts to set $x.end = 0$ using a CAS, but only if 0 exceeds the current value, ensuring that no positive *end* value is ever overwritten. Thus, if another operation reaches x and observes $x.end \geq 0$, it can infer that the operation which claimed x has already determined its result and that x may be safely removed from the list. Conversely, if $x.end = -1$, the claiming operation has not yet determined its result; in that case, an operation responsible for x's removal sets a *remove* flag in x to `true`, indicating that x should be removed by the ongoing operation that claimed it.

3.3 Pseudocode and Description

Before presenting the algorithm (Section 3.3.1–Section 3.3.3), we summarize each routine in Table 3.1 to give the reader an intuition of their roles.

Pseudocode appears in Figure 3.11–Figure 3.13. For ease of reference, subroutine calls are highlighted in purple, and instructions that modify shared memory are highlighted in yellow.

3.3.1 Operations and Operation Publication

Each SEARCH(k), INSERT(k), and DELETE(k) operation (shown in Figure 3.11) begins by creating an OpData object d with $d.key = k$ and $d.type$ corresponding to the operation (lines 16, 19, and 22), and then calls PUBLISH(d) (lines 17, 20, and 23) to claim a node x. After claiming x, each operation calls its core subroutine and returns either its result or, in the case of INSERT, its negation: SEARCH and INSERT call MTF(x), and DELETE calls REMOVEMATCHES(x) (lines 18, 21, and 24).

PUBLISH(d) (shown in Figure 3.11) repeatedly attempts to install d into the current sentinel's *data* field using a CAS (lines 27–29). Upon success, the operation linearizes, and the sentinel is claimed as the operation's node (line 30). Each publish attempt is followed by appending a

Routine	Description
SEARCH(Key k) : Boolean	Claims a node x via PUBLISH and returns the result of MTF(x), which returns `true` if k was in the set when x was claimed, and `false` otherwise.
INSERT(Key k) : Boolean	Claims a node x via PUBLISH and returns the negation of the result of MTF(x), which returns `true` if k was in the set when x was claimed (INSERT(k) returns `false`), and `false` otherwise (INSERT(k) returns `true`).
DELETE(Key k) : Boolean	Claims a node x via PUBLISH and returns the result of REMOVEMATCHES(x), which returns `true` if k was in the set when x was claimed, and `false` otherwise.
PUBLISH(OpData d) : Node	Publishes d into the sentinel node x to claim it, ensures a new sentinel node is appended, and returns x.
MTF(Node x) : Boolean	Subroutine of SEARCH(k) or INSERT(k) that claimed a node x. Traverses from x to determine if the key k was in the set when x was claimed. If so, promotes x to an insert node (if it was a search node), ensures a duplicate node with key k is removed, and returns `true`; otherwise returns `false`.
REMOVEMATCHES(Node x) : Boolean	Subroutine of DELETE(k) that claimed a node x. Traverses from x to check if k was in the set when x was claimed and removes all nodes with the key k it reached before a delete node with k (or the end of the list if no such delete node exists), and then removes x. Returns `true` if key k in the set when x was claimed, otherwise `false`.
INCREASEEND(Node x, int *end*) : void	Helper for MTF and REMOVEMATCHES. Ensures the traversal bound for MTF(x) (i.e., $x.end$) is at least *end*.
MARKANDREMOVE(Node x, Node y) : void	Helper for MTF and REMOVEMATCHES. Marks a node y and removes a contiguous chain of marked nodes that includes y, given an observed predecessor x.
FINDPRED(Node x, int *serial*) : Node	Helper for MARKANDREMOVE. Traverses from x to locate the predecessor of the node with serial number *serial*; returns the predecessor if found, or `NIL` otherwise.
SETBACKLINK(Node x, Node y) : void	Helper for MARKANDREMOVE. Ensures that y's backlink points to a predecessor of y whose serial number is at least $x.serial$, where x is an observed predecessor of y.

Table 3.1: Summary of the routines of our MTF list algorithm.

```
SEARCH(Key k) : Boolean
16. data ← OpData(k, search)          ▷ construct the operation's data object
17. node ← PUBLISH(data)              ▷ claim the sentinel by publishing data to it
18. return MTF(node)                  ▷ attempt a move-to-front for k

INSERT(Key k) : Boolean
19. data ← OpData(k, insert)
20. node ← PUBLISH(data)
21. return ¬MTF(node)                 ▷ INSERT(k) fails if k moved-to-front by operation

DELETE(Key k) : Boolean
22. data ← OpData(k, delete)
23. node ← PUBLISH(data)
24. return REMOVEMATCHES(node)        ▷ ensure node's matches are removed

PUBLISH(OpData data) : Node
25. node ← NIL
26. repeat                            ▷ try to claim sentinel and append new sentinel
27. |   sentinel ← Head
28. |   ▷ try to claim the sentinel by publishing data to it
29. |   if sentinel.data.CAS(NIL, data) = NIL then
30. |   |   node ← sentinel
31. |   ▷ append new sentinel with incremented serial pointing to old sentinel
32. |   newSentinel ← Node(sentinel.serial + 1, ⟨sentinel, false⟩)
33. |   Head.CAS(sentinel, newSentinel)
34. until node ≠ NIL
35. return node
```

Figure 3.11: SEARCH, INSERT, DELETE, and PUBLISH procedures.

new sentinel whose *next* pointer references the current sentinel and whose serial number is one greater (lines 32–33), ensuring that publish and append steps do not interleave and that each operation claims a distinct node uniquely identified by its serial (see Lemma 21). PUBLISH(d) terminates once d is successfully published, returning the claimed node x (lines 34 and 35).

```
MTF(Node node) : Boolean                          ▷ returns true ⟺ node.key in the set
36. first, last ← NIL
37. pred ← node
38. curr ← pred.next
39. ▷ walk until a serial number at most node.end found or a matching delete node
40. while curr.serial > max(0, node.end − 1) ∧ (curr.key ≠ node.key ∨ curr.type ≠ delete)
41.     if curr.key = node.key then                ▷ match found
42.         ▷ if match is an insert then record key as found and stop traversal
43.         if curr.type = insert then IncreaseEnd(node,curr.serial)
44.         if first = NIL then first ← ⟨pred, curr⟩   ▷ save first match reached
45.         last ← curr                            ▷ save last match reached
46.     pred ← curr
47.     curr ← pred.next
48.     ▷ if the last match confirms key membership, record success and stop traversal
49.     if last ≠ NIL ∧ last.end > 0 then IncreaseEnd(node, last.serial)
50. IncreaseEnd(node, 0)                          ▷ try to record key not found
51. if node.end > 0 then                          ▷ key is in the set
52.     node.type ← insert                        ▷ execute move-to-front
53.     if first ≠ NIL then                       ▷ ensure first is removed
54.         ▷ if first's operation op is traversing, flag first for removal by op
55.         if first.curr.end = −1 then first.curr.remove ← true
56.         if first.curr.end ≥ 0 then MarkAndRemove(first.pred, first.curr)
57. ▷ remove node if not promoted to insert or flagged for removal
58. if node.type = search ∨ node.remove then MarkAndRemove(Head, node)
59. return node.end > 0                           ▷ a positive end implies key present

RemoveMatches(Node node) : Boolean                ▷ returns true ⟺ node.key in the set
60. pred ← node
61. curr ← pred.next
62. matches ← Vector()                            ▷ stores matches reached by traversal
63. ▷ traverse until the list end or a matching delete node
64. while curr.serial > 0 ∧ (curr.key ≠ node.key ∨ curr.type ≠ delete)
65.     if curr.key = node.key then matches.add(⟨pred, curr⟩)
66.     pred ← curr
67.     curr ← pred.next
68. oldestInsert ← −1                             ▷ index of oldest insert in matches
69. for i ∈ {matches.size − 1, . . . , 0}         ▷ scan matches for an insert
70.     if matches[i].curr.type = insert then oldestInsert ← i break
71. for i ∈ {0, . . . , matches.size − 1}         ▷ remove every node in matches
72.     ▷ inform op that claimed match[i].curr its key in set
73.     if i < oldestInsert then IncreaseEnd(matches[i].curr, matches[i + 1].curr.serial)
74.     MarkAndRemove(matches[i].pred, matches[i].curr)
75. MarkAndRemove(Head, node)                     ▷ remove this delete node
76. return oldestInsert ≥ 0                       ▷ key present if oldestInsert set
```

Figure 3.12: MTF **and** RemoveMatches **procedures.**

```
IncreaseEnd(Node node, int end) : void                ▷ ensure node.end ≥ end
77. oldEnd ← node.end
78. if oldEnd < end then node.end.CAS(oldEnd, end)

MarkAndRemove(Node pred, Node node) : void            ▷ marks and removes node from list
79. pred ← FindPred(pred, node.serial)                ▷ ensure pred.next = node
80. if pred = NIL then return                         ▷ node already removed
81. SetBacklink(pred, node)                           ▷ set node.backlink before marking node
82. succ ← node.next
83. repeat                                            ▷ retry until node is marked
84. |   node.link.CAS(⟨succ, false⟩, ⟨succ, true⟩)
85. |   ⟨succ, mark⟩ ← node.link
86. until mark
87. ▷ remove the chain of marked nodes between pred and succ
88. repeat
89. |   while pred.mark                               ▷ follow backlinks to unmarked pred
90. |   |   pred ← pred.backlink
91. |   curr ← pred.next
92. |   while succ.mark                               ▷ skip over marked successors
93. |   |   succ ← succ.next
94. |   ▷ attempt to the remove the marked chain
95. |   pred.link.CAS(⟨curr, false⟩, ⟨succ, false⟩)
96. until pred.next.serial ≤ succ.serial              ▷ marked chain removed
97. SetBacklink(pred, pred.next)                      ▷ ensure curr.backlink skips the chain

FindPred(Node pred, int serial) : Node
98. ▷ find the predecessor of the node with serial number serial
99. curr ← pred.next
100. while curr.serial > serial
101. |   pred ← curr
102. |   curr ← curr.next
103. return (curr.serial = serial ? pred : NIL)      ▷ return the predecessor or NIL if none

SetBacklink(Node pred, Node node) : void
104. ▷ ensures node.backlink points to a node with a serial number at least pred.serial
105. backlink ← node.backlink
106. while backlink = NIL ∨ backlink.serial < pred.serial
107. |   node.backlink.CAS(backlink, pred)             ▷ try to set node.backlink to pred
108. |   backlink ← node.backlink
```

Figure 3.13: IncreaseEnd, MarkAndRemove, FindPred, **and** SetBacklink **procedures**.

3.3.2 MTF and RemoveMatches

MTF(x) and RemoveMatches(x) share two core responsibilities: determining whether the key $x.key = k$ was in the set when x was claimed, and removing duplicate nodes with the same key. Each begins a traversal from the node x. Both procedures succeed, returning `true`, if k was in the set when the node x was claimed and fail, returning `false`, otherwise (see Section 4.7).

MTF(x) begins its traversal at lines 37–38 by initializing *pred* as x and *curr* as *pred.next*, where *pred* and *curr* are the predecessor and current nodes visited during the traversal. MTF(x) then enters the main loop of its traversal at lines 40–49, advancing one node per iteration (lines 46–47). The loop terminates once either a delete node with key k is reached or a node with a serial number less than $x.end$ is encountered (see the exit condition at line 40).

Initially, $x.end = -1$ (see Figure 3.7), so the traversal proceeds to the tail node ($tail.serial = 0$) unless $x.end$ is increased to a positive value before or during traversal. The field $x.end$ is increased to a positive value during MTF(x)'s traversal either by MTF(x) itself or by a concurrent Delete(k) in its call to RemoveMatches, if the key k was present at the time x was claimed (as shown in Lemma 45 of Section 4.7). A positive value of $x.end$ therefore indicates that MTF(x) should return `true` and also limits the traversal of MTF(x).

To increase $x.end$, IncreaseEnd(x, *end*) is called by MTF and RemoveMatches. It is invoked with $end > 0$ (at lines 43, 49, and 73) to indicate the success of MTF(x), or with $end = 0$ (at line 50) by MTF(x) after it has completed its traversal to indicate its failure. IncreaseEnd(x, *end*) ensures $x.end \geq end$ by comparing the current value of $x.end$ with *end* and performing a CAS to increase it when *end* is larger (lines 77–78). As a result, $x.end$ increases monotonically, guaranteeing that a positive value, indicating a successful MTF(x), is never overwritten by 0 (proved as Lemma 41 in Chapter 4).

IncreaseEnd(x, *end*) is called with $end > 0$ when MTF(x) or a concurrent Delete(k) confirms that the most recent matching node of x, claimed by an Insert(k) or Delete(k), was claimed by an Insert(k). This confirmation occurs once it is known that an insert node z with key k could be reached from x without encountering any delete node for k, which can occur one of three ways:

1. When MTF(x) encounters such a node z, it calls IncreaseEnd(x, $z.serial$) (line 43), ensuring $x.end \geq z.serial$. This records the operation's success and causes the traversal

from x to terminate at the next evaluation of the loop's exit condition (line 40).

2. When MTF(x) sees that the previous match it encountered, saved in the local variable *last* (line 45), has $last.end > 0$, it concludes that the operation that claimed *last* has already succeeded. Since no delete node for k lies between x and *last* (by the exit condition at line 40), MTF(x) concludes the existence of such a z and calls IncreaseEnd(x, $last.serial$) (line 49) to mark success and terminate at the next loop evaluation.

   Monitoring $last.end$ at line 49 ensures correctness even if a concurrent MTF(*last*) removes an insert node that MTF(x) was approaching. Because MTF(*last*) must set $last.end > 0$ before removing such a node (see lines 51 and 56), MTF(x) can later verify this by reading $last.end$, confirming that k was in the set when x was claimed (see Figure 3.3).

3. A concurrent Delete(k) can also set $x.end > 0$ if it reaches a matching insert node z from x without encountering a delete node for k. It does so by setting $x.end$ to the serial number of the first matching node w of x that it encounters, thereby recording the result of MTF(x) and providing a stopping point for its traversal.

   After setting $x.end$ to $w.serial$, the concurrent Delete(k) can safely remove x, and any insert node along MTF(x)'s traversal path, knowing that MTF(x) has already been informed of its success (see Figure 3.5).

In addition to MTF(x) saving the previous matching node it encountered in *last*, MTF(x) also records the first matching node it encounters. At line 44, the pair $\langle v, w \rangle$ is saved as *first*, where w is the first matching node of x and v was its predecessor. If MTF(x) succeeds, this pair is later used to remove w, since the removal routine MarkAndRemove requires both the node and its predecessor as arguments (see line 56).

As mentioned, MTF(x) calls IncreaseEnd(x, 0) at the end of its traversal (line 50) to finalize its result. If no prior update set $x.end > 0$, this call records failure by setting $x.end$ to 0. However, since IncreaseEnd only increases $x.end$, any positive value assigned earlier during the traversal–indicating the success of MTF(x)–is preserved.

Thus, at line 51, if $x.end > 0$, MTF(x) has succeeded. The operation then updates $x.type$ to `insert` (line 52), effectively moving key k toward the front of the list, and removes the first matching node w stored in *first*. To avoid the scenario illustrated in Figure 3.6, w is

physically removed only after the operation that claimed it has determined its result. This is verified by checking that $w.end \geq 0$ before the call to MarkAndRemove($*$, w) at line 56.

If MTF(x) finds that MTF(w) is still traversing (i.e., $w.end = -1$ at line 55), it preemptively sets $w.remove \leftarrow$ `true` at line 55) to notify MTF(w) to remove w itself. This update occurs before the check at line 56 to avoid a race where MTF(w) completes before $w.remove \leftarrow$ `true` is set, causing it to evaluate $w.remove \neq$ `true` at line 58 and skip the subsequent call to MarkAndRemove($Head$, w), leaving w in the list.

At line 58, MTF(x) checks whether x is a search node or if $x.remove =$ `true`. If x is a search node, this indicates that MTF(x) failed because had it succeeded, x would have been promoted to an insert node at line 52. Such nodes are redundant since they never correspond to a key in the set. If $x.remove =$ `true`, a concurrent process has informed MTF(x) to remove x itself. In either case, MTF(x) removes x by calling MarkAndRemove. Finally, MTF(x) returns whether $x.end > 0$, which is `true` if k was in the set when the operation was linearized and `false` otherwise (line 59).

The procedure RemoveMatches(x) ensures that all nodes reachable from x with key k, up to the first delete node encountered with k, are removed after a Delete(k) claims x. Like MTF, it begins its traversal at lines 60–61 by setting $pred \leftarrow x$ and $curr \leftarrow pred.next$, where *pred* and *curr* denote the predecessor and current nodes being visited by the traversal. RemoveMatches(x) then enters the main loop of its traversal at lines 40–49, advancing one node per iteration (see lines 46–47). During traversal, RemoveMatches(x) adds every matching node it encounters, and its predecessor, as a pair in the vector *matches* (lines 62–65). The traversal continues until a matching delete node of x, or the end of the list, is reached (see line 64).

After collecting all matches, the routine scans *matches* in reverse to locate the index of the oldest matching insert node in *matches*, $oldestInsert$ (lines 69–70) ($oldestInsert = -1$ if no matching insert node found by line 68). It then removes every node in *matches* by calling MarkAndRemove (line 74) and finally removes x itself (line 75).

Before each removal, for every match $matches[i].curr$ that appears earlier in *matches* than $matches[oldestInsert].curr$, RemoveMatches calls IncreaseEnd on $matches[i].curr$ and the serial number $matches[i{+}1].curr.serial$ (lines 73–73). This informs the traversal of MTF($matches[i].curr$) that the key k was in the set when $matches[i].curr$ was claimed and instructs it to stop once it reaches or skips $matches[i{+}1].curr$. This ensures that

MTF($matches[i].curr$) returns the correct result even if RemoveMatches(x) removes an insert node along its traversal path (see Figure 3.5). The node $matches[i+1].curr$ is the first matching node of $matches[i].curr$ that MTF($matches[i].curr$) is responsible for removing, so if MTF($matches[i].curr$) finds $matches[i+1].curr$ has been skipped, it can safely stop its traversal knowing that $matches[i+1].curr$ has been removed. Finally, if no matching insert node of x was found (i.e., $oldestInsert < 0$), RemoveMatches(x) returns `false` (line 76); otherwise, it returns `true`.

3.3.3 Node Removal

MarkAndRemove(x, y) (shown in Figure 3.13) is invoked by MTF at lines 56 and 58, and by RemoveMatches at lines 74 and 75, to remove the node y from the list. It marks and removes y, possibly together with a contiguous chain of marked nodes that includes it (lines 78–96). The routine first calls FindPred(x, $y.serial$) to locate a predecessor x' of y such that $x'.next = y$ (line 79). This backlink allows concurrent removals that fail to update the link of y because y is marked to efficiently find an unmarked predecessor of y. Finally, the node y is marked by performing a CAS on its link until y is marked (lines 83–86).

After marking, MarkAndRemove extends the marked chain in both the forward and backward directions by following backlinks of x to an unmarked predecessor *pred*, and by following next pointers of y to an unmarked node *succ* (lines 89–93). It then attempts to remove the entire chain by performing a CAS to update *pred.next* from *curr* to *succ* (line 95). The loop terminates when $pred.next.serial \leq succ.serial$ (line 96), indicating that the chain has been skipped.

Finally, backlinks are updated using SetBacklink(*pred*, *curr*) to ensure that a backward link skips over the removed chain (line 97). As a result, marked chains are bypassed in both forward and backward directions, guaranteeing that any traversal beginning after the removal cannot reach any node in the removed chain. This is important for our step complexity proof, where we show that the number of marked nodes that can be reached by an operation by following links and backlinks is bounded by contention (see Lemma 56 of Section 5.4).

The procedure FindPred(x, $y.serial$) (shown in Figure 3.13) is used by MarkAndRemove (x, y) to locate a node that points to y. As a precondition, x must be newer than y. The routine traverses from x until it either finds a node with serial number $y.serial$ or reaches a node z with $y.serial \geq z.serial$ (lines 99–102). It returns the predecessor of the node with that serial number if found ($z.serial = y.serial$), or `NIL` otherwise ($z.serial > y.serial$)

(line 103).

SetBacklink(x, y) (shown in Figure 3.13) is called by MarkAndRemove to set the backlink of node y, ensuring that $y.backlink$ points to a node whose serial number is at least $x.serial$. As a precondition, x must be a node that currently or previously pointed to y. The routine performs a CAS on $y.backlink$ at line 107 in each iteration of the loop at lines 106–108, attempting to update its value (read at line 105 or 108) to x until the condition is satisfied (see the exit condition at line 106). This ensures that any active nodes with serial numbers between $x.serial$ and $y.serial$ are skipped in the backward direction when the routine completes.

Chapter 4

Correctness

4.1 Introduction

4.1.1 Linearization and Set Membership

To prove that our MTF list is *linearizable* (see Section 1.1.5), we must show that each operation on our list returns the same result as it would if operations were executed sequentially in their linearization order. To do so, we first define each operation's linearization point as the moment it publishes its operation data at line 29.

Definition 13. *The operation that constructs an OpData object d (at line 16, line 19, or line 22) is linearized at step s^d, when the* CAS *at line 29 successfully publishes d in the sentinel node read at line 27.*

This linearization ensures that if one operation terminates before another begins, the former is linearized before the latter.

Definition 14. *An **update operation** on key k is an operation that is either an* INSERT(k) *or a* DELETE(k).

We assume the set represented by the list is initially empty. The sequential specification of the set (as defined in Definition 2) implies a key k is present after a sequential execution if and only if the sequence contains an INSERT(k) and the most recent update operation on k is an INSERT(k).

For a concurrent set to be linearizable, the same condition must hold with respect to the linearization order: a key k is in the set when an operation *op* on k is linearized if and only if an INSERT(k) was linearized earlier and, among all update operations on k linearized before *op*, the one with the latest linearization point is an INSERT(k).

Accordingly, we show that a SEARCH(k) or DELETE(k) operation *op* returns `true` if and only if the most recent update operation on k linearized before *op* was an INSERT(k) (see Theorems 48 and 49 of Section 4.8). Thus, these operations return `true` exactly when k is in the set at their linearization point, consistent with the sequential order of operations defined by the linearization.

Similarly, an INSERT(k) operation *op* returns `false` if and only if the most recent update operation on k linearized before *op* was an INSERT(k).

4.1.2 Proof Roadmap

We begin in Section 4.2, where we show that only active nodes can appear in the list and that every link points to an active node. As such, we can consider only active nodes when reasoning about which nodes are reachable or can be traversed to from a claimed node.

In Section 4.3, we show that publish and append steps occur in strict order: each successful publication is immediately followed by exactly one append, which appends a node whose serial number is one greater than that of the previously claimed node. Hence, each operation receives a timestamp in the form of its node's serial number that reflects its linearization order.

In Section 4.4, we establish that every backlink corresponds to a pre–existing forward link, and that the backlink of a node is set to a non-`NIL` value before a node is marked. This guarantees the safety of cooperative removals and supports the properties of foward links formalized in Section 4.5.

In Section 4.5, we establish key invariants of forward links. We show that serial numbers strictly decrease along links, so if a traversal starts from the node claimed by some operation, it will only visit nodes belonging to operations that are linearized earlier. We further show that links never overlap, and all skipped nodes are marked. These properties allow us to precisely capture the state of the list as it evolves over time.

In Section 4.6, we show that an operation on the key k that is linearized between a DELETE(k) and the next INSERT(k), does not visit an insert node with key k during its traversal, which would cause the operation to incorrectly conclude that k was in the set when it was linearized.

In Section 4.7, we prove the correctness of the traversal routine of a call to MTF(x), invoked on a node x claimed by a SEARCH(k) or INSERT(k) operation. We show that MTF(x) returns `true` if and only if k was in the set when x was claimed; that is, if the most recent update

operation on the key k that linearized before x was an INSERT(k).

Finally, in Section 4.8, we show that each SEARCH, INSERT, and DELETE operation returns a result consistent with its linearization (as defined by Definition 13) and the sequential specification of a set (as defined by Definition 2).

4.2 Active Nodes

We begin our proof by establishing some facts about active nodes, which helps us reason about which nodes can be in the list at a given time. We first observe the following.

Observation 15. *The sentinel node is an active node x such that $x \neq tail$.*

This holds because at initialization $Head$ points to the dummy node that points to *tail*, and it is updated only by a successful CAS on $Head$ at line 33 to a node that we newly created at line 32. Next, we show that a node pointed to by a node must be active.

Invariant 16. *Let x be a node. If $x.next = y$ and $y \neq$ `NIL`, then y is active.*

Proof. When the list is initialized, $Head.next = tail$ is the only link that exists, and *tail* is one of the two dummy nodes created when initializing the list, so the invariant holds. Assume the invariant holds for the prefix α of an execution. We show the invariant continues to hold after executing the next step s (IH). The invariant can be falsified if s is a step that points a node x to a node y and y is not active.

A node x is set to point to a node y in one of two places: at line 32, and line 95.

Case 1: If the link of a node x is updated to a node y at line 32, $Head = y$ was true at line 27, so y satisfies the claim by Observation 15.

Case 2: If the link of a node x is updated to a node y at line 95, y was read as *node.next* at line 85 or as *succ.next* at line 93, where *node* and *succ* are nodes, so y satisfies the claim by (IH).

□

Together, Observation 15 and Invariant 16 imply that only active nodes are in the list; nodes constructed by operations at line 32 that are not appended at line 33 cannot be reached from an active node.

4.3 Publish Steps and Serial Numbers

In this section, we prove that each terminating operation successfully publishes the OpData object d it constructs (at line 16, 19, or 22) by successfully storing it in the sentinel node during PUBLISH(d) (called at line 17, 20, and 23).

The loop in PUBLISH(d) (line 26) terminates only once $node \neq$ `NIL` (line 34), and returns *node* (line 35). Since *node* is initialized to `NIL` (line 25), and it is only updated at line 30 if the CAS at line 29 successfully sets $sentinel.data$ to d, where *sentinel* is the node in $Head$ at line 27, the following observation holds.

Observation 17. *In a completed call* PUBLISH(d)*, d is published into an active node x that is stored in $Head$ at line 29, which is returned at line 35.*

Furthermore, once d has been stored in $x.data$ by an operation op, $x.data$ never changes. This implies an operation $op' \neq op$ cannot claim x after it has been claimed by op.

Observation 18. *A node's data field is assigned a non-`NIL` value only at line 29 and remains unchanged thereafter.*

In Lemma 21, we relate publication and append steps and serial numbers. Each operation first publishes to the current sentinel and only then appends a new node whose serial number is exactly one greater than the sentinel's serial number. Consequently, in an execution, each publication step is followed by the append step of the next node to be claimed, and the serial numbers of claimed nodes are consecutive, increasing by one at each claim.

We first observe that the serial number of a node is assigned at line 32, before it is appended to the list, which implies the following.

Observation 19. *The serial number of a node never changes.*

Next, we provide some useful notation to describe the relative order of steps and configurations in an execution.

Definition 20. *We write $s \prec s'$ if step s occurs before step s' in an execution, and $C \prec C'$ if configuration C occurs before configuration C'. Similarly, $s \preceq s'$ (or $C \preceq C'$) indicates that s (or C) occurs before or is equal to s' (or C') in the execution.*

Finally, we show Lemma 21.

Lemma 21. *Suppose $m \geq 1$ operations successfully publish by a* CAS *at line 29, starting from the initial configuration. For each such operation op_i ($1 \leq i \leq m$), define:*

- *s^{d_i}: the step where op_i successfully publishes its OpData object d_i to the sentinel at line 29, and*
- *s^{x_i}: the step where the node x_i claimed by op_i is appended to the list by a* CAS *at line 33.*

*Then the sequence of **all** publish and append steps is*

$$s^{d_1} \prec s^{x_2} \prec s^{d_2} \prec \ldots \prec s^{x_m} \prec s^{d_m}.$$

Moreover, the published nodes receive consecutive serial numbers:

$$\forall\, i \in \{1, \ldots, m\}: \quad x_i.serial = i.$$

Proof. We use induction on the number of successful publications.

The first publication step s^{d_1} is the first execution of the CAS at line 29, so it precedes any append at line 33. Furthermore, s^{d_1} claims the dummy node in $Head$ created at intialization (see Section 3.2.2) by lines 27–29, and so by the fact that $Head.serial = 1$ initially, we have $x_1 = 1$.

Assume the lemma holds for the first i publications (IH). To show the lemma holds for the $(i+1)$–st publication, we first show that exactly one append occurs between s^{d_i} and $s^{d_{i+1}}$.

Claim 21.1. *At least one append step occurs between s^{d_i} and $s^{d_{i+1}}$.*

Proof. Immediately after s^{d_i}, node x_i serves as the sentinel, since by (IH) no new node is appended between s^{x_i} and s^{d_i}. By Observation 18, each node can be claimed by only one operation. Hence, while $Head = x_i$, any attempt to publish at line 29 after s^{d_i} must fail, as x_i was already claimed by op_i at s^{d_i}.

Therefore, the next successful publication $s^{d_{i+1}}$ cannot occur until the head changes by a successful CAS at line 33 after s^{d_i}. Consequently, at least one append must occur between s^{d_i} and $s^{d_{i+1}}$. □

Claim 21.2. *At most one append step occurs between s^{d_i} and $s^{d_{i+1}}$.*

Proof. Assume, for contradiction, that at least two distinct nodes are appended between s^{d_i} and $s^{d_{i+1}}$; that is, two different operations each perform a successful CAS at line 33 to append a node with $data = \texttt{NIL}$ (see lines 32–33).

Let *op* and *op′* be the first two such operations, with *op*'s append preceding that of *op′*. This implies that *op* executed lines 27–33 of PUBLISH before *op′*, as guaranteed by the CAS at line 33 and the absence of ABA on $Head$: every CAS to $Head$ always installs a newly created node (see line 32).

Before *op′* appends, it must first execute lines 27–29 of PUBLISH. At line 27, it would read the node appended by *op* as the sentinel. Next, *op′* attempts to claim that sentinel by performing a CAS at line 29. If this CAS succeeds, *op′* has claimed the sentinel, contradicting the fact that the next successful claim after s^{d_i} occurs at $s^{d_{i+1}}$. If it fails, then some other operation must have already claimed the sentinel, again contradicting that $s^{d_{i+1}}$ is the next claim after s^{d_i}.

Therefore, the assumption that two appends occur in this interval leads to a contradiction; hence, at most one append can occur between s^{d_i} and $s^{d_{i+1}}$. □

By Claim 21.1 and Claim 21.2, exactly one append step occurs between s^{d_i} and $s^{d_{i+1}}$. By (IH), no append step occurs between s^{x_i} and s^{d_i}, so node x_i remains the sentinel after s^{d_i} until the head changes. By Observation 18, x_i cannot be claimed by op_{i+1}. Because the sentinel changes exactly once–when a new node x' is appended–between s^{d_i} and $s^{d_{i+1}}$, and each operation claims the current sentinel by lines 27–29, the newly appended node must be the one claimed by op_{i+1}; that is, $x' = x_{i+1}$.

Hence, the sequence of all publish and append steps from s^{d_i} through $s^{d_{i+1}}$ is:

$$s^{d_i} \prec s^{x_{i+1}} \prec s^{d_{i+1}}.$$

Together with the inductive hypothesis (IH), this establishes the sequence of publish and append steps for the first $i + 1$ publications.

By (IH), $x_j.serial = j$ for all $1 \leq j \leq i$. Using the fact that the sequence of all publish and

append steps from s^{d_i} through $s^{d_{i+1}}$ is:

$$s^{x_i} \prec s^{d_i} \prec s^{x_{i+1}} \prec s^{d_{i+1}},$$

Head points to x_i throughout the interval $(s^{x_i}, s^{x_{i+1}})$. And by Observation 19, the serial number of x_i never changes, so before the CAS at line 33 appends x_{i+1}, op_{i+1} read $Head = x_i$ at line 27 and set $x_{i+1}.serial = x_i.serial + 1$ (lines 32–33). Hence

$$x_{i+1}.serial = x_i.serial + 1 = i + 1.$$

The serial numbers of nodes do not change, so we have $\forall\ i \in \{1, \ldots, m\} : x_i.serial = i$. □

Given the fact that the serial number of the *tail* node is 0, and we have $Head.serial = 1 > 0 = tail.serial$ initially (see Section 3.2.2), we obtain the following corollary.

Corollary 22. *The following properties hold:*

(i) *Every active node has a unique non-negative integer serial number, and*

(ii) *the sentinel node has the maximum serial number among all active nodes.*

Another straightforward corollary of Lemma 21 is the following.

Corollary 23. *If $s_x \prec s_y$, then $x.serial < y.serial$.*

4.4 Backlinks

In this section, we prove properties related to the backlinks of nodes, which we use in Section 4.5 to establish properties about nodes that point to each other in the forward direction. We first observe that the backlink of a node is set before the node is marked. This follows from the fact that a node y is marked at line 84 of a call to MarkAndRemove(x, y). Before this, SetBacklink(x', y) is called at line 81, and it returns only after $y.backlink \neq$ `NIL` by line 106.

Observation 24. *If a node y is marked, then $y.backlink \neq$ `NIL`.*

Next, we aim to prove Lemma 27: if $y.backlink = x'$ in a configuration C', then there exists a configuration $C \preceq C'$ in which x' is active and satisfies $x'.next = y$. To do so,

we first establish Lemma 26, which shows that if *pred* $\neq$ `NIL` after the execution of line 80 in MarkAndRemove, then the local variables *pred* and *node* satisfied *pred.next* = *node* where *pred* is active. This implies that whenever SetBacklink(x, y) is invoked at line 81 or line 97 of MarkAndRemove, there exists a configuration preceding the call in which x is active and $x.next = y$.

To prove Lemma 26, we first show that x is active and that $x.next = y$ held before a call to MarkAndRemove(x, y), provided that $x \neq Head$ when the call is made.

Lemma 25. *Let C' be a configuration immediately before some call to* MarkAndRemove*(x, y). The following holds in some configuration $C \preceq C'$:*

(i) *x is active,*

(ii) *and if $x \neq Head$ in C', then $x.next = y$.*

Proof. We begin by proving a useful property of the variables *pred* and *curr* at the start of each iteration of the loop in lines 40–49 of MTF and the loop in lines 64–67 of RemoveMatches. This is important because the nodes passed to MarkAndRemove were previously held in these variables.

Claim 25.1. *Let C' be a configuration at the start of an iteration of the loop at lines 40–49 or the loop at lines 64–67. Let x and y be the values of pred and curr in C'. Then, there $\exists\ C \preceq C'$ in which $x.next = y$ and x is active.*

Proof. We prove the claim by induction on the number of loop iterations.

If C' is at the start of the first iteration of the loop at line 40, let C be the configuration after *curr* is read from *pred.next* at line 38. By Observation 15, MTF is invoked with an active node as its argument at lines 17–18 or 20–21. We conclude that *pred* is active in C.

Similarly, if C' is at the start of the first iteration of the loop at line 64, let C be the configuration after *curr* is read from *pred.next* at line 61. By Observation 15, RemoveMatches is invoked with an active node as its argument at lines 23–24. Since line 60 sets *pred* to the argument of RemoveMatches, it again follows that *pred* is active.

For the inductive step, assume the claim holds for some number of iterations of either loop (IH). In subsequent iterations, let C be the configuration after *curr* is read from *pred.next*

at line 47 (for the loop at line 40) or line 67 (for the loop at line 64) of the previous iteration. And by (IH), line 46 (for the loop at lines 40–49) and line 66 (for the loop at lines 64–61) and Invariant 16, *pred* is active in C. In all cases, we have $pred.next = curr$ and *pred* is active in $C \preceq C'$. □

Now that we have shown Claim 25.1, we prove the lemma. MarkAndRemove(x, y) is invoked at lines 56, 58, 74, and 75.

At line 56 of MTF, MarkAndRemove it is called with arguments $x = first.pred$ and $y = first.curr$, where *first* was assigned $\langle pred, curr \rangle$ at line 44 during an iteration of the loop at line 40. By Claim 25.1, *pred* is active and $pred.next = curr$ held in a configuration $C \preceq C'$, so claim **(i)** holds and the consequent of claim **(ii)** holds.

At line 74, MarkAndRemove is called with a pair $\langle pred, curr \rangle$ from the *matches* vector, populated at line 65 during an iteration of the loop at line 64 in RemoveMatches. Again, Claim 25.1 ensures *pred* is active and $pred.next = curr$ held in a configuration $C \preceq C'$, so claim **(i)** holds and the consequent of claim **(ii)** holds.

At lines 58 and 75, the operation MarkAndRemove$(Head, node)$ is called by the procedures MTF$(node)$ and RemoveMatches$(node)$, respectively. By Observation 15, the node in $Head$ is always active, so take C as C', and claim **(i)** holds. Claim **(ii)** is trivially true because its antecedent is false. □

Lemma 26. *If $pred = x' \neq$ `NIL` at line 80 of a call to* MarkAndRemove*$(x, node)$, then x' is active and $x'.next = node$ at some moment during the execution of* FindPred *at line 79.*

Proof. The call to FindPred$(x, node.serial)$ at line 79 returns the non-`NIL` node x'. By Lemma 25, there exists a configuration preceding line 79 in which x is active. By Invariant 16, every node reachable from an active node is itself active.

The node $x' \neq$ `NIL` returned by FindPred$(x, node.serial)$ is reached from x via line 99 and lines 101–102—is active. Furthermore, by Corollary 22, active nodes have distinct serial numbers. Therefore, by line 103 and the fact that $x' \neq$ `NIL`, we conclude that $x'.next.serial = node.serial$, and hence $x'.next = node$ when $x'.next$ was last read at line 99 or line 102. □

We now combine Lemma 26 with Invariant 16 to establish the following lemma, which shows

that whenever a backlink from y to x exists, it mirrors a forward link that connected (or connects) x to y.

Lemma 27. *If $y.backlink = x \neq$ `NIL` in a configuration C', then in some configuration $C \preceq C'$, node x is active and $x.next = y$.*

Proof. When the list is initialized, all backlinks are `NIL`, so the lemma holds vacuously. Assume the lemma holds for some prefix α of an execution (IH). We show that it continues to hold after the next step s.

We must show that, whenever s sets $y.backlink$ to a node x, there is some configuration $C \preceq C'$ when x is active and and $x.next = y$.

The backlink of a node y is set to a non-`NIL` node x only by a successful CAS at line 107 of SetBacklink(x, y). This procedure is invoked either at line 81 or at line 97 of a call to MarkAndRemove.

Case 1. Call at line 81. If SetBacklink(x, y) is invoked at line 81 of MarkAndRemove, then x and y are the values of MarkAndRemove's local variables *pred* and *node*. By Lemma 26, x is active and $x.next = y$ at some point during the execution of FindPred at line 79.

Case 2. Call at line 97. If SetBacklink(x, y) is called at line 97 of MarkAndRemove, then x and y are the values of the local variable *pred* and *pred.next*. Let C be the configuration after *pred.next* is read at line 97. At C, $x.next = y$.

By Lemma 26, when the test at line 80 fails, *pred* points to an active node. If *pred* is later updated at line 90, it continues to point to an active node by (IH). Thus, at C, $x = pred$ is active.

□

4.5 Links

In this section we prove properties about the links between active nodes. These properties are expressed by Lemma 30 in the form of invariants that help us capture the state of the list in each configuration. Property 1 gives invariants about local variables in MarkAndRemove, which help us reason about the links between nodes after a link is updated at line 95.

Property 2 establishes key invariants for the link of every active node except the *tail*: its link always points to another node (Property 2 **(i)**); the node's relative position in the list is determined by its serial number (Property 2 **(ii)**); any active node that a link skips over must be marked(Property 2 **(iii)**); and the links of active nodes never overlap (Property 2 **(iv)**).

Before proving Lemma 30, we establish key properties of marked nodes. In particular, the next observation helps us prove that once a chain of marked nodes is skipped, an unmarked node can not later appear in that chain.

Observation 28. *A marked node never becomes unmarked, and its link field never changes.*

This follows from the fact that a node's *mark* field only changes at line 84, where it is updated from `false` to `true`, and the *link* field of a node can be changed only by the CAS at line 95, which succeeds only if the node is unmarked.

Next, we show that only active nodes that are not the *tail* node are marked, ensuring that a chain of marked nodes cannot extend beyond *tail*.

Lemma 29. *Only active nodes can be marked, and the tail node is never marked.*

Proof. By Observation 24, if a node x is marked, then $x.backlink \neq$ `NIL`. By Lemma 27 and Invariant 16, this implies x is active.

The *tail* node is initially unmarked (Section 3.2.2), and marking occurs only at line 84 in MarkAndRemove($*$, x). In every such call:

1. At lines 56 and 74, the argument x is the node *curr* at line 44 or 65. In both cases $curr \neq tail$ since $curr.serial \neq 0$ by the condition at line 40 or 64 (recall $tail.serial = 0$ by Section 3.2.2).

2. At lines 58 and 75, the argument x is the sentinel node claimed by the operation (Observation 17, lines 17–18, 20–21, or 23–24), and by Observation 15 this sentinel is not the *tail*.

Thus, no call to MarkAndRemove ever marks the *tail*. □

Now we show Lemma 30.

Lemma 30.

Property 1. *When the condition at line 92 of* MarkAndRemove *is evaluated as false, the following properties hold for the nodes in the local variables pred, curr, succ, and node:*

(i) *succ* $\neq$ `NIL` *is an unmarked node,*

(ii) *pred and succ are active,*

(iii) $pred.serial > node.serial > succ.serial$*,*

(iv) *and for every active node* z *with* $pred.serial > z.serial > succ.serial$*,* $z.mark =$ `true`*.*

Property 2. *In every configuration, if* $x \neq tail$ *is an active node, and* $x.next = y$*,*

(i) $y \neq$ `NIL`*,*

(ii) $x.serial > y.serial$*,*

(iii) *for every active node* z *with* $x.serial > z.serial > y.serial$*,* z *is marked,*

(iv) *no configuration can have an active node* x' *with* $x'.next = y' \neq$ `NIL` *such that* $x'.serial > x.serial > y'.serial > y.serial$*.*

Proof. When no steps have taken place, Property 1, vacuously holds. Property 2 follows from the fact that the list is composed of the $Head$ and $tail$ dummy nodes. $Head.next = tail$ so Property 2 **(i)** holds. Since $Head.serial = 1 > 0 = tail.serial$, Property 2 **(ii)** and Property 2 **(iv)** holds. Property 2 **(iii)** holds by the fact that no active nodes other than $Head$ and $tail$ exist.

Assume the invariant holds for a prefix α of an execution (IH). We show the lemma holds for $\alpha \cdot s$, where s is some step of the algorithm.

Proof of Property 1

We show if s evaluates the condition at line 92 as false in some call MR to MarkAndRemove, then Property 1 **(i)**–Property 1 **(iv)** hold.

Property 1 **(i)** follows directly from the condition at line 92 evaluating to `false`. We have *succ* $\neq$ `NIL` at line 92, since an active node x that is marked at line 86 or 92 pointed to *succ* at line 85 or 93, respectively. By Lemma 29, $x \neq tail$. Hence, by Property 2 **(i)** of (IH) and the fact that $x.next = succ$ held with $x \neq tail$, it follows that *succ* $\neq$ `NIL`.

To show Property 1 **(ii)**–Property 1 **(iv)** hold, we show that these properties hold at specific points during the loop at lines 88–95 before the step s is executed: immediately before the loop at lines 89–90 (Claim 30.1), and after each iteration of the loop at lines 89–90 (Claim 30.2) and the loop at lines 92–93 (Claim 30.3).

Claim 30.1. *Property 1* ***(ii)****–Property 1* ***(iv)*** *hold at the beginning of each iteration of* MR*'s loop at lines 88–96 before s is executed.*

Proof. We prove the claim by cases.

Case 1. MR is at the beginning of the first iteration of the loop at lines 88–96.

By Lemma 26, there is a moment during MR's call to FindPred at line 79 at which *pred* is active and *pred.next* = *node*, which by Invariant 16 implies that *node* is active at that moment. So we can apply Property 2 of (IH) to *pred* and *node*.

Using the fact that *node* is active and *node.next* = *succ* at line 85, *succ* is active by Invariant 16, establishing Property 1 **(ii)**.

Next, because *pred* was active with *pred.next* = *node* during FindPred, and with *succ* is read as *node.next* at line 85, by (IH) on Property 2 **(ii)**, we have *pred.serial* > *node.serial* > *succ.serial* after line 86. Thus the required serial number inequalities of Property 1 **(iii)** hold at line 88.

Finally, by Property 2 **(iii)** of (IH), every active node with a serial number strictly between *node* and *succ* is marked after the last execution of line 85, and every active node with a serial number strictly between *pred* and *node* is marked by Property 2 **(iii)** of (IH) and the fact that *pred* was active and *pred.next* = *node* during FindPred. In addition, *node* itself is marked by line 85 and the check at line 86. So after the last execution of line 85, every active node z with *pred.serial* > *z.serial* > *succ.serial* is marked, proving Property 1 **(iv)** holds at line 88.

Case 2. MR has completed at least one iteration of the loop at lines 88–96.

By (IH), when the test at line 92 of the previous iteration was false, Property 1 **(i)**–Property 1 **(iv)** held. Since *pred*, *node*, and *succ* have not changed between that time and the beginning of the loop at lines 88–96, they are still true.

□

Claim 30.2. *Property 1* ***(ii)****–Property 1* ***(iv)*** *hold after each iteration of* MR*'s loop at lines 89–90 before s is performed.*

Proof. By Claim 30.1, Property 1 **(ii)**–Property 1 **(iv)** hold before entering the loop at lines 89–90. Assume that properties Property 1 **(ii)**–Property 1 **(iv)** hold after some arbitrary number of iterations of the loop at lines 89–90 ((IH) of Claim 30.2). We show Property 1 **(ii)**–Property 1 **(iv)** hold at the end of that iteration.

Property 1 **(ii)** holds because *succ* is active at the start of the iteration by (IH) of Claim 30.2, and *pred* is then set to *pred.backlink* at line 90, which by Lemma 27 is an active node.

Property 1 **(iii)** holds because $pred.serial > node.serial > succ.serial$ at the start of the iteration, and by Lemma 27, *pred* is updated at line 90 to a node $pred'$ that is active and previously pointed to *pred*. By Property 2 **(ii)** of (IH), $pred'.serial > pred.serial$, which implies Property 1 **(iii)** holds at the end of the iteration.

Property 1 **(iv)** follows similarly: by Lemma 27 and Property 2 **(iii)** of (IH), all nodes with serial numbers strictly between *pred* and $pred' = pred.backlink$ are marked, and *pred* itself is marked by the check at line 89. Thus, when *pred* is updated to $pred'$ at line 90, Property 1 **(iv)** holds. □

Claim 30.3. *Property 1* ***(ii)****–Property 1* ***(iv)*** *hold after each iteration of* MR*'s loop at lines 92–93 before s is performed.*

Proof. By Claim 30.2, Property 1 **(ii)**–Property 1 **(iv)** hold after the final execution of the loop at lines 89–90. So they hold at the beginning of the first iteration of the loop at lines 92–93. Assume these properties hold at the beginning of some iteration of the loop at lines 92–93 ((IH) of Claim 30.2). We now show that they continue to hold at the end of that iteration.

Let $succ'$ denote the value of *succ* after it is updated to *succ.next* at line 93. We first show that $succ'$ is active, that $succ.serial > succ'.serial$, and that all nodes with serial numbers in $(\, succ'.serial,\ succ.serial \,]$ are marked after the iteration. By (IH) of Claim 30.3 on Property 1 **(ii)**, *succ* is active. Then, by Invariant 16 and the read at line 93, *succ*'s next node $succ'$ is also active. Since *succ* is marked at line 92, Lemma 29 ensures that $succ \neq tail$;

hence, by Property 2 **(i)** of (IH) we have $succ' \neq \texttt{NIL}$. Furthermore, by Property 2 **(ii)** and Property 2 **(iii)** of (IH), we have $succ.serial > succ'.serial$ and all nodes with serial numbers in $(\, succ'.serial,\ succ.serial \,)$ are marked. Since $succ$ is marked by the check at line 92, and because serial numbers are distinct (Corollary 22) and marked nodes never become unmarked (Observation 28), it follows that all nodes with serial numbers in $(succ'.serial, succ.serial]$ are marked.

We now show each property in Claim 30.3 holds after the iteration. Property 1 **(ii)** holds because $succ'$ is active. Property 1 **(iii)** holds because, by Property 1 **(iii)** of (IH) of Claim 30.3, we have $pred.serial > node.serial > succ.serial$, and we have already shown $succ.serial > succ'.serial$. Finally, Property 1 **(iv)** holds because, by Property 1 **(iv)** of (IH) of Claim 30.3, all nodes with serial numbers in $(succ.serial, pred.serial)$ are marked at the start of the iteration, and we have shown that all nodes with serial numbers in $(succ'.serial, succ.serial]$ are also marked. Therefore, using the fact that marked nodes never become unmarked (Observation 28), all nodes with serial numbers in $(succ'.serial, pred.serial]$ remain marked. Thus, Property 1 **(ii)**–Property 1 **(iv)** hold after the iteration. □

Because Property 1 **(ii)**–Property 1 **(iv)** hold after each iteration of the loop at lines 92–93, and we have shown that Property 1 **(i)** holds, Property 1 **(i)**–Property 1 **(iv)** hold when the condition at line 92 is evaluated as false.

Proof of Property 2

Now we show that Property 2 **(i)**–Property 2 **(iv)** hold by showing they hold after s makes a new node active at line 33 or s sets the link of an active node x at line 95.

Case 1. If a node x is made active by a successful CAS at line 33 of PUBLISH, $x.next = y$ immediately after that CAS, where $Head = y$ at line 27. By Observation 15, y is an active node, so we have $y \neq \texttt{NIL}$ (Property 2 **(i)**). Furthermore, we have $x.serial > y.serial$ by the fact that $x.serial = y.serial + 1$ by line 32 (Property 2 **(ii)**). Property 2 **(iii)** and Property 2 **(iv)** hold because no node can have a serial number strictly between $x.serial = y.serial + 1$ and $y.serial$ since serial numbers are integers (by Corollary 22).

Case 2. If $x.next$ is set to y by a successful CAS at line 95, then $pred = x$ and $succ = y$, and $pred.next = curr$ immediately before the CAS.

Property 2 **(i)** holds because, by Property 1 **(i)** of (IH), $succ \neq \texttt{NIL}$ is an unmarked

node when the condition at line 92 was evaluated as `false` before line 95. So $succ = y \neq$ `NIL` after line 95.

By Property 1 **(iii)** of (IH), we have $pred.serial > succ.serial$ at line 95, and by Observation 19 the serial number of a node never changes, so we have $x.serial > y.serial$ after line 95.

By Property 1 **(iv)** of (IH), every active node with a serial number in the range $(succ.serial, pred.serial)$ is marked. By Observation 28, marked nodes never become unmarked, so all active nodes with serial numbers strictly between $pred.serial = x.serial$ and $succ.serial = y.serial$ are marked after line 95.

Finally, we prove Property 2 **(iv)**. Assume, for contradiction, that the step s produces a configuration with active nodes x and x' such that $x'.next = y' \neq$ `NIL`, $x.next = y \neq$ `NIL`, and $x'.serial > x.serial > y'.serial > y.serial$. If the link of x or x' is set at line 33, there cannot be a violation of Property 2 **(iv)**, as explained in **Case 1.**. Hence, the links of x and x' must be set by successful CAS operations at line 95. Let s be the later of these two steps.

Subcase 1. The link $x'.next$ is set to y' before the link $x.next$ is set to y by s.

Because x' is active, by Property 2 **(iii)** of (IH), every node with a serial number strictly between $x'.serial$ and $y'.serial$ is marked when $x'.next$ is set to y'. This implies that x is marked then. By Observation 28, x never becomes unmarked, so x is still marked in the configuration before s. This contradicts the assumption that s performs a successful CAS at line 95, since that CAS cannot succeed if the link is marked, so the assumption is false.

Subcase 2. The link $x.next = y$ is set before the link $x'.next = y'$ is set by s.

For an operation to set $x'.next$ to y' at line 95, it must be the case that y' is unmarked and that all active nodes with serial numbers between $x'.serial$ and $y'.serial$ are marked when the condition at line 92 was evaluated as false (by Property 1 **(i)** and Property 1 **(iv)** of (IH)). This contradicts the fact that x was unmarked when $x.next$ was set

to y by the CAS at line 95, and y' must be marked at that moment by the (IH) of Property 2 **(iii)**.

□

We now show that the link of a node can only be updated to point to a node with a lower serial number than the node it currently points to.

Lemma 31. *If there exists an active node $x \neq tail$ such that $x.next = y$ in C and $x.next = y' \neq y$ in C' where $C \prec C'$, then $y'.serial < y.serial$.*

Proof. By Property 2 **(i)** of Lemma 30, $x.next \neq \texttt{NIL}$. The field $x.next$ is updated from one node to another only by a CAS at line 95. Furthermore, by Property 2 **(ii)** of Lemma 30, $x.next$ is only updated at from y to a node y' with $y'.serial < x.serial$. Assume that $x.next$ is updated to a node y' with $y.serial < y'.serial < x.serial$ by a successful CAS at line 95. By Property 2 **(iii)** of Lemma 30, because $y.serial < y'.serial < x.serial$, y' is marked when $x.next = y$ was read at line 91 before the CAS. By Observation 28, this contradicts the fact that y' is unmarked at the execution of line 92 before the CAS. Therefore, $x.next$ cannot be updated to the node y'. □

We now show that if an active node z is skipped by the link of a node x in some configuration, then in that configuration and all subsequent ones, z cannot be reached from any node x' with $x'.serial \geq x.serial$.

Lemma 32. *If an active node z is unreachable from an active node x in some configuration C, then z is unreachable from every active node x' with $x'.serial \geq x.serial$ in all configurations C' such that $C \preceq C'$.*

Proof. Suppose z is unreachable from an active node x in configuration C. Let y be the last node reachable from x such that $y.serial > z.serial$. By Property 2 **(ii)** of Lemma 30, serial numbers strictly decrease along along the path of nodes formed by next pointers starting from x. By Corollary 22, each active node has a unique serial number. Hence, in C, there exists an active node y such that $y.next = y'$ and $y.serial > z.serial > y'.serial$.

By Lemma 31, if $y.next$ is updated to some node y'' between C and C', we have $y''.serial < y'.serial$. Thus, in C', we have $y.next = y''$ with $y''.serial \leq y'.serial$. By Property 2 **(iv)** of Lemma 30, no node with a serial number larger than $y.serial$ can point to a node whose

serial number is in $(\, y''.serial,\ y.serial\,)$, which includes $z.serial$. Hence no new link from any node with a serial number greater than or equal to y—including x (since y is reachable from x in C by Invariant 16 and Property 2 **(ii)** of Lemma 30) or any node with serial number larger than x—can point to z in C'.

Therefore, by Property 2 **(ii)**, once z becomes unreachable from x in configuration C, it remains unreachable from all nodes x' with $x'.serial \geq x.serial$ in every later configuration C'. □

The contrapositive of Lemma 32 implies the following corollary by taking x' as x and the node y as z.

Corollary 33. *If $x.next = y$ for an active node x in a configuration C', then in every configuration $C \preceq C'$, $y \in R(x, C)$.*

This implies that a traversal starting from a node x can only reach nodes that were reachable from x before the traversal began.

Lemma 34. *If a traversal that begins from node x in configuration C' reaches a node z, then $z \in R(x, C)$ for every configuration $C \preceq C'$.*

Proof. When no steps have executed, no traversals have begun, so the lemma holds vacuously. Assume it holds for some prefix α of an execution (IH), and consider the next step s. Since a node is always reachable from itself, a step that begins a traversal at x cannot violate the lemma. A violation could only occur if s is a step of a traversal that began at x in configuration C' and reads a node z such that $z \notin R(x, C)$ for some configuration $C \preceq C'$.

The step s must reach z by following a pointer of a node x' that was already reached from x; that is, it read $x'.next = z$. Let C'' be the configuration in which this read occurred. By (IH), $x' \in R(x, C)$ for all $C \preceq C'$, and by Corollary 33, $z \in R(x', C)$ for all $C \preceq C''$. Since $C' \preceq C''$, we have $z \in R(x', C)$ for all $C \preceq C'$. Thus, z is reachable from x in all configurations $C \preceq C'$ by concatenating the path that is witness to $x' \in R(x, C)$ with the path that is witness to $z \in R(x', C)$. □

Now we show that if a node x is unmarked, it can be reached from every active node newer than x.

Lemma 35. *Let x and y be active nodes with $x.serial < y.serial$. If x is unmarked, then*

every traversal from y that reaches a node with a serial number less than or equal to $x.serial$ must reach x.

Proof. Assume an active node x is unmarked and it was not reached by a traversal beginning from an active node y where $x.serial < y.serial$ that reaches a node with a serial number less than or equal to $x.serial$. If it reaches a node with a serial number equal to $x.serial$, then it has reached x because the serial numbers of active nodes are unqiue (by Observation 18), which contradicts the assumption that x was not reached.

Assume the traversal reached a node with a serial number less than $x.serial$. Let $r_0, r_1, \ldots, r_i$ be the nodes traversed by the traversal from y in the order they were reached, with $r_0 = y$. By Property 2 **(ii)** of Lemma 30, we have $r_j.serial > r_{j+1}.serial$ for $(0 \leq j < i)$, and for x to be skipped by the traversal, we must have $r_j.serial > x.serial > r_{j+1}.serial$ for some $0 \leq j < i$. By Property 2 **(iii)** of Lemma 30, this implies x is marked, which contradicts the assumption x is unmarked. Therefore the assumption that x is not reached by a traversal from y that reaches a node with a serial number less than or equal to $x.seria$ is false. □

We now show that at some moment during MarkAndRemove(x, y), the node y cannot be reached from $Head$.

Lemma 36. *At some moment during a call to* MarkAndRemove(x, y) *that terminates, there is an active node x' with $x'.next = y'$ such that $x'.serial > y.serial > y'.serial$.*

Proof. MarkAndRemove(x, y) exits the loop at lines 88–96 when $pred.next.serial \leq succ.serial$ at line 96. By Property 1 **(ii)** and Property 1 **(iii)** of Lemma 30, when the condition at line 92 was last evaluated as false, $pred$ was active and $pred.serial > y.serial > succ.serial$. Since $pred$ and $succ$ remain unchanged between that moment and line 96, we have $pred.serial > y.serial > succ.serial > pred.next.serial$ at line 95. Taking $x' = pred$ and $y' = pred.next$ proves the lemma. □

Finally, we prove that once MarkAndRemove(x, y) completes, the node y is no longer reachable from $Head$ in any subsequent configuration.

Lemma 37. *Let C be the configuration when* MarkAndRemove(x, y) *completes. The node y is unreachable from $Head$ in any configuration C' where $C \preceq C'$.*

Proof. By Lemma 36, during the execution of MarkAndRemove(x, y), there is a moment

when an active node x' satisfies $x'.next = y'$ with $x'.serial > y.serial > y'.serial$. By Lemma 32, y is unreachable from any active node whose serial number is greater than or equal to $x'.serial$. Finally, by Corollary 22 **(ii)**, the sentinel node always has the largest serial number among active nodes. Together, these observations establish the desired claim. □

4.6 Delete Ensures Matches are Removed

In this section, we show that a delete node x is marked only after every match of x that can be reached before x's newest matching delete node (or the end of the list if no matching delete node exists) is unreachable from x (Invariant 39). This ensures that traversals in lines 40–49 of MTF and lines 64–67 of RemoveMatches, when starting from a node appended after x, cannot reach a matching insert node appended before x, thereby preventing them from potentially returning incorrect results (see Invariant 42).

We first show that a delete node x can be marked only by the same Delete operation that claimed it—that is, the Delete that published its OpData object d to x at line 29 of its call to Publish(d) at line 23.

Lemma 38. *If x is a delete node, it can be marked only by the call to* MarkAndRemove *at line 75 within* RemoveMatches(x)*, which is invoked exclusively by the* Delete *operation that claimed x.*

Proof. MarkAndRemove($*$, x) marks a node x at line 84. We examine all call sites of MarkAndRemove($*$, y) other than the call at line 75 of RemoveMatches(x) and show that $y \neq x$.

Case 1. At line 56, MarkAndRemove($*$, y) is called on ($pred$, $curr$), where $curr$ is a node read at line 38 or line 47 and saved in *first* at line 44 of the loop at lines 40–49. Such a node cannot be a delete node, as ensured by the loop guard at line 40. Thus $y \neq x$.

Case 2. At line 58, MarkAndRemove($*$, y) is called with the second argument equal to the search node appended in Search at line 16. This node has $type =$ `search`, so $y \neq x$ because $x.type =$ `delete`.

Case 3. At line 74, MarkAndRemove($*$, y) is called on a pair from the `matches` vector. These matches are collected in the loop at lines 64–67. The test on line 64 and line 65 ensure that y is not a delete node when $\langle *, y \rangle$ is added to *matches*.

Case 4. At line 75 of a call RemoveMatches(y) where $y \neq x$, MarkAndRemove($*$, y) is called with $y \neq x$.

Therefore, the only call of the MarkAndRemove procedure that can mark x is the call MarkAndRemove($*$, x) at line 75 of RemoveMatches(x).

Since each operation claims a unique node (by Lemma 21), RemoveMatches(x) is invoked only by the Delete operation that claimed x (by lines 23–24 and Observation 17).

□

Invariant 39. *Let x be a delete node claimed by a* Delete(k) *operation.*

Let $\mathcal{D} = \{d \mid d \text{ is an active delete node, } d.key = k, \ d.serial < x.serial\}$, and define

$$m = \begin{cases} \max_{d \in \mathcal{D}} d.serial & \text{if } \mathcal{D} \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

If x is marked, then for every node y such that

$$y.key = k \quad \text{and} \quad x.serial > y.serial > m,$$

the node y is not reachable from x.

Proof. Initially, before any steps are taken, the invariant holds trivially. Assume the invariant holds after some execution prefix α. We show it continues to hold after the next step s.

By Lemma 32, once a node becomes unreachable from x, it cannot later become reachable from x. Therefore, step s cannot make a node y that was unreachable from a marked x become reachable from x.

To violate the invariant, s would have to be a step that marks a delete node x. By Lemma 38, this can only occur in a call to MarkAndRemove($*, x$) at line 75 in RemoveMatches, where x is the node claimed by the Delete operation that invoked RemoveMatches(x) (see line 24 and line 23).

RemoveMatches(x) traverses the list starting from x (at lines 60–67), proceeding through

nodes with strictly decreasing serial numbers (by Property 2 **(ii)** of Lemma 30), until it encounters a matching delete node for key k (whose serial number must be less than or equal to m) or the end of the list (see the exit condition at line 64).

Let $x_0, x_1, \ldots, x_i$ be the nodes visited by this traversal, where $x_0 = x$ and x_i is the node stored in *curr* at the ith execution of the test on line 65 for $i > 0$. Since x_{j+1} is read from the *next* field of x_j (at line 61 or 66), it follows from Property 2 **(ii)** of Lemma 30 that $x_j.serial > x_{j+1}.serial$ for all $j < i$. Fix any node y with $y.key = k$ and $m > y.serial > x.serial$. We show that y is not reachable from x when x is marked.

Case 1. y **is visited (i.e.,** $y = x_j$ **for some** $j > 0$**).** Then $\langle x_{j-1}, x_j \rangle$ is added to the *matches* vector at line 65, and MarkAndRemove(x_{j-1}, x_j) is called at line 74, before x is later marked at line 75. By Lemma 36, there is a moment during MarkAndRemove(x_{j-1}, x_j) at which an active node x' satisfies $x'.next = y'$ and $x'.serial > y.serial > y'.serial$. By Lemma 32, y cannot be reached from any node whose serial number is greater than or equal to $x'.serial$ from that moment onward.

Since x remains unmarked during MarkAndRemove(x_{j-1}, x_j), we must have $x'.serial \leq x.serial$. This is because $y.serial < x.serial$ since y was reached from x (by Invariant 16 and Property 2 **(ii)** of Lemma 30). So if $x.serial > x'.serial$, then x would already be marked when $x'.next = y'$, since every node in $(y'.serial, x'.serial)$ is marked by Property 2 **(iii)** of Lemma 30–contradicting the fact that x is unmarked throughout MarkAndRemove(x_{j-1}, x_j).

Using the fact that $x.serial \geq x'.serial$, by Lemma 32, x cannot reach y at the moment when $x'.next = y'$ holds in MarkAndRemove(x_{j-1}, x_j), or at any later time. Consequently, when x is later marked by s in the call to MarkAndRemove at line 75, y cannot be reached from x.

Case 2. y **is not visited**. Then y has a serial number strictly between x_j.serial and x_{j+1}.serial for some j. By Lemma 32, y is unreachable from any node whose serial number is at least x_j.serial, including x.

In both cases, y is unreachable from x when x is marked, so the invariant is preserved. □

4.7 Correctness of MTF(x)

We now prove the correctness of MTF(x), which determines whether the key $k = x.key$ was in the set at the moment x was claimed. Intuitively, MTF(x) should return `true` if k was contained in the set at that time, and `false` otherwise.

Because operations claim nodes in strictly increasing order of serial number (by Lemma 21), and we linearize operations when they claim a node (by Definition 13), k was in the set when x was claimed if and only if a node u was claimed by an INSERT(k) operation with $u.serial < x.serial$, such that no DELETE(k) claimed a node with a serial number in $(u.serial, x.serial)$.

To formalize this, we define the set U_x.

Definition 40. *Let x be a node claimed by an operation on the key k. Then U_x is defined to be the set of nodes u such that:*

- $1 \leq u.serial < x.serial$, *and*
- $op(u)$, *the operation that claimed u, is either an* INSERT(k) *or a* DELETE(k).

Since MTF(x) returns `true` exactly when $x.end > 0$ at line 59, it suffices to establish the following equivalence to prove the correctness of MTF(x):

$$\begin{aligned} x.end > 0 \text{ at line 59 of MTF}(x) \iff \exists\, u \in U_x \text{ such that } \mathrm{op}(u) = \textsc{Insert}(k), \text{and} \\ \forall\, u' \in U_x,\ \Big(u.serial < u'.serial < x.serial \Rightarrow \mathrm{op}(u') \neq \textsc{Delete}(k)\Big) \end{aligned} \tag{4.1}$$

Equation (4.1) implies that MTF(x) returns `true` if and only if k was in the set when x was claimed, and `false` otherwise. (When $U_x = \varnothing$, the right-hand side is false, so MTF(x) should return `false`.) We prove Equation (4.1) formally in Lemma 46, using the helper lemmas presented in Lemma 41–Lemma 45.

We begin with the following observation.

Lemma 41. *For every active node x, we have $x.end = -1$ initially, and $x.end$ is monotonically increasing thereafter.*

Proof. When our list is initialized, both dummy nodes in the list are assigned $end = -1$ (see Figure 3.7 and Section 3.2.2). When a new node becomes active through a successful

CAS at line 33, it also has $end = -1$ by line 32 (and the initial value of end as shown in Figure 3.7). Hence, the first claim of the lemma follows.

Only a call IncreaseEnd(x, end) can modify $x.end$ (at line 78). Such a call reads $x.end$ as $oldEnd$ at line 77 and performs a CAS to set $x.end$ to end only if $oldEnd < end$. Therefore, every successful update strictly increases the value of $x.end$, proving the second claim of the lemma. □

Now we prove an invariant that helps us prove the only if direction: if $x.end > 0$, then the newest matching insert or delete node of x is an insert node. Recall that an insert node is a node that either was claimed by an Insert operation, or a search node claimed by a Search operation that was promoted to an insert node.

Invariant 42. *If $x.end > 0$ for an appended node x with $x.key = k$, then there is an insert node z with $z.key = k$ and $z.serial < x.serial$ such that for every node y with $y.key = k$ where $z.serial < y.serial < x.serial$, $y.type \neq$ `delete`.*

Proof. When the algorithm has taken no steps, the invariant vacuously holds because no nodes have been appended. Assume the invariant holds for some prefix α of an execution. We show that the invariant continues to hold after the next step s of the execution.

The step s can violate the invariant only if $x.end$ is set to a positive value for some appended node x. The field $x.end$ is updated at line 78 in a call to IncreaseEnd(x, end), where $end > 0$. IncreaseEnd is invoked at lines 43, 49, 50, and 73, but a call to IncreaseEnd(x, end) at line 50 cannot violate the invariant since $end = 0$ in that case.

Before proving Invariant 42 by cases, we show a useful claim.

Claim 42.1. *Suppose a traversal from x in* MTF(x) *or* RemoveMatches(x) *(from its initialization through its main loop—for* MTF*, lines 37–49; for* RemoveMatches*, lines 60–67) reaches a node $curr$ with key k, and no matching delete node has been encountered. Then there is no delete node u with key k and $x.serial > u.serial > curr.serial$.*

Proof. Assume, for contradiction, that such a delete node u exists at a moment when $curr$ is reached as described. Let $x_0, x_1, \ldots, x_r$ be the nodes reached by the traversal, where $x = x_0$, $x_r = curr$, and $x_j.next = x_{j+1}$ for all $0 \leq j < r$.

Case 1: u was visited by the traversal.

Both traversals terminate upon reading a matching delete node: in MTF, by the second exit condition at line 40; in RemoveMatches, by the condition at line 64. Hence if u had been read, the traversal would have halted before reaching *curr*, which contradicts the assumption that *curr* is reached.

Case 2: u **was not visited by the traversal.**

By Lemma 21, every node with serial number in $[1, x.serial)$ is already claimed before the traversal begins, so u must have been claimed by a Delete(k) operation before the traversal began. Since each traversal follows links $x_j.next = x_{j+1}$ with strictly decreasing serial numbers (by Lemma 30 Property 2 **(ii)**), skipping u implies that there exists some nodes x_j and x_{j+1} with

$$x_j.serial > u.serial > x_{j+1}.serial.$$

By Lemma 30 Property 2 **(iii)**, u must have been marked when that link was followed by the traversal. Then, by Invariant 39, no matching node of u with a smaller serial (in particular *curr*) is reachable from u at that time. By Lemma 32, if *curr* is not reachable from u at that moment, it is not reachable from x_j at that moment either. Thus, by Lemma 34, a traversal beginning from x_j at that moment cannot reach *curr*, contradicting the assumption that the traversal reaches *curr*.

Both cases are impossible; hence, no such delete node u can exist. □

Now we prove Invariant 42 by cases.

Case 1: If IncreaseEnd is called at line 43 on an insert node z with $z.key = k$ (identified at line 41), then z was reached from x by the traversal of MTF(x) at lines 37–49. The exit condition of the loop at lines 40–49 guarantees that no delete node with key k was encountered between x and z. Hence, by Claim 42.1, there is no delete node y such that $x.serial > y.serial > z.serial$.

Case 2: If IncreaseEnd(x, $q.serial$) is called at line 49 of MTF(x), then q is an insert or search node with key k that was reached from x (see lines 37–38 and the loop at lines 40–49), without finding any delete nodes with key k on the path from x to q (see the second exit condition at line 40), and $q.end > 0$ (by the second condition of the if statement at line 49).

By Claim 42.1, no node y with $q.serial < y.serial < x.serial$ can be a delete node. By (IH), because $q.end > 0$ there must be an insert node z with $z.serial < q.serial$ with $z.key = k$ such that every node y with $y.key = k$ and $q.serial < y.serial < z.serial$ has $y.type \neq$ `delete`. So z is the required node with $z.key = k$ and $z.serial < x.serial$ and every node y with $y.key = k$ and $z.serial < y.serial < x.serial$ has $y.type \neq$ `delete`, so the invariant holds.

Case 3: If IncreaseEnd is called at line 73, it is invoked on the second node of a pair $matches[i]$ with $i < oldestInsert$, where $oldestInsert$ is the index of another pair in $matches$.

Each entry in $matches$ is a pair $\langle pred', curr' \rangle$ representing two consecutive nodes reached by the traversal of RemoveMatches(x) at lines 60–67, which begins from x. Here, $curr'$ is a node with the key k (see line 65), and $pred'$ is the node read as its predecessor (see lines 60–61 and 66–67).

Because each pair is recorded at line 65 before the traversal advances to the next node (at lines 66–67), $matches$ preserves the traversal order. Let z be the second node in $matches[oldestInsert]$. Since $i < oldestInsert$, the second node in $matches[i]$ was reached before z. By Property 2 **(ii)** of Lemma 30, $x.serial > z.serial$ and all nodes encountered between x and z have serial numbers strictly between $x.serial$ and $z.serial$. By the exit condition of the loop at line 64, the traversal never encountered a delete node with the key k between x and z. Hence, by Claim 42.1, no delete node y with $y.key = k$ and $z.serial < y.serial < x.serial$ exists. Finally, we have $z.type =$ `insert` by line 70.

□

Recall that a node can become an insert node if it was claimed by an Insert, or if a Search promoted a search node to an insert node at line 52. We now show that if there exists an insert node z with the key k, then there must exist a node u originally claimed by an Insert(k) operation with $u.serial \leq z.serial$ such that there is no delete node with the key k and a serial number in the range $[u.serial, z.serial]$. Consequently, whenever a traversal encounters an insert node z with the key k, we can trace z back to a valid Insert(k) operation without any intervening deletes.

Invariant 43. *Suppose there exists a node z with key k and $z.type =$ `insert`. Then there exists a node u with $u.serial \leq z.serial$ that was claimed by an* Insert(k) *operation such*

that no node with key k and a serial number in $[u.serial, z.serial]$ is of type `delete`*.*

Proof. The invariant is trivial if z was claimed by an INSERT(k) operation (take $u = z$). Otherwise, z was claimed by a SEARCH(k) operation and it was later changed to an insert node. Let s be the step at line 52 that updates $z.type$ to `insert` and assume the invariant holds before s (IH). We show that the invariant holds in the configuration after s.

The step s can take place only if $z.end > 0$ at line 51. By Invariant 42, this implies the existence of an insert node u' with $u'.serial < z.serial$ such that no delete node with key k has a serial number in $(u'.serial, z.serial)$.

By (IH), applied to u', there exists a node u with $u.serial \leq u'.serial$ that was claimed by an INSERT(k) and such that no delete node with key k has a serial number in $[u.serial, u'.serial]$.

Thus, no delete node with key k has a serial number in $[u.serial, z.serial)$, and since z is not a delete node (and serial numbers are unique by Lemma 21), no delete node with key k has a serial number in $[u.serial, z.serial]$. We also have $u.serial \leq u'.serial < z.serial$, so the invariant holds after s. □

Lemma 45 states that if the newest matching update node for a node x at the time x is claimed is an insert node $z(x)$ (see Definition 44), then during the traversal of MTF(x), once a node with the same key k and a serial number less than or equal to $z(x).serial$ is reached, the condition $x.end \geq z(x).serial$ holds at the next evaluation of the loop condition at line 40 (see **Property 5** of Lemma 45). This guarantees that $x.end > 0$ at line 51 of MTF(x) (by Corollary 22 **(i)** and Lemma 41), which is used to prove *if* direction of the correctness condition of MTF(x) (Equation (4.1)).

Although proving $x.end > 0$ holds at line 51 of MTF(x) would suffice for showing the correctness of MTF(x), we strengthen the statement to $x.end \geq z(x).serial$. This stronger bound is later used in the complexity analysis (Section 5.6) to show that the traversal length of MTF(x) is proportional to the working set size of the operation that claimed x, up to an additive contention term.

We use **Property 1**–**Property 4** of Lemma 45 to prove **Property 5** of Lemma 45; they help us ensure that $x.end \geq z(x).serial$ remains true even as nodes with the key k and serial numbers in the range $[z(x).serial, x.serial]$ are marked, and potentially removed from MTF(x)'s traversal path. We also use **Property 3** to justify that there always exists an unmarked insert node with the key k and a serial number in $[z(x).serial, x.serial)$ if x is

unmarked (in Theorem 49).

Definition 44. *Let x be a node claimed by either* SEARCH(k) *or* INSERT(k). *Define $z(x)$ to be the newest node z such that when x was claimed with $z.key = k$, $z.type =$ `insert`, $z.serial < x.serial$, and no* `delete` *node with key k has a serial number in $(z.serial, x.serial)$. If no such z exists, define $z(x)$ to be* `NIL`.

Lemma 45. *Suppose x is a node claimed by a* SEARCH(k) *or an* INSERT(k) *operation, and that $z(x) \neq$* `NIL`.

Each node y with key k and serial number in $(z(x).serial, x.serial]$, (which must be a search or insert node, by definition of $z(x)$) satisfies the following properties.

Property 1. $z(y) \neq$ `NIL`.

Property 2. *If y is marked, then $y.end \geq z(y).serial$.*

Property 3. *If y is unmarked and every node with key k and serial number in the range $[z(x).serial, y.serial)$ is marked, then $y.end \geq z(y).serial$ and $y.type =$ `insert`.*

Property 4. *Suppose the call* MTF(x) *reaches a node with serial $\leq z(x).serial$ in some step s. Let*

$$y' = \begin{cases} x, & \text{if } last = \texttt{NIL}, \\ last, & \text{otherwise,} \end{cases}$$

in C_s, where last is the local variable of MTF(x) *initialized at line 36. Then every node with key k and a serial number in $(z(x).serial, y'.serial)$ is marked in C_s.*

Property 5. *If* MTF(x) *reaches a node whose serial number is at most $z(x).serial$ at line 38 or line 47, then $x.end \geq z(x).serial$ holds before the next execution of line 40.*

Proof. Assume the lemma holds for some execution α, we show it holds for $\alpha \cdot s$, where s is a step (IH).

We first prove **Property 1**.

By Invariant 43, the existence of $z(x)$ implies a node u with $u.serial \leq z(x).serial$ was claimed by an INSERT(k) operation such that no delete node with key k has a serial number in $[u.serial, z(x).serial]$. By Lemma 21, u was claimed before y. Since $z(y)$ is defined as the newest insert node when x was claimed with key k and a serial number less than $y.serial$,

and there is no delete node with key k has a serial number in $(z(y).serial, y.serial)$, the existence of u guarantees that such a node exists, and therefore $z(y) \neq \texttt{NIL}$.

Before showing **Property 2**–**Property 4** hold after a step s, we show some useful claims.

Recall x is a node claimed by a SEARCH(k) or an INSERT(k) operation, $z(x) \neq \texttt{NIL}$, and each node y with the key k and a serial number in $(z(x).serial, x.serial]$ in an insert or search node.

Claim 45.1. *If* REMOVEMATCHES *performs a* CAS *on $y.end$ at line 78 in an invocation of* INCREASEEND$(y, *)$ *at some step $s' \prec s$, then $y.end \geq z(y).serial$ for every step after s'.*

Proof. If a call REMOVEMATCHES(x') calls INCREASEEND(y, $*$) at line 73, the node y must appear as the second node of a pair in the *matches* vector of REMOVEMATCHES(x'). Thus, y is a node with the same key as x' that was reached from x' during the traversal at lines 60–67 (see line 65). Hence, by Property 2 **(ii)** of Lemma 30, for y to be a member of *matches*, we must have $x'.serial > y.serial$.

When REMOVEMATCHES(x') sets $y.end$, either

Case 1. y is marked,

Case 2. y is unmarked and all nodes with key k and a serial number in $(z(y).serial, y.serial)$ are marked, or

Case 3. y is unmarked, and some node y'' with key k and a serial number in the range $(z(y).serial, y.serial)$ is unmarked.

In **Case 1** and **Case 2**, by **Property 2** and **Property 3** of (IH), $y.end \geq z(y).serial$ is true, so by Lemma 41, it remains true after s. In **Case 3**, by Lemma 35, since $x'.serial > y''.serial$ and y'' is unmarked before s, the traversal in lines 60–67 of REMOVEMATCHES(x') will reach y'' and collect it at line 65 if the traversal reaches a node with serial number less than or equal to $y''.serial$.

The traversal in REMOVEMATCHES(x') continues until it encounters a matching delete node for x' or the *tail* node with serial 0 (line 64). Because there is no delete node with key k in $(z(y).serial, x'.serial)$ and the traversal visits (and collects) nodes in strictly decreasing serial order (by Property 2 **(ii)** of Lemma 30), it must eventually reach a node with serial less than or eqaul to $y''.serial$. Hence, the traversal reaches y'' and collects it at line 65.

Because nodes are collected in *matches* in order of serial number, the argument *end* passed to IncreaseEnd(y, *end*) at line 78 is at least $y''.serial$. Hence, after RemoveMatches(x') sets $y.end$, we have $y.end \geq y''.serial \geq z(y).serial$, and by Lemma 41, $y.end \geq z(y).serial$ holds thereafter. □

Claim 45.2. *Assume* MTF(y) *reaches a node with a serial number less than* $z(y).serial$ *at line 38 or line 47 before* s. *Let* z *be the first such node encountered by* MTF(y). *Then either:*

- **(i)** $y.end \geq z(y).serial$ *holds immediately after* z *is reached (at line 38 or 47),*
- **(ii)** z *is reached at line 47, and* MTF(y) *calls* IncreaseEnd($y, last.serial$) *at the next execution of line 49, or*
- **(iii)** z *is reached at line 47, and* MTF(y) *has already called* IncreaseEnd($y, z(y).serial$) *at line 43.*

Proof. Let s' be the step at line 38 or 47 where MTF(y) reaches z. In $C_{s'}$, the node $z(y)$ is either marked or unmarked.

Case 1. Assume $z(y)$ is marked in $C_{s'}$. By **Property 4** of (IH), every node with key k and serial in the interval ($z(y).serial$, $y'.serial$) is marked in $C_{s'}$, where

$$y' = \begin{cases} y, & \text{if } last = \texttt{NIL}, \\ last, & \text{otherwise.} \end{cases}$$

Thus, all nodes with key k and serial in $[z(y).serial, y'.serial)$ are marked in $C_{s'}$.

Subcase 1. If $y' = y$: If y is marked in $C_{s'}$, then **Property 2** of (IH) yields $y.end \geq z(y).serial$. If y is unmarked in $C_{s'}$, all nodes with key k and serial in ($z(y).serial$, $y.serial$) are marked, so **Property 3** of (IH) again gives $y.end \geq z(y).serial$. Thus **(i)** holds.

Subcase 2. If $y' = last$: Since $last = \texttt{NIL}$ at line 38, the step s' must occur at line 47. Because $last$ is visited by MTF(y) before reaching z (by lines 41 and 45), Property 2 **(ii)** of Lemma 30 implies $last.serial \in (z(y).serial, y.serial)$, and line 41 ensures $last.key = k$. Thus (IH) applies to $last$, and by **Property 1**, $z(last) \neq \texttt{NIL}$.

Applying **Property 2–Property 3** of (IH) to *last* yields $last.end \geq z(last).serial$ in $C_{s'}$. Since $z(last).serial > 0$ (by Corollary 22 **(i)**), we have $last.end > 0$ in $C_{s'}$, and this value only increases thereafter (by Lemma 41). Therefore, at the next execution of line 49 after s', MTF(y) invokes IncreaseEnd($y, last.serial$). Hence **(ii)** holds.

Case 2. If $z(y)$ is unmarked in $C_{s'}$, then by Lemma 35 and the fact that MTF(y) visits nodes in decreasing serial order (by Property 2 **(ii)** of Lemma 30), MTF(y) must have reached $z(y)$ at line 38 or 47 before s'. Since $z(y)$ is an insert node with key k, MTF(y) then invokes IncreaseEnd($y, z(y).serial$) at line 43 (via line 41) before s'. Thus, **(iii)** holds.

□

Claim 45.3. *If* MTF(y) *performs a* CAS *on* $y.end$ *at line 78 in a call to* IncreaseEnd($y, *$) *at some step* $s' \prec s$*, then* $y.end \geq z(y).serial$ *holds for every step after* s'*.*

Proof. MTF(y) can call IncreaseEnd(y, $*$) at line 43, 49, or 50. Consider the *first* CAS on $y.end$ at line 78 performed by a call to IncreaseEnd(y, $*$) during MTF(y). If this CAS fails, then $y.end$ must have been updated by a call to IncreaseEnd(y, $*$) in some other operation's RemoveMatches, so $y.end \geq z(y).serial$ holds before s by Claim 45.1. By Lemma 41, if $y.end \geq z(y).serial$ holds before s, it holds after s. Thus, we need only show that if the first CAS by MTF(y) succeeds, it sets $y.end \geq z(y).serial$, which is equivalent to showing that the second argument of IncreaseEnd(y, $*$) is greater than or equal to $z(y).serial$ (by line 78).

Case 1: MTF(y) first calls IncreaseEnd(y, $*$) at line 43 or line 49 immediately after reaching a node whose serial number is greater than or equal to $z(y)$ at line 38 or line 47.

Then $y.end \geq z(y).serial$, since nodes are visited in decreasing order of serial number (by Property 2 **(ii)** of Lemma 30), and the variable *last* always stores the most recently visited node with key k (lines 45 and 46–47).

Case 2: MTF(y) first calls IncreaseEnd(y, $*$) at line 43 or line 49 after reaching a node with a serial number less than $z(y).serial$ (at line 38 or line 47).

Let z be the first node encountered by MTF(y) with $z.serial < z(y).serial$. By

Claim 45.2, when z is reached at line 38 or line 47, we have:

(i) $y.end \geq z(y).serial$,

(ii) IncreaseEnd$(y, last.serial)$ is called at line 49 immediately after the read of z at line 47, or

(iii) IncreaseEnd$(y, z(y).serial)$ was executed at line 43 before reaching z.

If **(i)** holds, then Claim 45.3 holds straightaway, since $y.end \geq z(y).serial$ when z is reached, and by Lemma 41 this remains true after s. Because we are assuming that the *first* call to IncreaseEnd$(y, *)$ made by MTF(y) occurs only *after* reaching a node with serial number less than $z(y).serial$, **(iii)** is impossible. If **(ii)** holds, the first call to IncreaseEnd$(y, *)$ by MTF(y) is IncreaseEnd$(y, last.serial)$ at line 49, executed immediately after reaching z at line 47. We have $last.serial > z(y).serial$ because *last* records the most recent node with key k encountered by MTF(y) (lines 45 and 46–47). Since z is the first node encountered by MTF(y) with key k and a serial number is less than $z(y).serial$, Property 2 **(ii)** of Lemma 30 implies $last.serial > z(y).serial$.

Case 3: MTF(y) first calls IncreaseEnd$(y, *)$ at line 50.

Since the call IncreaseEnd$(y, 0)$ at line 50 successfully sets $y.end$ at line 78, we must have $y.end = -1$ throughout the traversal at lines 37–49. Hence, the loop at lines 37–49 can only exit by the first exit condition at line 40 if it reaches the *tail* node. The traversal starts at y (lines 37–38) and advances forward one node at a time (lines 46–47) with strictly decreasing serial numbers (Property 2 **(ii)** of Lemma 30). Because no delete node with key k exists with a serial number in $[z(y).serial, y.serial)$, the loop cannot stop at a node with a serial number in $[z(y).serial, y.serial)$, so before the loop exits, MTF(y) must reach a node z with a serial number smaller than $z(y).serial$. Hence, by Claim 45.2, either $y.end \geq z(y).serial$ holds when z is reached, or the call to IncreaseEnd$(y, *)$ by MTF(y) at line 50 is not the first. If $y.end \geq z(y).serial$ holds when z is reached, Lemma 41 $y.end \geq z(y).serial$ holds after s.

□

Proof of Property 2.

To show that **Property 2** is still true when the execution is extended by step s, it suffices to consider a step s that marks a node y with key k and a serial number in $[z(y).serial, x.serial)$.

Case 1: If a call to RemoveMatches(x') marks y, it must be at line 74, because line 75 marks a delete node (x' is a delete node by lines 23–24 and Observation 17). Then, RemoveMatches(x') must have collected y as the second node of a pair in the *matches* vector at line 65 before s. Let w be the oldest node in *matches* at line 68 of RemoveMatches(x') that is unmarked and has a serial number in $[z(y).serial, y.serial]$ (such a node exists and is at least as old as y because y is unmarked until s). Because RemoveMatches(x') adds every unmarked node with key k and a serial number in $[z(y).serial, y.serial)$ as the second node of a pair to *matches* at lines 64–65 (by Lemma 35, Property 2 **(ii)** of Lemma 30, and the fact that no delete node for key k has a serial number in $[z(y).serial, y.serial)$), any node with key k and a serial number in $[z(y).serial, y.serial)$ that is not the second node of a pair in *matches* at line 68 must have been marked. Thus all nodes with key k and serial in $[z(y).serial, w.serial)$ are marked at line 68, and since w is unmarked, **Property 3** of (IH) applies to w.

If $y = w$, we have $y.end \geq z(y).serial$ at line 68 by **Property 3** of (IH), so **Property 2** holds. So for the remainder of the case, assume $w.serial < y.serial$. Then $w.type =$ `insert` at line 68 by **Property 3** of (IH). So when RemoveMatches(x') reads *matches* in reverse order at lines 69–70, it will set *oldestInsert* to an index at least that of w's index, which is greater than the index of y in *matches* because nodes are collected in *matches* at line 65 in decreasing order of serial number by line 61, lines 66–67, and Property 2 **(ii)** of Lemma 30.

Hence, before marking y at line 74, RemoveMatches(x') will attempt to set $y.end$ in a call to IncreaseEnd(y, $*$) at line 73 (by the check at line 73). If $y.end$ is successfully set by RemoveMatches(x'), then by Claim 45.1, $y.end \geq z(y).serial$ holds after s. If it fails to do so, some RemoveMatches or MTF(y) must have set $y.end$, so by Claim 45.1 and Claim 45.3, $y.end \geq z(y).serial$ holds after s.

Case 2: MTF(y) marks y at line 58.
This line is executed only after MTF(y) attempts to set $y.end$ at line 50. If MTF(y) succeeds in setting $y.end$, then by Claim 45.3, $y.end \geq z(y).serial$, and if it fails, some call to RemoveMatches or MTF(y) itself must have set $y.end$ beforehand so by Claim 45.3 and Claim 45.1 $y.end \geq z(y).serial$ holds after s.

Case 3: MTF(y') with $y' \neq y$ marks y at line 56, then $y.end \geq 0$ by the same line.

If $y.end \geq 0$ at line 56 of MTF(y'), MTF(y) or RemoveMatches(x') must have set $y.end$ before s ($y.end = -1$ initially by Figure 3.9). Hence, by Claim 45.3 and Claim 45.1 $y.end \geq z(y).serial$ holds after s.

Proof of Property 3

Let y be a node with key k and serial number in $[z(x).serial,\ x.serial)$, and let $U = \{\text{ node } u \mid u.key = k \text{ and } z(x).serial \leq u.serial < y.serial\}$. A violation of **Property 3** can occur only if y is unmarked while every node in U is marked after step s. Let u be the newest node in U. Consider the step before or at s that marks u.

To derive a contradiction, suppose a call to RemoveMatches(x') for some node x' marked the node u. Since no delete node with the key k with a serial number in $[z(y).serial,\ y.serial)$ exists, the delete node x' must have $x'.serial > y.serial$ (by Property 2 **(ii)** of Lemma 30). Since RemoveMatches(x') marks nodes in the order they are reached during its traversal, it would have reached and marked y before u (by Lemma 35), contradicting the fact that y is unmarked at s. Hence, RemoveMatches did not mark u.

So u must have been marked in a call to MTF, which occurs in the following cases.

Case 1: MTF(u) at line 58 if $u.type =$ `search` at line 58, or

Case 2: by MTF(u) at line 58 if $u.remove =$ `true`, or

Case 3: by a call MTF(v), where $v \neq u$ at line 56.

First we show that **Case 1** cannot occur by proving that $u.type \neq$ `search` at line 58 of MTF(u). Since $u \in U = \{\, u \mid u.key = k,\ z(x).serial \leq u.serial < y.serial \,\}$ and $y.serial < x.serial$, we may apply Claim 45.3 and Claim 45.1 to u. In MTF(u), line 50 calls IncreaseEnd(u, 0). We consider two subcases.

Subcase 1: Assume $u.end$ is updated in IncreaseEnd(u, 0). Then the CAS at line 78 successfully updates $u.end$, and by Claim 45.3 we have $u.end \geq z(u).serial$ thereafter.

Subcase 2: Assume $u.end$ is not updated in IncreaseEnd(u, 0). Then either $u.end$ was already non-negative at line 78, or a concurrent operation updated $u.end$ between line 77 and the CAS at line 78. So $u.end$ was already updated before

line 78 by either MTF(u) or a call to RemoveMatches. By Claim 45.3 and Claim 45.1, $u.end \geq z(u).serial$ holds thereafter.

Thus, in both subcases, $u.end \geq z(u).serial$ holds after the call to IncreaseEnd(u, 0) at line 50. Since $z(u).serial > 0$ (by Corollary 22 **(i)**), it follows that $u.end > 0$ at line 51 of MTF(u). Thus $u.type$ is set to `insert` at line 52, so $u.type \neq$ `search` at line 58.

Now consider **Case 2** and **Case 3**. In either case, some call MTF(v) with $v \neq u$ must have reached u during its traversal (lines 37–49) in order to mark u (Case 2) or to set $u.remove \leftarrow$ `true` (Case 3). Moreover, by lines 41 and 44, u must be the *first* node with key k encountered by MTF(v). Hence, by Property 2 **(ii)** of Lemma 30, we have $v.serial > u.serial$.

By Lemma 21, no node with key k was claimed between u and y. Since y is unmarked at s, a call MTF(v) with $v.serial > y.serial$ is impossible: by Lemma 35 and Property 2 **(ii)** of Lemma 30, MTF(v) would reach y before u. This contradicts the fact that u is the *first* node with the key k encountered by MTF(v). Hence, $v = y$.

Marking u (line 56) or setting $u.remove$ to `true` (line 55) occurs only after line 50 in MTF(y), so MTF(y) must have already executed IncreaseEnd($y, *$) at line 50 before s. If the CAS at line 78 of this call to IncreaseEnd(y, $*$) succeeds, then by Claim 45.3 we have $y.end \geq z(y).serial$ thereafter. Otherwise, $y.end$ must have been set earlier by MTF(y) or by a concurrent RemoveMatches, and in either case, by Claim 45.3 and Claim 45.1, $y.end \geq z(y).serial$ still holds after s.

Proof of Property 4.

Consider a traversal by MTF(x) at lines 37–49 that reaches a node z with $z.serial \leq z(x).serial$. The traversal begins at the node x (see line 37), and each node encountered in the traversal is read at line 38 or line 47. MTF(x) advances through nodes in strictly decreasing order of serial numbers (by Property 2 **(ii)** of Lemma 30). If $last =$ `NIL`, no node with key k was encountered on the path from x to z (see lines 41 and 45), so all nodes with key k and serial numbers in ($z.serial$, $y'.serial$) are marked by Property 2 **(iii)** of Lemma 30. If $last \neq$ `NIL`, then $last$ was the last node with key k encountered on the path from x to z (again by lines 41 and 45), so again, all nodes with serial numbers in ($z.serial$, $y'.serial$) are marked by Property 2 **(iii)** of Lemma 30.

Proof of Property 5.

The step s could violate **Property 5** if it reaches line 40 after a node with a serial number

less than or equal to $z(x).serial$ is read at line 38 or 47. If that node is not the *first* such node encountered by MTF(x), then **Property 5** continues to hold after s by **Property 5** of (IH) and Lemma 41.

Let s be a step that reaches line 40 after MTF(x) encounters a node z, where z is the first node visited with $z.serial \leq z(x).serial$. By Claim 45.2 either $y.end \geq z(y).serial$ holds when z is visited, which by Lemma 41 implies $y.end \geq z(y).serial$ after s, or MTF(y) made a call IncreaseEnd(y, $*$) before s, which by Claim 45.3 implies that $y.end \geq z(y).serial$ after s. □

Finally, using Invariant 42, Invariant 43, and Lemma 45, we prove the condition for the correctness of MTF outlined in Equation (4.1).

Lemma 46. *Let x be a node claimed by an* Insert(k) *or* Search(k). *At line 51 of* MTF(x), *the following are equivalent (note if $U_x = \varnothing$ the right-hand side is false):*

$$
\begin{aligned}
&x.end > 0 \text{ at line 51 of } \mathrm{MTF}(x) \\
&\quad \iff \Big(\exists\, u \in U_x \text{ with } op(u) = \textsc{Insert}(k), \\
&\qquad \text{and } \forall\, u' \in U_x,\ u.serial < u'.serial < x.serial \Rightarrow op(u') \neq \textsc{Delete}(k)\Big).
\end{aligned}
$$

Proof. ($\Rightarrow$) Assume $x.end > 0$ at line 51. By Invariant 42, there exists an insert node z with $z.serial < x.serial$ and no delete node with key k and a serial number in $(z.serial, x.serial)$. By Invariant 43, this implies an Insert(k) operation $op(w)$ claimed a node w with $w.serial \leq z.serial$ such that no node with the key k and a serial number in $[w.serial, z.serial]$ is a delete node. Thus, we have $op(w) \in U_x$, and for all $u' \in U_x$ with $u.serial < u'.serial < x.serial$ we have $u'.type \neq$ `delete`, which implies op(u') $\neq$ Delete(k) by lines 22–23 and Observation 17. Thus the right-hand side holds.

($\Leftarrow$) Assume there exists a $u \in U_x$ with op(u) = Insert(k) and no Delete(k) claimed a node with a serial number in $(u.serial, x.serial)$. Let $z(x)$ be the node defined by Definition 44. By **Property 1** of Lemma 45, $z(x) \neq$ `NIL`. We consider two cases.

Case 1: If MTF(x) reaches a node with a serial number less than or equal to $z(x).serial$, then when the first such node is reached at line 38 or line 47, we have $x.end \geq z(x).serial$ at the next iteration of line 40 by **Property 5** of Lemma 45.

Case 2: If MTF(x) does not reach any node with a serial number less than or equal to

$z(x).serial$, then $x.end \geq z(x).serial$ holds at some iteration of line 40. This is because there is no delete node with key k in $(z(x).serial, x.serial)$ and MTF(x) visits nodes in strictly decreasing serial order (Property 2 **(ii)** of Lemma 30). So the loop at lines 40–49 can terminate before reaching a node with a serial number less than or equal to $z(x).serial$ only if $x.end \geq z(x).serial$ holds at some iteration of line 40 (by the second exit condition at line 40).

Because $z(x).serial > 0$ (by Corollary 22 **(i)**), and $x.end$ only increases in value over time (by Lemma 41), we have $x.end \geq z(x).serial > 0$ at line 51.

□

Corollary 47 (Correctness of MTF(x))**.** MTF(x) *returns* `true` *if and only if key* $k = x.key$ *was in the set when* x *was claimed; otherwise it returns* `false`*.*

Proof. By Lemma 46 and the choice of linearization points (Definition 13) in concert with Lemma 21, the right-hand side of the equivalence is precisely the statement that, at the time x is claimed, there is an INSERT(k) not followed by any DELETE(k). Hence $x.end > 0$ at line 51 if and only if k was in the set when x was claimed. Since MTF(x) returns `true` exactly when $x.end > 0$ at that line, the claim follows. □

4.8 Correctness of Search, Insert, and Delete

We prove in the following two Theorems that our list is a linearizable implementation of a set.

Theorem 48 (Correctness of SEARCH and INSERT)**.** *Let* x *be a node claimed on key* k *by a call to either* SEARCH(k) *or* INSERT(k) *at line 29 of* PUBLISH(d)*, where* d *is the data object created at line 16 or 19, respectively. Then the following hold:*

(i) SEARCH(k) *returns* `true` *if and only if* k *was in the set when it claimed its node* x*.*

(ii) INSERT(k) *returns* `true` *if and only if* k *was not the set when it claimed its node* x*.*

Proof. Both SEARCH(k) and INSERT(k) invoke MTF(x) (at line 18 and 21, respectively) after claiming x. By Corollary 47, MTF(x) returns `true` if and only if k was in the set at the time x was claimed. In SEARCH(k), the return value of MTF(x) is returned directly (line 18). Hence SEARCH(k) returns `true` exactly when k is in the set at x's claim and

`false` otherwise. In INSERT(k), the return value is the logical negation of MTF(x) (line 21). Hence SEARCH(k) returns `true` exactly when k is absent from the set at x's claim and `false` otherwise. Therefore, both SEARCH and INSERT are correct with respect to the set abstraction (see Definition 2). □

Theorem 49 (Correctness of DELETE(k))**.** *Let x be the node claimed by a call to* DELETE(k) *at line 29 of* PUBLISH(d) *(called at line 23), where d is the operation's data object created at line 22. Then* DELETE(k) *returns* `true` *if and only if k was in the set when x was claimed, and* `false` *otherwise.*

Proof. After claiming x, DELETE(k) calls REMOVEMATCHES(x) at line 24.

(⇒) Only-if. Assume DELETE(k) returns `true`. Then the traversal of REMOVEMATCHES(x) (lines 64–67) collected a matching insert into *matches* and set $oldestInsert \geq 0$ (lines 69–70), passing the test at lines 73–76. Hence, a matching insert node z for k was reached from x by the traversal.

To derive a contradiction, suppose that a DELETE(k) claimed a node between the claim of z and the claim of x. By Lemma 21, such a node must have a serial number in $(z.serial, x.serial)$. Let y be the oldest node claimed by a DELETE(k) with a serial number in $(z.serial, x.serial)$. By Property 2 **(ii)** of Lemma 30, the traversal of REMOVEMATCHES(x) must follow a link from some node y' to a node y'' where $y'.serial \geq y.serial > y''.serial$ on the path from x to z. If $y' = y$, then the traversal would stop at y by second exit condition at line 64 and the fact that y is a delete node with the key k, and therefore would never reach z.

If $y'.serial > y.serial$, then y is skipped by the traversal when the link $y'.next$ is followed to y''. By Property 2 **(iii)** of Lemma 30, skipping y implies it is marked. Since y is marked, and y is the oldest delete node for key k with a serial number in $(z.serial, x.serial)$, Invariant 39 implies z is unreachable from y'' when y is skipped. Hence, by the contrapositive of Lemma 34, a traversal that begins at y'' cannot reach z. This contradicts the assumption that REMOVEMATCHES(x) reaches z.

By Invariant 43, the existence of z implies the existence of a node z' with $z'.serial < z.serial$ claimed by an INSERT(k) operation such that no delete nodes with serial numbers in $(z'.serial, z.serial)$ were claimed by a DELETE(k) operation. Lemma 21 and Definition 13 imply that the latest update operation on the key k linearized before the claim of x is an INSERT(k). Thus, if the current DELETE(k) returns `true`, then k was in the set when x was

claimed.

(⇐) If. Assume k was in the set when x was claimed. By Lemma 21 and Definition 13, there exists an INSERT(k) that claimed a node z before x such that no node with a serial number in ($z.serial$, $x.serial$) was claimed by DELETE(k). Let y be the newest matching node of x.

Claim 49.1. *The newest matching node y of x can be marked only after* REMOVEMATCHES(x) *completes its traversal at lines 60–68.*

Proof. Since z exists and no delete node for key k has a serial number in ($z.serial$, $x.serial$), the newest matching node of x is not a delete node. Moreover, REMOVEMATCHES(x) only marks nodes at lines 74 and 75, both of which occur *after* the traversal phase at lines 60–73. Thus, REMOVEMATCHES(x) could not have caused y to be marked before its traversal completes.

Furthermore, a call REMOVEMATCHES(x') with $x' \neq x$ could not have resulted in y being marked during REMOVEMATCHES(x)'s traversal. REMOVEMATCHES(x') marks only nodes with key k that it reaches during its own traversal from x' (lines 60–73). Because REMOVEMATCHES(x') is invoked by the DELETE(k) that claimed x' (by Observation 17 and lines 23–24), and there is no delete node with key k in ($z.serial$, $x.serial$), for REMOVEMATCHES(x') to reach and mark y (which satisfies $z.serial \leq y.serial < x.serial$), we must have $x'.serial > x.serial$ (by Property 2 **(ii)** of Lemma 30). By Lemma 38, only the operation that claimed x can mark x via a call to MARKANDREMOVE($*, x$) at line 75 of REMOVEMATCHES(x). So by Lemma 35 and Property 2 **(ii)** of Lemma 30, the traversal by REMOVEMATCHES(x') must reach x before reaching y. Upon reaching x, REMOVEMATCHES(x') would be unable to proceeed to y to collect y into its *matches* set, by the second exit condition at line 71. Hence, no call to REMOVEMATCHES(x') with $x' \neq x$ can mark y at line 74 before REMOVEMATCHES(x) completes its traversal.

It remains to show that no call to MTF can mark y before REMOVEMATCHES(x)'s traversal completes, which we show by cases.

Case 1. No call MTF(y') with $y' \neq y$ can mark y or set $y.remove$ to `true` before the traversal of REMOVEMATCHES(x') finishes.

Assume, for contradiction, that some call MTF(y') with $y' \neq y$ marks y at line 56 or sets $y.remove$ to `true` at line 55 before the traversal of REMOVEMATCHES(x)

completes. Such an update is possible only if MTF(y') already completed its traversal from y' (lines 37–49) and y was the *first* node with key k it reached (by lines 41 and 44).

Because y is the newest matching node of x, no node with key k has a serial number in ($y.serial$, $x.serial$). Thus, for MTF(y') to reach y, we must have $y'.serial > x.serial$ (by Property 2 **(ii)** of Lemma 30). Since traversals visit nodes in decreasing serial order (by Property 2 **(ii)** of Lemma 30), and x can only be marked after RemoveMatches(x') completes its traversal (by Lemma 38), MTF(y') must have reached x before reaching y (by Lemma 35). By the second exit conditon at line 40, MTF(y')'s traversal would terminate at the next execution of line 40 after reading x, before reaching y. This contradicts the assumption that y is the first node with key k reached by MTF(y') during its traversal. Thus no call MTF(y') with $y' \neq y$ can mark y or set $y.remove$ to `true` before the traversal of RemoveMatches(x) completes.

Case 2. The call MTF(y) cannot mark y before RemoveMatches(x)'s traversal completes.

The call MTF(y) can mark y at line 58 only if $y.type =$ `search` or $y.remove =$ `true` holds. By lines 52 and 51, $y.type =$ `search` at line 58 would require $y.end \leq 0$ at line 51. However, since z was claimed by an Insert(k) and no `delete` with key k exists in ($z.serial$, $y.serial$), Lemma 46 implies that $y.end > 0$ holds at line 51 of MTF(y). Thus $y.type \neq$ `search` at line 58.

The remaining way for MTF(y) to mark y at line 58 is to have $y.remove =$ `true`. But the only way to set $y.remove =$ `true` is for some traversal to *reach* y and set $y.remove$ to `true` at line 55 before line 58. By **Case 1**, no MTF(y') with $y' \neq y$ can set $y.remove$ to `true` before RemoveMatches(x)'s traversal completes. Hence $y.remove \neq$ `true` when MTF(y) reaches line 58. Therefore, MTF(y) cannot mark y at line 58 before RemoveMatches(x)'s traversal completes.

□

Let s be the step immediately before the traversal of RemoveMatches(x) (lines 60–73) completes. During this traversal, every node with key k and a serial number in the range $[z.serial, x.serial)$ that is unmarked in C_s is added as the second node of a pair to *matches* at line 65. The traversal visits nodes in decreasing order of serial number (by Property 2 **(ii)**

of Lemma 30) and stops only if it reaches the end of the list or a delete node with key k (line 88). Because no delete node with key k has a serial number in $[z.serial, x.serial)$, the traversal must reach every unmarked node in C_s with key k and a serial number in $[z.serial, x.serial)$ (by Lemma 35) and add then as the second node of a pair to *matches*.

By Claim 49.1, y is unmarked in C_s, and we have $y.serial \in [z.serial, x.serial)$. Therefore, RemoveMatches(x) must reach y and add it as the second node of a pair to *matches* at line 65.

Let u be the oldest unmarked node with key k that appears as the second node of a pair in *matches* in C_s. Such a node u exists because y itself is unmarked and appears as the second node of a pair in *matches* in C_s.

Since z was claimed by an Insert(k) operation, Lemma 21 ensures that $z.type =$ `insert` when u was claimed. Moreover, no delete node with key k has a serial number in the range $[z.serial, u.serial)$, so $z(u) \neq$ `NIL`. Hence, Lemma 45 applies to u.

Because every unmarked node in C_s with key k and a serial number in $[z.serial, x.serial)$ appears as the second node of a pair in *matches*, and u is the oldest such node, all nodes with key k and a serial number in $[z.serial, u.serial)$ are marked in C_s. Hence, by **Property 3** of Lemma 45, we have $u.type =$ `insert` in C_s.

So when *matches* is scanned in reverse order (lines 69–70) after s, the variable *oldestInsert* is set at line 70 to a positive serial number corresponding to at least the position of u's pair in *matches*. Therefore, the check at line 73 evaluates to $(oldestInsert \geq 0) =$ `true`, and by line 24, Delete(k) returns `true`.

□

Chapter 5

Amortized Step Complexity

We assume each *read*, *write*, and CAS operation takes $O(1)$ time. The concurrent working set is defined in Section 5.1, and the worst-case amortized costs of the routines of our algorithm are established in Section 5.2–Section 5.9. Our analysis begins by bounding the step complexity of helper routines and progressively builds up to proving the final bounds for SEARCH, INSERT, and DELETE in Theorem 71.

5.1 The Concurrent Working Set

Definition 50. *Let $S(\ell)$ denote the abstract set immediately before step ℓ.*

Because operations overlap in the concurrent setting, we define the working set using linearization points to capture the notion of keys accessed since k's last access.

Definition 51. *Fix an operation op on the key k with linearization point ℓ such that $k \in S(\ell)$.*

*Let ℓ^k be the most recent linearization point among operations on the key k that completed before ℓ. The **working set** $w(op)$ is the set of distinct keys in $S(\ell)$ that were accessed by an operation whose linearization point ℓ' satisfies $\ell^k \preceq \ell' \prec \ell$.*

We show in Section 5.2–Theorem 71 (with final bounds in Theorem 71) that an operation op on key k, linearized at step ℓ with $k \in S(\ell)$, completes in worst-case amortized $O(|w(op)| + \dot{c}(op))$ steps. Because we measure the working set of op starting from the most recent linearization point among operations on k that have *completed* before ℓ, our definition of concurrent working set is looser than a definition that applies the sequential definition of working set directly to the linearization order.

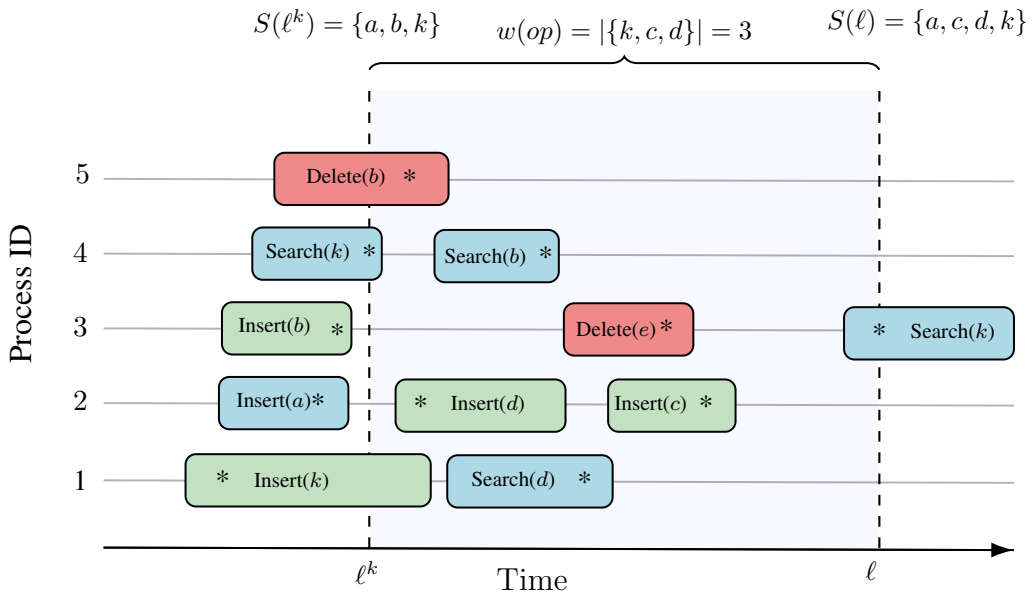

Figure 5.1: Concurrent working set size for a Search(k) in an execution in a system with five processes. Each box represents an operation's duration, with a $*$ marking its linearization point. DELETE, SEARCH, and INSERT operations are shown in red, blue, and green, respectively. For the SEARCH(k) operation *op* linearized at ℓ, the working set size $w(op)$ is the number of distinct keys in the set $S(\ell)$ with operations on their key linearized after the most recent linearization point among completed accesses to k (at ℓ^k) and before ℓ. The brace between ℓ^k and ℓ indicates this working-set window for *op*, with $w(op) = |\{k, c, d\}| = 3$.

To see why our definition is weaker, consider that the most recent *completed* access to k before *op* may have a much earlier linearization point ℓ^k than another access to k that is still pending at ℓ (see Figure 5.2).

We use our looser working set definition because a successful SEARCH(k) linearizes when it claims its node but does not promote that node until after determining its result. This means an operation *op* on key k may encounter search nodes from pending operations on key k during its traversal before reaching a matching insert node, even if those pending operations linearized before *op*. As a result, the runtime of operations on our list cannot be captured by a definition of concurrent working set that applies the sequential working set definition directly to the linearization order. Whether a stronger working set bound is achievable for concurrent lists remains an open question.

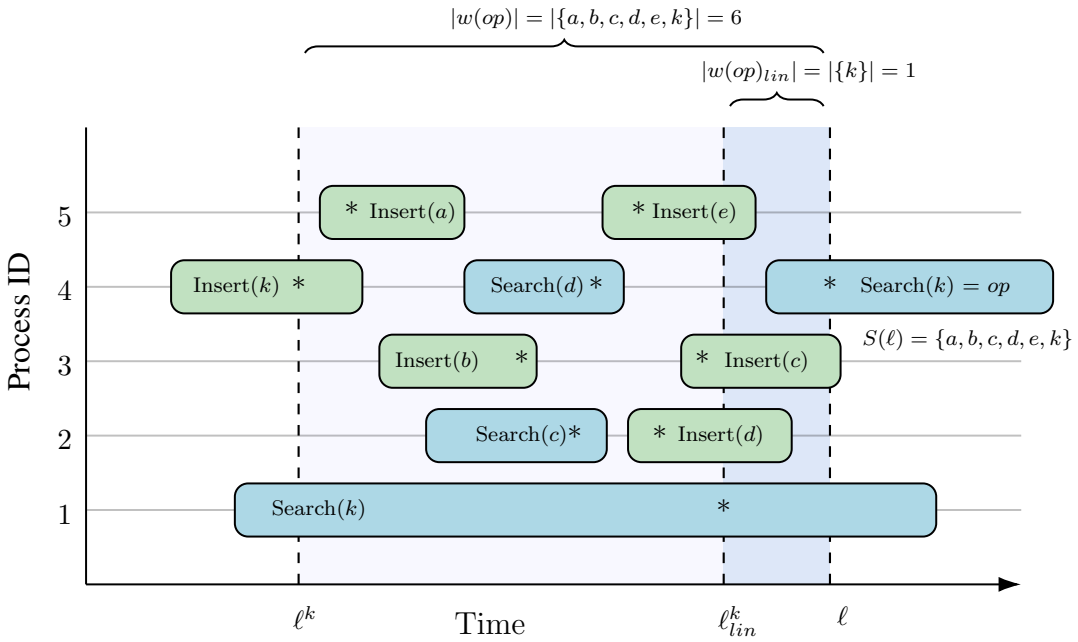

Figure 5.2: A comparison of concurrent working set definitions. The latest successful SEARCH(k) operation, *op*, linearizes at step ℓ. We use ℓ^k to denote the most recent linearization point among accesses to k that *completed* before ℓ, and ℓ^k_{lin} to denote the most recent linearization point among *all* accesses to k before ℓ. Under our concurrent working set definition (Definition 51), the working set of *op* is $w(op) = \{a, b, c, d, e, k\}$, so $|w(op)| = 6$. If instead we applied the sequential working set definition directly to the linearization order, the working set $w(op)_{lin}$ would contain only k, giving $|w(op)_{lin}| = 1$. INSERT operations are shown in green, SEARCH operations in blue, and the two working-set windows in light and dark blue, respectively.

5.2 Analysis of SetBacklink

Let *op* be an operation executing SETBACKLINK(x, y). Line 105 is a single step. The loop at lines 106–107 terminates once $y.backlink$ is set to a node whose serial number is at least $x.serial$. In each iteration, if $y.backlink$ has not yet been set to such a node, *op* attempts to set $y.backlink$ to x using a CAS at line 107. A failed CAS by *op* implies that another concurrent operation op' updated $y.backlink$ at line 107 between the last read of $y.backlink$ by *op* (at line 105 or the previous execution of line 108) and the current CAS attempt.

We charge each successful CAS by an operation op' at line 107 an additional $O(\dot{c}(op'))$ steps (as defined in Section 1.1.2) to cover the constant number of steps spent in the iterations of the loop by concurrent processes whose CAS attempts at line 107 it causes to fail. Consequently, all iterations of the loop at lines 106–107 are free except for an iteration at which *op*'s

CAS succeeds, which costs $O(\dot{c}(op))$ steps. Therefore, the amortized cost of SetBacklink by an operation op is $O(\dot{c}(op))$.

5.3 Analysis of FindPred

FindPred is called only on line 79 of MarkAndRemove. Let C be the configuration in which a call MarkAndRemove(x, y) is made by an operation op.

We first note that $x = Head$ in C only for calls to MarkAndRemove made at line 58 or line 75. By contrast, when MarkAndRemove(x, y) is invoked with a node x obtained by a forward traversal (at line 56 in MTF or line 74 in RemoveMatches), that traversal begins from the operation's own claimed node x'. Concretely, the call to Publish made by the operation at line 17, line 20, and line 23, returns the node x' the operation claims (by Observation 17); this very node x' is then passed as the argument to MTF (see lines 17–18) and lines 20–21), or to RemoveMatches (see lines 23–24). MTF begins its traversal at x' by lines 37– 38 and advances in lines 40–49, while RemoveMatches begins its traversal at x' by lines 60–61 and advances in lines 64–67. The predecessor x actually passed to MarkAndRemove is a node read by these traversals (see line 44 and line 56 for MTF; line 65 and line 74 for RemoveMatches). Since, by Property 2 **(ii)**, every node reached during such a forward walk has a strictly smaller serial number than x', whereas $Head$ is always the sentinel with the largest serial (by Corollary 22 **(ii)**), the traversal-derived predecessor x cannot satisfy $x = Head$ in C.

Now we analyze the running time of FindPred$(x, y.serial)$, invoked by op at line 79 by cases depending on whether $x = Head$ in C.

Case 1: $x = Head$ in C.

Here y is the node claimed by op (by Observation 17 and lines 17, 20, and 23), and MarkAndRemove$(Head, y)$ is invoked at line 58 or line 75. Between the moment y was claimed and the moment $Head$ was read at line 58 or line 75, the $Head$ sentinel may have changed. By Corollary 22 **(ii)** and Corollary 23, any active node z with $x.serial \geq z.serial > y.serial$ must have been appended concurrently with op. Thus, the number of such nodes is bounded by the interval contention of op, so there are $O(\bar{c}(op))$ (as defined in Section 1.1.2) such nodes. By Property 2 **(ii)**, the loop at lines 100–102 in FindPred visits only nodes in this serial range (save the final check that may observe $\leq y.serial$). Thus the traversal performs $O(\bar{c}(op))$ steps in this case.

Case 2: $x \neq Head$ in C.

By Lemma 25 **(ii)**, there is an earlier moment at which x is active and $x.next = y$. By Lemma 31, after that moment $x.next$ can only change to a node with strictly smaller serial number than $y.serial$. Therefore, at the time of the call, the current successor of x has serial number less than or equal to $y.serial$, so the first check at 100 fails immediately (from 99), and FindPred returns in $O(1)$ steps.

In the worst case then, a call to FindPred$(x, y.serial)$ by operation op runs in $O(\bar{c}(op))$ time.

5.4 Analysis of MarkAndRemove

Let us analyze a call to MarkAndRemove(x, y) made by an operation op.

5.4.1 Analysis of lines 79–86

By Section 5.3, line 79 takes $O(\bar{c}(op))$ steps in the worst case. Line 80 takes $O(1)$ steps. Line 81 takes an amortized $O(\dot{c}(op))$ steps by Section 5.2, and line 82 takes $O(1)$ steps. So lines 79–82 take amortized $O(\dot{c}(op))$ steps.

The loop at lines 83–86 terminates once y is detected as marked (see line 86). Any failed CAS at line 84 indicates that another concurrent CAS at the same line succeeded between the last update to *succ* (at line 82 or 85) and when the current CAS was executed. We charge each successful CAS of operation op' $O(\dot{c}(op'))$ steps to account for the $O(1)$ steps done in each iteration where one of the $O(\dot{c}(op'))$ failed CAS attempts occurs. This way, each iteration made by op that fails its CAS attempt becomes free, and the loop takes an amortized $O(\dot{c}(op))$ steps.

We also charge the operation op' that successfully marked y at line 84 an additional $O(\dot{c}(op'))$ steps to account for up to $O(\dot{c}(op'))$ failed CAS attempts at line 95 by operations that read y as unmarked at line 89 before op' marked it, but executed their CAS afterward (see Section 5.4.2). Combined with the $O(\dot{c}(op))$ steps preceding the loop (lines 79–82), lines 79–86 take an amortized $O(\bar{c}(op))$ steps.

5.4.2 Analysis of the loop at lines 88–96

Analysis of the nested loop at lines 89–90.

In this section, we calculate the cost of traversing through nodes in the backwards direction by following backlinks in the loop at lines 89–90. To achieve an amortized bound of $O(\bar{c}(op))$ for these steps over all iterations of the loop at lines 88–96, we argue that a marked node is only traversed backwards by an operation that is concurrent with the operation that marked it. As a first step towards this aim, we show that MarkAndRemove ensures the node passed as the second argument is skipped by a backlink and a corresponding link (see Figure 5.3).

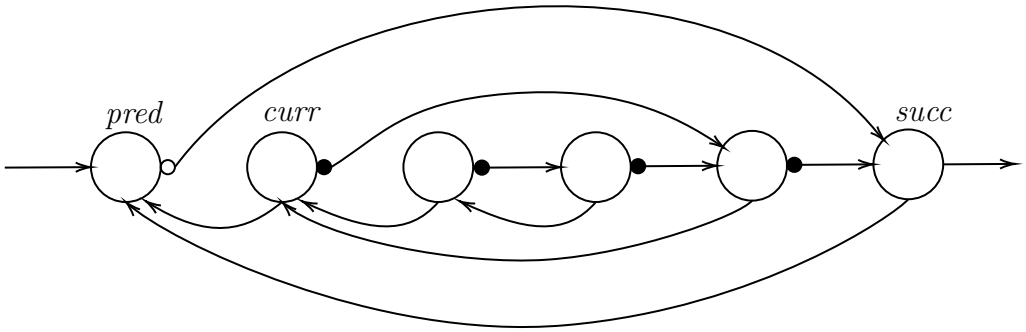

Figure 5.3: A chain of marked nodes removed by MarkAndRemove. A possible state of the MTF list immediately after *pred.next* is updated from *curr* to *succ* by MarkAndRemove. Note that links and backlinks are non-overlapping, and all nodes skipped by the link between *pred* and *succ* are marked.

Lemma 52. *At some moment during a completed call to* MarkAndRemove(x, y)*, we have* $y'.backlink = x'$*, where* $x'.serial > y.serial > y'.serial$*, and where, in an earlier configuration,* x' *is active and* $x'.next = y'$*.*

Proof. At line 97, MarkAndRemove calls SetBacklink$(pred, pred.next)$. By Property 1 **(iii)** of Lemma 30, when the condition at line 92 was last evaluated as false, we had $pred.serial > y.serial > succ.serial$. Moreover, by the exit condition on line 96, we know that at line 97 we have $succ.serial > pred.next.serial$. Combining these inequalities gives $pred.serial > y.serial > succ.serial > pred.next.serial$ at the moment SetBacklink$(pred, pred.next)$ is invoked.

By the check at line 106, SetBacklink$(pred, pred.next)$ sets $pred.next.backlink$ to a node

whose serial number is at least *pred.serial* at some point during its execution. Let x' denote this node and let $y' = pred.next$, so that at some moment during the execution of SetBacklink$(pred, pred.next)$ we have $y'.backlink = x'$ and $x'.serial > y.serial > y'.serial$. Finally, by Lemma 27, node x' is active and satisfies $x'.next = y'$ at an earlier point in the execution.

□

We now show that once a node's backlink points to a node x, it can only be changed to a node with a higher serial number than x.

Observation 53. *If $y.backlink = x$ for some node x, $y.backlink$ can only be updated to a node x' such that $x'.serial > x.serial$.*

Proof. The backlink of a node y is only set at line 107 by a CAS, which updates $y.backlink$ from the node stored in the variable *backlink* (read at line 105 or line 108) to the node in *pred*. Before each CAS attempt at line 107, we have $pred.serial > backlink.serial$ by the check at line 106. Thus, when the CAS succeeds, $y.backlink$ is updated to a node with a strictly larger serial number than before. □

We obtain the following corollary from Lemma 52 and Observation 53.

Corollary 54. *When* MarkAndRemove(x, y) *completes, $y'.backlink = x'$ always holds for some nodes x' and y', where x' is a node with $x'.serial > y.serial > y'.serial$.*

Combining Lemma 27 and Property 2 **(ii)**, we derive the following.

Corollary 55. *Let $y_0, y_1, \ldots, y_i$ be a backlinked chain of nodes such that $y_j.backlink = y_{j+1}$ $(0 \leq j < 0)$. We have $y_0.serial > y_1.serial > \ldots > y_i.serial$.*

Now we show that the number of marked nodes the operation *op* reaches in the backwards direction is $O(\bar{c}(op))$.

Lemma 56. *An operation op cannot reach a marked node y by following links or backlinks if the call to* MarkAndRemove$(*, y)$ *that marked y completed before op began.*

Proof. Let C_{op} be the configuration after *op* claims its operation node x_{op}. C_{op} is after a call to MarkAndRemove$(*, y)$ that marked y completed. By Corollary 54, in C_{op}, $y'.backlink = x$

holds for some nodes x and y' where x is some node with $x.serial > y.serial > y'.serial$ and by Lemma 27, x is active and $x.next = y'$ held in some configuration $C' \preceq C_{op}$. COm

We now show that no node visited by *op* has a serial number in the range $(y'.serial, x.serial)$.

Firstly, by Lemma 21, x_{op} is the active node with the highest serial number at C_{op}, so $x_{op}.serial > x.serial$ because x is active before C_{op}. The active nodes with serial numbers in $(y'.serial, x.serial)$ are not reachable from x in C', and $x_{op}.serial > x.serial$, so by Lemma 32, no active node with a serial number in $(y'.serial, x.serial)$ is reachable from x_{op} in C'. By Lemma 34, this implies that no node in the range $(y'.serial, x.serial)$ can be reached by *op*'s traversal from x_{op} in the forward direction, including y.

By Corollary 55, *op* can only reach a node w' with a serial number in $(y'.serial, x.serial)$ by following backlinks if it follows a backlinked chain from a node with a smaller serial number than $y'.serial$. Let w be the *first* node with $w.backlink = w'$ in such a chain. This means we have $x.serial > w'.serial > y'.serial > w.serial$. Now by Property 2 **(iv)** of Lemma 30, we cannot ever have forward links with $x.serial.next = y'$ and $w'.serial = w$. By Lemma 27, every backlink corresponds to a forward link. Therefore, we cannot have ever have $w.backlink = w'$ and $y'.backlink = x$. Hence, it is not possible for *op* to reach a node w' by following backlinks. Because the node w' is an arbitrary active node with a serial number in the range $(y'.serial, x.serial)$, take $w' = y$, and the lemma holds. □

As established above, an operation op_1 cannot reach a node y that was marked by some op_2 that completed before op_1 by following backlinks. Consequently, after y is marked, $O(\bar{c})(op_2)$ operations concurrent with op_2 can reach y via a backlink. If we charge each such operation op_2 an additional $O(\bar{c})(op_2)$ steps; these steps pay for the next iteration of the loop at lines 89–90 performed by the operation op_1 that read the node as marked at line 89. Hence an operation pays only an amortized $O(1)$ steps to execute lines 89–90 to evaluate the condition at line 89 as false a single time, giving *op* an amortized $O(1)$ cost for the loop at lines 89–90 in each iteration of the loop at lines 88–96.

Analysis of the nested loop at lines 92–93.

The nested loop at lines 92–93 traverses through a chain of marked nodes. We show that each node read in this loop must have been marked by an operation concurrent with the traversing operation.

By Lemma 56, if an operation op' marks a node y, only operations concurrent with op' can

reach y in its marked state. Consequently, $O(\bar{c}(op'))$ operations can traverse a node marked by op'. Each iteration of the loop at lines 92–93 can therefore be charged to the operation op' that marked the node checked at line 92. By assigning $O(\bar{c}(op'))$ steps to the marking operation op' to account for iterations of the loop by other operations, those operations can perform lines 92–93 for free. Therefore the loop at lines 92–93 take an amortized $O(1)$ steps to pay for the test at line 92 that causes the loop to exit.

Total amortized cost of the loop at lines 88–96.

We have shown that the nested loops at lines 89–90 and 92–93 each have an amortized cost of $O(1)$ per iteration of the loop at lines 88–96, and the remaining steps (lines 91 and 95–96) also take $O(1)$ time per iteration.

In each iteration of the loop at lines 95–96 by op, the CAS at line 95 succeeds or fails. Lemma 31 implies that if the CAS at line 95 succeeds, $pred.next.serial \leq succ.serial$ holds at line 96. Therefore, once op succeeds in performing the CAS at line 95, it exits the loop at the next evaluation of the exit condition at line 96. Only a failed CAS attempt can cause another iteration of the loop at lines 88–96.

Each successful CAS on the link of a node x' at line 95 by an operation op' may cause up to $O(\dot{c}(op'))$ other operations active at that moment to fail their own CAS attempts. These are operations that have read $x'.next$ at line 91 but have not yet executed their own CAS at line 95. We charge op' $O(\dot{c}(op'))$ steps to account for the $O(\dot{c}(op'))$ iterations of the loop at lines 88–96 performed by these operations in which their CAS attempt at line 95 failed because of op'.

Similarly, if an operation op' marks a node $pred$ after another operation observes it as unmarked at line 80 but before that operation performs its CAS at line 95, that CAS will fail. At most $\dot{c}(op')$ such failures can occur, so we charge op' an additional $O(\dot{c}(op'))$ steps to cover for the iteration of the loop by operations whose CAS attempts were invalidated by op' (see Section 5.4.1).

Hence, each iteration of the loop at lines 88–96 by op in which the CAS at line 95 fails is free for op, while an iteration in which the CAS succeeds costs $O(\dot{c}(op))$ steps. There can be at most one iteration in which the CAS succeeds, so total amortized cost of the loop for op is $O(\dot{c}(op))$.

5.4.3 Final Bound

We have shown (in Section 5.4.1) that lines 79–83 take an amortized $O(\bar{c}(op))$ steps, and (in Section 5.4.2) that the loop at lines 88–96 takes an amortized $O(\dot{c}(op))$ steps. Line 97 takes an amortized $O(\dot{c}(op))$ steps by Section 5.2. In total then, the amortized cost of MarkAndRemove is $O(\dot{c}(op))$.

5.5 Analysis of IncreaseEnd

IncreaseEnd performs a read at line 77, and a conditional check and CAS attempt at line 78, both of which take $O(1)$ steps. So IncreaseEnd takes $O(1)$ steps.

5.6 Analysis of MTF

In this section, we analyze the running time of MTF at line 18 or line 21. We first show that the sequence of nodes visited by a traversal from a node x is a subsequence of the nodes that were reachable from x at any moment before the traversal began.

Definition 57. *For sequences of nodes A, B, we write $A \sqsubseteq B$ if A is a subsequence of B.*

Definition 58. *We use the notation $T(x, C, q) = (t_0, t_1, \ldots, t_i)$ to denote any forward traversal that begins at x in configuration C and whose final node is $q = t_i$.*

Lemma 59. *Let x be an active node in a configuration C and let $T(x, C', q)$ be the sequence of nodes encountered by a traversal T that begins at x in configuration $C' \succeq C$ and ends at q. Then $T(x, C', q) \sqsubseteq R(x, C)$.*

Proof. By Lemma 34, we have $t_i \in R(x, C)$ for every $t_i \in T(x, C', q)$. Both $T(x, C', q)$ and $R(x, C)$ enumerate nodes in strictly decreasing serial order (by Property 2 **(ii)** of Lemma 30). So, $T(x, C', q) \sqsubseteq R(x, C)$. □

Since both $T(x, C', y)$ and $R(x, C)$ list nodes in strictly decreasing serial order (by Property 2 **(ii)** of Lemma 30), $T(x, C', q)$ cannot include any node that appears after q in $R(x, C)$. Thus every node in $T(x, C', q)$ is a member of the *path* from x to q in C.

Definition 60. *Let $R(x, C) = (r_0, r_1, \ldots, r_i)$ be the nodes reachable from x in configuration C. For any $q = r_j \in R(x, C)$, the **path** from x to q is the prefix $P(x, C, q) = (r_0, r_1, \ldots, r_j = q)$.*

Corollary 61. *If x is active in C and $C' \succeq C$, then $T(x, C', q) \sqsubseteq P(x, C, q)$.*

Now let us consider a call to MTF(x) by an operation *op* on the key k that claimed the node x at step ℓ.

MTF(x) performs only $O(1)$ steps at lines 36–37. Lines 43 and 49 take $O(1)$ steps by Section 5.5). Lines 50–59 involve a constant number of steps, and perhaps some calls to MarkAndRemove, which by Section 5.4 take an amortized $O(\dot{c}(op))$ steps. Only the loop at lines 40–49 remains to be analyzed.

The loop at lines 40–49 traverses the list starting from x (see lines 37–38), where each iteration follows the pointer from the current node to the next node in the traversal (see lines 46–47). Because each iteration of the loop at lines 40–49 takes $O(1)$ steps, the runtime of the loop is bounded by the number of nodes reached during this traversal. Let $T(x, C, q)$ be the traversal performed by MTF(x) in the loop at lines 40–49, where C is the configuration at the start of this traversal. Because *op* claims x before calling MTF(x) (see lines 17–18, 20–21, and Observation 17), we have $C_\ell \preceq C$, where ℓ is the step at which x was claimed by *op*. Applying Corollary 61 with $C' = C$ and $C = C_\ell$ implies $T(x, C, q) \sqsubseteq P(x, C_\ell, q)$.

5.6.1 Setup: Case Analysis of Traversal Length by Set Membership.

Now we bound the number of nodes reached by the traversal $T(x, C, q)$ in the loop at lines 40–49 of MTF(x) depending on whether $k \in S(\ell)$, where $S(\ell)$ is the state of the abstract set immediately before ℓ (see Definition 50).

First we show that the *tail* node is always reachable from an active node, which helps give us a worst-case upper-bound on the number of nodes that can be traversed by MTF(x).

Invariant 62. *The tail node is always reachable from every active node.*

Proof. Assume *tail* is not reachable from an active node x in some configuration C. Let $(r_0, r_1, \ldots)$ be the sequence of nodes reachable from x. By Invariant 16, each r_j $(0 \leq j)$ is an active node. So by Property 2 **(i)**–Property 2 **(ii)** of Lemma 30, each r_j has a non-`NIL`*next* pointer, and the serial numbers of nodes in the sequence strictly decrease; that is, we have $r_j.serial > r_{j+1}.serial$ $(0 \leq j)$. Furthermore, the serial numbers of active nodes are distinct by Corollary 22 **(i)**, so for *tail* to not be some r_j, there must be some node $r_j \neq tail$ in the sequence with $r_j.serial > 0$ and $0 > r_{j+1}.serial$. But this contradicts Corollary 22 **(i)**, which

states that the serial number of an active node is positive. Hence, the assumption that *tail* is not reachable in C from x is false. □

Now we bound the number of nodes reached by MTF(x) by cases.

Case 1. $k \notin S(\ell)$**.** In the worst case, the traversal from x stops when it encounters a node q with serial number 0 (by the first condition at line 40), which by Corollary 22 **(i)** and the initialization of the list (see Section 3.2.2) implies that $q = tail$. Thus, by Corollary 61, $|T(x, C, q)| = |T(x, C, tail)| = O(|P(x, C_\ell, tail)|)$.

Case 2. $k \in S(\ell)$**.** By Lemma 21, there must be a node z claimed by an INSERT(k) operation before x such that there is no node y with $x.serial > y.serial > z.serial$ claimed by a DELETE(k) operation. Let $z(x)$ be the newest matching insert node of x at ℓ. By **Property 5** of Lemma 45, at line 40 we have $x.end \geq z(x).serial$ after reaching a node q with $q.serial \leq z(x).serial$. So by the exit condition at line 40, the traversal stops after reaching q.

Let u be the node visited before q in the traversal. Since $u.next = q$, we have $|T(x, C, q)| = |T(x, C, u)| + 1$. By Corollary 61, $T(x, C, u) \sqsubseteq P(x, C_\ell, u)$, so $|T(x, C, q)| = |T(x, C, u)| + 1 = O(|P(x, C_\ell, u)|)$.

5.6.2 Safe and Expiring Nodes

In **Case 1**, we have $|T(x, C, q)| = O(|P(x, C_\ell, tail)|)$, and in **Case 2**, we have $|T(x, C, q)| = O(|P(x, C_\ell, u)|)$. To bound the size of these paths, we partition the unmarked nodes in C_ℓ by defining *expiring* nodes and *safe* nodes. We then separately bound the expiring nodes, safe nodes, and marked nodes reachable from x in C_ℓ.

Definition 63. *Fix a node x claimed at step ℓ, and let $k = x.key$. An unmarked node e with key k and $e.serial < x.serial$ is an* ***expiring node*** *of x if either:*

(i) *there exists an unmarked insert or search node newer than e with key k, or*

(ii) *a node with a serial number in $(e.serial, x.serial)$ was claimed by a* DELETE(k) *operation.*

Any other unmarked node with key k and serial number less than $x.serial$ in C_ℓ is called a ***safe node*** *of x.*

Informally, the *expiring* nodes of x are unmarked nodes that were destined to be removed from the list before ℓ, while the *safe* nodes of x are unmarked nodes that were not.

By bounding the *expiring* and *safe* nodes of x, and counting the number of marked nodes reachable from x in C_ℓ, we can obtain bounds on the size of the paths $P(x, C_\ell, tail)$ and $P(x, C_\ell, u)$. We first bound the number of expiring nodes of x by $O(\dot{c}(op))$.

Bounding the expiring nodes of x.

Let E be the set of expiring nodes of x. In Lemma 66, we show $|E| = O(\dot{c}(op))$. We first show the number of expiring nodes whose operations are still pending at step ℓ is $O(\dot{c}(op))$. We then show that every expiring node $e \in E$ whose operation completed before ℓ can be injectively mapped to a distinct operation that is pending at ℓ. This implies that the number of expiring nodes in E claimed by completed operations is also $O(\dot{c}(op))$.

To construct the injective mapping, we use Lemma 65. The lemma states that if an insert or search node y has a newer node y' with the same key k and no node with key k has a serial number in ($y.serial$, $y'.serial$), then once both of the operations that claimed y and y' complete, the node y must be marked. Thus, for any node y that remains unmarked at ℓ, at least one of the two operations that claimed y or y' must still be pending at ℓ.

To see how Lemma 65 helps us construct the injective mapping, consider an expiring node e of x whose operation completed before ℓ. Let e' be the newest node with key k and $e'.serial > e.serial$. Since e is unmarked at ℓ, Lemma 65 implies that the operation that claimed e' is pending at ℓ. Furthermore, if e_1 and e_2 are two distinct expiring nodes of x whose operations both completed before ℓ, then $e_1.serial \neq e_2.serial$, so e_1 and e_2 can be mapped to distinct pending operations.

Observation 64. *If $y.end = -1$ for a node y claimed by an* Insert *or* Search*, then the operation that claimed y has not yet executed line 50 of* MTF*.*

Proof. If $y.end$ is not changed from its initial value -1 (see Figure 3.9), before the CAS at line 78 of the call IncreaseEnd(y, 0) at line 50 of MTF(y), then the CAS changes it to 0. If $y.end$ has changed, its value is increased by Lemma 41. Either way, it is at least 0 in the configuration after the CAS at line 78 and can only increase thereafter by Lemma 41. □

Lemma 65. *Let x and x' be nodes such that $x'.serial > x.serial$, $x.key = x'.key = k$, and $x.type \neq$ `delete`. Assume further that every active node y with $x'.serial > y.serial >$*

$x.serial$ has $y.key \neq k$. If the configuration C is after the operations that claimed x and x' have both completed, x is marked.

Proof. Let s be the last step that belongs to one of the two operations that claimed x and x'. To derive a contradiction, suppose x is not marked in the configuration after s.

Case 1. Assume x' was claimed by a DELETE(k) operation. Because $x'.serial > x.serial$ and x is unmarked at s, Lemma 35 implies that the traversal of DELETE(k) in REMOVEMATCHES(x') reaches x before it reaches any node with a serial number less than or equal to $x.serial$.

The traversal at lines 60–67 of REMOVEMATCHES(x') continues until it encounters a matching delete node for x' or the *tail* node with serial number 0 (line 64). Since x' is only node with key k and a serial number in ($x.serial$, $x'.serial$], and the traversal visits nodes in strictly decreasing serial order (by Property 2 **(ii)** of Lemma 30), it must eventually reach a node with serial at most $x.serial$. Thus the traversal reaches and collects x (line 65), after which REMOVEMATCHES(x') calls MARKANDREMOVE($*$, x) at line 74 of the loop at lines 71–74, which ensures $x.mark =$ `true` by line 92. Since x is unmarked in C, REMOVEMATCHES(x') must therefore be pending after s, which contradicts the assumption it completed.

Case 2. Assume x' was claimed by a successful SEARCH or a failed INSERT operation. Because $x'.serial > x.serial$ and x is unmarked through s, Lemma 35 implies that if the traversal at lines 37–49 of MTF(x') ever reaches a node with serial number at most $x.serial$, then it reaches x.

Assume for contradiction that MTF(x') never reaches any node with serial number at most $x.serial$. For this to happen, the traversal must stop while it is still visiting nodes with serial numbers larger than $x.serial$. By the loop condition at line 40 and Property 2 **(ii)** of Lemma 30, this can occur if either $x'.end \geq x.serial$ holds at line 40 before the traversal could reach x or if it encounters a delete node with the key k and a serial in ($x.serial, x'.serial$).

Because there is no node for the key k with a serial number in ($x.serial, x'.serial$), MTF(x') could not have stopped before reaching x due to encountering a delete node with key k with a serial number in ($x.serial, x'.serial$). The only way for MTF(x') to stop before reaching x is due to $x'.end \geq x.serial$ holding at line 40 before the traversal could reach x.

We now show that neither MTF(x') nor call RemoveMatches can set $x'.end > x.serial$.

- **MTF(x') cannot set $x'.end > x.serial$.**

  MTF(x') only calls IncreaseEnd(x', $*$) at line 43 or line 49 after it encounters a node with key k during its traversal. By the assumption that no operation on key k claimed a node with a serial number in $(x.serial, x'.serial)$, there is no node with key k in this range. Hence any node with key k that MTF(x') could use in a call to IncreaseEnd must have serial number less that or equal to $x.serial$. By Property 2 **(ii)** of Lemma 30, such a node would be visited only after the traversal reaches a node with serial number at most $x.serial$. In particular, since x is unmarked and reachable, Lemma 35 implies that MTF(x') would reach x before it could call IncreaseEnd with an argument at most $x.serial$. This contradicts the assumption that MTF(x') never visits a node with serial number at most $x.serial$.

- **RemoveMatches(v) cannot set $x'.end > x.serial$.**

  A call RemoveMatches(v) starts its traversal from v (lines 60–67) and moves through nodes in strictly decreasing order of serial numbers (again by Property 2 **(ii)** of Lemma 30). It collects nodes with key k into the vector *matches* at line 65, and when it finishes this traversal it can set $x'.end$ based on the first node in *matches* it encountered after x' (line 64).

  Because there is no node with key k whose serial number is in the range $(x.serial, x'.serial)$, the first node with key k it encountered during its traversal from v and has serial less than $x'.serial$ must have serial number at most $x.serial$. Hence, RemoveMatches(x') could only set $x.end$ to a value less than or equal to $x.serial$.

In both cases we obtain a contradiction. Therefore our assumption was false, and MTF(x') must reach x.

Moreover, since no node with key k has a serial number in $(x.serial, x'.serial)$, x is the first node with key k encountered by the traversal after x'. Hence, when MTF(x') records the first matching pair at lines 36–45, it records the pair $\langle *, x \rangle$.

MTF(x') would check at line 55 if x has $x.end = -1$, which by Observation 64,

implies the operation that claimed x has not yet executed line 50. It then sets $x.remove =$ `true` to notify the operation that claimed x to mark and remove x. If the operation that claimed x has not performed line 50 between x' finding that $x.end = -1$, and setting $x.remove =$ `true`, then it will read $x.remove =$ `true` at line 58, and mark x by calling MarkAndRemove on it at line 58. If MTF(x') determines that the operation still has not yet performed line 50 after setting $x.remove$, it calls MarkAndRemove($*$, x) at line 56, which ensures x is marked by lines 83–86. In any case, x is marked before both operations have completed.

Case 3. Assume x' was claimed by a failed Search, or a successful Insert operation. This implies that x was claimed by a failed Search operation, because if x were claimed by an Insert, Theorem 48 implies that x' was claimed by a successful Search or a failed Insert. And if x were claimed by a successful Search, there must exist an Insert node z with k and $x.serial > z.serial$ such that there is no node y claimed claimed by a Delete(k) with $x.serial > y.serial > z.serial$ (Invariant 43). By Lemma 21 and Theorem 48, this would imply that x' was claimed by a successful Search or a failed Insert. Hence x can only be claimed by a failed Search.

Now by Lemma 46, the operation that claimed x must have $x.end \leq 0$ at line 51 of its call to MTF(x) at line 18 , so $x.type =$ `search` at line 58 (the only line at which $x.type$ is changed is at line 52, which is not executed by the condition at line 58). Thus, MarkAndRemove($Head$, x) is called at line 58 of MTF(x) by the condition at line 58, and this call ensures that x is marked by lines 83–86.

Thus, we have established that the lemma holds for each possible type of operation that claims x'. □

Now we bound the number of expiring nodes of a claimed node x by point contention.

Lemma 66. *A node x claimed at step ℓ has $O(\dot{c}(op))$ expiring nodes.*

Proof. Let E be the set of expiring nodes of x. We have to show $|E| = O(\dot{c}(op))$.

Every unmarked node that is older than x in C_ℓ (other than *tail*) was claimed either by an operation that is still *active* at ℓ or by one that *completed* earlier (by Lemma 21). So we have $E = E_{pend} \cup E_{comp}$, where E_{pend} is the set of expiring nodes of x that are pending at ℓ, and E_{comp} is the set of expiring nodes of x that have completed before ℓ.

We handle these two classes separately and show there are $O(\dot{c}(op))$ nodes of each type.

(a) Bounding $|E_{pend}|$.

At most $\dot{c}(op)$ operations are pending at ℓ, and each may contribute at most one expiring node (i.e., the node it claimed) to E_{pend}. Hence, $|E_{pend}| = O(\dot{c}(op))$.

(b) Bounding $|E_{comp}|$.

Case 1. Node e was claimed by a DELETE.

A DELETE operation calls MARKANDREMOVE($Head$, e) on the node e it claimed at line 75 before it completes. By line 92 of MARKANDREMOVE($Head$, e), e is marked before the operation that claimed it completes and by Observation 28, y remains marked thereafter. expiring nodes are unmarked by Definition 63, so there are no expiring nodes of x of this type.

Case 2. Node e was claimed by failed SEARCH.

A failed SEARCH invokes MARKANDREMOVE on its claimed node y (line 58), before completing (by Section 4.7, line 58). At line 92, y is marked before completion, and by Observation 28 it remains marked thereafter. Since expiring nodes are unmarked by Definition 63, there are no expiring nodes of x of this type.

Case 3. Node e was claimed by an INSERT or a successful SEARCH.

We show that we can define a one-to-one function from each node $e \in E_{comp}$ of these types to an operation that is pending at ℓ, proving that $|E_{comp}| = O(\dot{c}(op))$.

Let e be such a node with key k'. Since e is a expiring node of x, Definition 63 guarantees the existence of at least one node v with $x.serial > v.serial > e.serial$ in C_ℓ such that either

Subcase 1. v is an unmarked insert or search node for k' and no node with key k' claimed by a DELETE(k') has a serial number in $(e.serial, v.serial)$, or

Subcase 2. v is a node that was claimed by a DELETE(k') operation.

In either case, there exists a node newer than e that was claimed by an operation

on the key k'. For each such e, let v_e be the *oldest* node with key k' and serial number in $(e.serial, x.serial)$.

Because the operation that claimed e has completed, Lemma 65 implies that, for e to remain unmarked at step ℓ, the operation that claimed v_e must still be pending at ℓ; otherwise, e would be marked, contradicting the fact that e is an expiring node.

We now show that the mapping $e \mapsto v_e$ is one-to-one. Suppose, for contradiction, that there exist distinct expiring nodes $e, e' \in E_{pend}$ with $v_e = v_{e'}$. Without loss of generality, assume $v_e.serial = v_{e'}.serial > e'.serial > e.serial$. Then e' is a node with key k' strictly between e and v_e in serial order, which contradicts the definition of v_e as the oldest node newer than e with key k'. Hence, if $e \neq e'$, then $v_e \neq v_{e'}$.

Therefore, each expiring node $e \in E_{pend}$ with key k' of this type is injectively mapped to a distinct pending operation (the one that claimed v_e). Since there are $O(\dot{c}(op))$ pending operations at ℓ, $|E_{pend}| = O(\dot{c}(op))$.

We have $|E_{pend}| = O(\dot{c}(op))$ and $|E_{comp}| = O(\dot{c}(op))$, so we have $|E| = |E_{pend} \cup E_{comp}| = O(\dot{c}(op))$, so the lemma is proved. □

Bounding safe nodes.

To bound the number of safe nodes of x on a path from x in C_ℓ, we prove the following lemma.

Definition 67. *Fix a node x that was claimed at step ℓ, and let i be an integer with $0 < i < x.serial$.*

Let $\mathcal{S}_i$ denote the set of safe nodes of x whose serial numbers lie in the range $[i, x.serial)$.

Let W_i denote the set of keys in $S(\ell)$, where $S(\ell)$ denotes the keys in the abstract set immediately before step ℓ (see Definition 50) for which some operation claimed a node with a serial number in the range $[i, x.serial)$.

Lemma 68. *For every node x claimed at step ℓ and every integer i with $0 < i < x.serial$,*

$$|\mathcal{S}_i| = O\big(|W_i| + \dot{c}(op)\big).$$

Proof. Partition the safe nodes in the range into

$$\mathcal{S}_i^{\text{in}} = \{\, y \in \mathcal{S}_i \mid y.key \in S(\ell) \,\}, \qquad \mathcal{S}_i^{\text{out}} = \mathcal{S}_i \setminus \mathcal{S}_i^{\text{in}}.$$

Safe nodes of x whose keys are not in $S(\ell)$.

Let $y \in \mathcal{S}_i^{\text{out}}$, so $y.key \notin S(\ell)$. By the definition of safe node, no DELETE($y.key$) operation was linearized after y was claimed. Since $y.key \notin S(\ell)$, some operation on $y.key$ must have removed $y.key$ from the abstract set. Hence the operation that claimed y must be either a DELETE($y.key$) operation or a failed SEARCH($y.key$) operation.

Both of these operations mark the node they claimed before completing (by lines 75 and 58). Thus, if y is unmarked at step ℓ, the operation that claimed y must still be pending at ℓ. By Observation 18, this operation is unique for each such y.

Therefore, each $y \in \mathcal{S}_i^{\text{out}}$ can be mapped to a distinct operation that is pending at ℓ. Since the number of pending operations at ℓ is $O(\dot{c}(op))$, we obtain

$$|\mathcal{S}_i^{\text{out}}| = O\big(\dot{c}(op)\big).$$

Safe nodes of x whose keys are in $S(\ell)$. Now take $y \in \mathcal{S}_i^{\text{in}}$, so $y.key \in S(\ell)$ and $i \le y.serial < x.serial$. By the definition of safe node, for each key $k' \in S(\ell)$ there is *at most one* safe node with key k' and a serial number in $[i, x.serial)$: it is the newest unmarked insert or search node for k' in that range such that no DELETE(k') was claimed with a serial number in $(y.serial, x.serial)$.

In particular, if $y \in \mathcal{S}_i^{\text{in}}$ has key k', then

- the operation that claimed y has serial in $[i, x.serial)$, so $k' \in W_i$, and
- there is no other safe node in $\mathcal{S}_i^{\text{in}}$ with key k'.

Hence the map

$$\varphi : \mathcal{S}_i^{\text{in}} \to W_i, \qquad \varphi(y) = y.key,$$

is injective. Therefore,

$$|\mathcal{S}_i^{\text{in}}| \le |W_i|.$$

Since $\mathcal{S}_i = \mathcal{S}_i^{\text{in}} \cup \mathcal{S}_i^{\text{out}}$, we have

$$|\mathcal{S}_i| = |\mathcal{S}_i^{\text{in}}| + |\mathcal{S}_i^{\text{out}}| \leq |W_i| + O\big(\dot{c}(op)\big) = O\big(|W_i| + \dot{c}(op)\big),$$

which proves the lemma. □

5.6.3 Case Analysis of Path Length by Set Membership

Before proving the final bounds on sizes of the paths, we show there can be $O(\dot{c}(op))$ marked nodes reachable from x when x is claimed.

Lemma 69. *At any moment during an operation op, there are $O(\dot{c}(op))$ marked nodes reachable from $Head$.*

Proof. Let y be a marked node reachable from $Head$ in some configuration C during the execution of *op*. The mark on y must have been set at line 84 of a call to MarkAndRemove($*$, y).

By Lemma 37, once MarkAndRemove($*$, y) completes, y becomes unreachable from $Head$ in the configuration at completion and in all subsequent configurations. Therefore, if y is reachable in C, the call to MarkAndRemove($Head, y$) that marked it has not yet completed–i.e., it is still pending in C.

Since $O(\dot{c}(op))$ operations can be pending, and each pending MarkAndRemove call contributes at most one such marked node that remains reachable while it is pending, there can be $O(\dot{c}(op))$ marked nodes reachable from $Head$ in C. □

Because $Head = x$ in C_ℓ, the number of marked nodes M in a path beginning from x in C_ℓ is $O(\dot{c}(op))$.

Path length bounds.

Case 1. $k \notin S(\ell)$**.** We have shown that $T(x, C, q) = T(x, C, tail)$, where C is the configuration in which the traversal begins, q is the node where the traversal ends, and that

$$|T(x, C, tail)| = O\big(|P(x, C_\ell, tail)|\big).$$

The path $P(x, C_\ell, tail)$ is composed of the *tail* node together with the safe nodes in $\mathcal{S}_1$, the expiring nodes in E, and the marked nodes in M. Thus

$$|P(x, C_\ell, tail)| = O(|\mathcal{S}_1| + |E| + |M|).$$

By Lemma 68, $|\mathcal{S}_1| = O(|W_1| + \dot{c}(op))$, and by Lemma 21, the keys that appear in W_1 are exactly the keys in $S(\ell)$, so $W_1 = S(\ell)$. By Lemma 69, since $x = Head$ in C_ℓ, the number of marked nodes reachable from x is $|M| = O(\dot{c}(op))$. By Lemma 66, we have $|E| = O(\dot{c}(op))$.

Because $|S(\ell)| = n$, combining these bounds gives

$$O(|P(x, C_\ell, tail)|) = O(|\mathcal{S}_1| + |E| + |M|) = O(|S(\ell)| + \dot{c}(op)) = O(n + \dot{c}(op)).$$

Case 2. $k \in S(\ell)$

We have shown that $|T(x, C, q)| = O(|T(x, C, u)|) = O(|P(x, C_\ell, u)|)$, where $u.next = q$ is the final link followed by the traversal.

Let z be the newest insert node claimed by an operation on the key k that has completed before ℓ. Then $z.serial > z(x).serial$, where $z(x)$ is the newest insert node with the key k at ℓ, and by **Property 4** of Lemma 45 and line 40 we also have

$$z(x).serial > u.serial.$$

Thus, by Definition 67 and Lemma 21,

$$|W_{u.serial}| = O(|W_{z(x).serial}|) = O(|W_{z.serial}|) = O(|w(op)|).$$

By Lemma 68, $|\mathcal{S}_{u.serial}| = O\big(|W_{u.serial}| + \dot{c}(op)\big) = O\big(|w(op)| + \dot{c}(op)\big)$. Hence there are $O(|w(op)|)$ safe nodes in $P(x, C_\ell, u)$.

There are also $|E| = O(\dot{c}(op))$ expiring nodes in $P(x, C_\ell, u)$ and $|M| = O(\dot{c}(op))$ marked nodes in $P(x, C_\ell, u)$.

Therefore,

$$|P(x, C_\ell, u)| = O\big(|w(op)| + \dot{c}(op)\big),$$

and consequently,

$$|T(x, C, q)| = O\big(|P(x, C_\ell, u)|\big) = O\big(|w(op)| + \dot{c}(op)\big).$$

5.6.4 Final Bounds

The amortized cost of a call to MTF(x) by an operation *op* linearized at step ℓ is:

$$\text{Cost}(\text{MTF}(x)) = \begin{cases} O\big(|w(op)| + \dot{c}(op)\big), & \text{if } k \in S(\ell), \\ O(n + \dot{c}(op)), & \text{if } k \notin S(\ell). \end{cases}$$

5.7 Analysis of RemoveMatches

We first analyze the traversal performed in lines 64–67 of RemoveMatches. Let x be the node claimed by a Delete(k) operation that invokes RemoveMatches(x). By Invariant 62, *tail* is always reachable from x, and $tail.serial = 0$. Thus, in the worst case, the traversal proceeds from x to *tail*, terminating when the exit condition at line 64 becomes true once *tail* is read. We now analyze the amortized cost of this traversal depending on whether RemoveMatches returns `true` or `false`.

5.7.1 Amortized Runtime of *op* if $k \in S(\ell)$

Let y be the oldest insert node with key k when x is claimed, such that no Delete(k) operation has claimed a node with a serial number in ($y.serial$, $x.serial$). Because a successful RemoveMatches(x) implies that k was present at the time of publication (by Theorem 49), such a node y must exist by Lemma 21 and the linearization order (Definition 13).

Let y_0 denote the newest insert node with key k and $x.serial > y_0.serial$ at the time x is claimed. Recursively define y_{j+1} to be the newest insert node with key k with $y_j.serial > y_{j+1}.serial$ when y_j is claimed, continuing until the oldest insert $y_i = y$ is reached. No delete node with k has a serial number between $y_j.serial$ and $y_{j+1}.serial$ ($0 \leq j < i$) by construction.

As established in Section 5.6, the operation that claims y_j already pays for the traversal from y_j to y_{j+1} at the time y_j is claimed, since the most recent operation on key k that completed before this point is either the one that claimed y_{j+1} or an older one. Similarly, the operation

that appended the oldest insert $y_i = y$ pays for the final segment from y_i to *tail*. Thus, every subpath from y_j to y_{j+1} $(0 \leq j < i)$, as well as the terminal path from y_i to *tail*, has already been charged to a preceding insert operation.

By Lemma 32, two distinct active nodes x and y with $x.serial > y.serial$ cannot differ in reachability over any smaller serial range. Formally:

Corollary 70. *Let x and y be two active nodes with $x.serial > y.serial$. Then, for any interval $[i, j] \subseteq [0, y.serial]$, all active nodes with serial numbers in $[i, j]$ that are reachable from x when it was claimed are also reachable from y when it was claimed.*

For each $j < i$, define the interval of serial numbers

$$I_j = (y_{j+1}.serial,\, y_j.serial].$$

By construction, no delete node for k appears within I_j. By Corollary 70, all nodes in I_j that are reachable from any later node (in particular, from x) were also reachable from y_j when y_j was claimed. If each operation that claimed y_j is charged twice for traversing I_j (as in Section 5.6), a later successful RemoveMatches traversing the same region incurs no additional amortized cost. The final segment from y_i to *tail* is similarly paid for by the operation that appended y_i.

By Invariant 39, once a delete node z marks itself at line 74, no nodes with key k remain reachable from z. Together with the absence of delete nodes in each I_j, this ensures that at most one successful RemoveMatches—either concurrent or earlier—can traverse a given segment. Any subsequent RemoveMatches halts at the first unmarked delete node (line 71), avoiding unecessary traversals.

Let x' be the node claimed by the next successful Delete(k) operation after the successful Delete(k) operation that claimed x. Then there exists a node $y_{i'}$ claimed by an Insert(k) operation with $y_{i'}.serial \in (x.serial,\, x'.serial)$ that is older than all other insert nodes in that range (by Lemma 21 and our linearization shown in Definition 13). This $y_{i'}$ defines a new *amortization frontier*: the operation that claimed $y_{i'}$ pays for traversing all nodes with serial numbers in $(0,\, y_{i'}.serial]$ that were reachable when $y_{i'}$ was claimed. Consequently, no operation that claimed some y_j or y_i pays more than twice for the segment it paid for. Thus the amortized bounds of the operations that claim those nodes remain within a constant factor of 2 of their existing bounds.

Finally, since there are $O(|w(op)| + \dot{c}(op))$ nodes on the path from x to y_0 (see Section 5.6 for the argument; it is identical to the case of a successful MTF), and by Lemma 59, a successful RemoveMatches incurs amortized traversal cost $O(|w(op)| + \dot{c}(op))$.

5.7.2 Amortized Runtime of op if $k \notin S(\ell)$

If RemoveMatches(x) fails, then at the step when x is claimed, the number of safe nodes in the list is $O(n)$, where n is the number of distinct keys in the set. There are $O(\bar{c})$ expiring nodes of x and $O(\dot{c})$ marked nodes (by Lemma 66 and Lemma 69) reachable from x, so the total number of nodes on the path from x to *tail* when x is claimed is $O(n + \dot{c})$. Hence, by the fact that RemoveMatches(x) traverses a subsequence of that path (Lemma 59), the traversal from x to *tail* at lines 60–67 takes amortized $O(n + \dot{c})$ steps.

5.7.3 Final Bounds

Amortized removal cost.

Each operation op' that appends a node contributes $O(\dot{c}(op'))$ steps toward its potential removal by RemoveMatches(x). Because every node is removed at most once by a RemoveMatches (by Invariant 39, no two calls to RemoveMatches can collect the same matching node), and the cost of each invocation of MarkAndRemove at line 74 costs $O(\dot{c}(op))$ steps (see Section 5.4), each of these lines is free. There are $O(\dot{c}(op))$ matches of a node x when x was claimed by the fact that there are $O(\dot{c}(op))$ expiring nodes of x and $O(\dot{c}(op))$ marked nodes reachable from x (by Lemma 66 and Lemma 69), so the total cost of the removal loop at lines 71–74 is $O(\dot{c})$.

The removal of x at line 75 costs $O(\dot{c}(op))$ steps by the amortized analysis in Section 5.4, and line 76 takes $O(1)$ steps.

Conclusion.

Combining these arguments, the amortized step complexity of RemoveMatches(x) is:

$$\text{Cost}(\textsc{RemoveMatches}(x)) = \begin{cases} O\big(|w(op)| + \dot{c}(op)\big), & \text{if } k \in S(\ell), \\ O(n + \dot{c}(op)), & \text{if } k \notin S(\ell), \end{cases}$$

where ℓ is the step at which node x is claimed.

5.8 Analysis of Publish

Lines 25 and 35 take $O(1)$ steps. The remaining cost comes from the loop at lines 26–34. Every line within a single iteration of this loop also takes $O(1)$ steps, so all that remains is to analyze the amortized number of iterations.

PUBLISH(d) repeatedly executes the loop at line 26–34 until the CAS at line 29 successfully stores d in the sentinel node (see lines 25, 27–30, and 34). So an operation can perform at most one iteration of the loop at lines 26–34 in which the CAS at line 29 succeeds.

A failed CAS by an operation *op* implies that another operation *op*′ completed a successful CAS at line 29 after *op* read *Head* at line 27; hence, we charge *op*′ $O(\dot{c}(op'))$ steps to pay for each iteration of the loop at lines 26–34 by each such *op*. Hence, each iteration of an operation *op* in which the CAS at line 29 fails is free. Therefore, a call to PUBLISH by any operation *op* takes an amortized $O(\dot{c}(op))$ steps to pay for a single iteration in which the CAS at line 29 succeeds.

5.9 Analysis of Insert, Search, and Delete

Each operation *op*, whether SEARCH(k), INSERT(k), or DELETE(k), first constructs its OPDATA object d (line 16, 19, or 22) in $O(1)$ steps, then calls PUBLISH(d) (line 17, 20, or 23) to claim a node x and linearize at step ℓ (by Observation 17 and Definition 13). By Section 5.8, this call to PUBLISH takes amortized $O(\dot{c}(op))$ steps.

Finally, if *op* is a SEARCH(k) or INSERT(k) operation, it calls MTF(x) (at lines 18 and 21, respectively), and if *op* is a DELETE(k) operation, it calls REMOVEMATCHES(x) (at line 24). Both procedures take amortized $O(|w(op)|+\dot{c}(op))$ steps $k \in S(\ell)$, where $S(\ell)$ is the abstract set immediately before ℓ, and $O(n + \dot{c}(op))$ steps otherwise, where $|S(\ell)| = n$ (proved in Section 5.6–Section 5.7). Since these bounds dominate the $O(1)$ work for constructing d and the $O(\dot{c}(op))$ steps of PUBLISH(d), these bounds give the final amortized runtime of *op*.

Theorem 71. *Let op be any* SEARCH(k), INSERT(k), *or* DELETE(k) *operation, and let ℓ be its linearization point. Let $S(\ell)$ be the abstract set immediately before ℓ, and let $n = |S(\ell)|$. Then the amortized number of steps performed by op is*

$$Cost(op) = \begin{cases} O\big(|w(op)| + \dot{c}(op)\big), & \text{if } k \in S(\ell), \\ O\big(n + \dot{c}(op)\big), & \text{if } k \notin S(\ell), \end{cases}$$

where $w(op)$ is the concurrent working set of op (see Definition 51) and $\dot{c}(op)$ is the point contention of op.

Chapter 6

Online Analysis

6.1 A Proof of Static Optimality

6.1.1 Setup and Notation

We consider a static list; that is, a list with no update operations. Let the list contain the keys $\{k_1, k_2, \ldots, k_n\}$, indexed in decreasing order of access frequency within a set σ of SEARCH operations, where each k_i $(1 \leq i \leq n)$ is accessed at least once. Let f_i denote the number of occurrences of key k_i in σ, such that

$$\sum_{i=1}^{n} f_i = |\sigma|, \qquad f_1 \geq f_2 \geq \cdots \geq f_n, \qquad \forall i,\ f_i \geq 1.$$

In contrast to competitive analysis in the sequential setting, where accesses occur one after another and σ represents a total *access sequence* (see [14]), here σ is a *set of concurrent accesses*. SEARCHES may be concurrent.

6.1.2 Cost of the Optimal Static List

The optimal static list (OPT) stores items in decreasing order of frequency. Hence, the cost of accessing k_i is proportional to its position i in that list:

$$Cost_{\mathrm{OPT}}(\sigma) = \sum_{i=1}^{n} f_i \cdot i.$$

6.1.3 Global Adversarial Scheduling Model

We fix the set σ of SEARCH operations, but allow an adversarial scheduler to determine when each operation on OPT and on our MTF list begins, is linearized, and completes. The only restriction placed on the adversary is that every operation in σ executed by the adversary must complete, which is an implicit assumption for sequences of requests in competitive analysis more generally.

We may assume that the initial state of the MTF list was created by executing a "ghost" sequence of INSERT operations of the form $(\textsc{Insert}(k_n), \textsc{Insert}(k_{n-1}), \ldots, \textsc{Insert}(k_1))$. This sequence may or may not have actually occurred before σ is scheduled, but the assumption is safe: the ghost initialization creates exactly the same initial state that would otherwise be constructed "out of thin air" for the operations in σ to run on.

Because every SEARCH in σ completes, and the ghost initialization gives each key k_i an operation on k_i that completed before the adversary schedules σ, the *window* of each operation in σ linearized on our MTF list is well-defined.

Definition 72. *Let op be an operation on key k linearized at step ℓ, and assume some operation on k completed before ℓ. Let ℓ^k be the most recent linearization point of any such completed operation.*

The ***window*** *of op is*

$$\mathcal{W}(op) := \{\, \ell' \;\mid\; \ell^k \preceq \ell' \prec \ell \,\}.$$

Let op be any operation on key k_i in σ that is linearized on our MTF list by the adversary at step ℓ. Because $\mathcal{W}(op)$ is well-defined, the working set $w(op)$ is also well-defined. By Definition 51,

$$w(op) = \{\, k_j \in S(\ell) \mid \text{some operation on } k_j \text{ has a linearization point } \ell' \in \mathcal{W}(op) \,\}.$$

Furthermore, $k_i \in S(\ell)$ because the set of keys is static. Thus, by Chapter 5 (with final bounds stated in Theorem 71), op takes an amortized

$$O\big(|w(op)| + \dot{c}(op)\big)$$

steps on our MTF list.

Moreover, by Definition 72 and Definition 51, $|w(op)|$ is exactly the number of distinct keys in the set that were accessed by operations whose linearization points lie in $\mathcal{W}(op)$.

Therefore, the adversary's goal is to maximize the number of distinct keys accessed by operations with linearization points in $\mathcal{W}(op)$, thereby maximizing $|w(op)|$.

Let $op_{i,t}$ denote the t-th SEARCH(k_i) linearized by the adversary. The adversary acts as follows:

- Any key accessed as frequently as, or more frequently than, k_i has an operation linearized in every window $\mathcal{W}(op_{i,t})$, and thus always appears in $w(op_{i,t})$.
- Any key $k_j \neq k_i$ with $f_j < f_i$ cannot appear in every window $\mathcal{W}(op_{i,t})$, because there are only f_j operations on k_j but f_i distinct windows for k_i. To maximize $\sum_t |w(op_{i,t})|$, the adversary places each linearization point of k_j into a different window of k_i whenever possible.

6.1.4 Bounding the Total Cost

We now analyze the total cost of our MTF list for the adversarial scheduling of σ. For each $op_{i,t}$, let $L(op_{i,t})$ be the number of distinct keys accessed less frequently than k_i whose operations have linearization points in $\mathcal{W}(op_{i,t})$.

Then

$$|w(op_{i,t})| = (i-1) + L(op_{i,t}) + O(\dot{c}(op)),$$

because the adversary linearizes an access for each key k_j with $f_j > f_i$ at a point in $\mathcal{W}(op_{i,t})$, accounting for the $(i-1)$ most frequent keys.

As such, the total cost of executing σ on our MTF list is

$$\begin{aligned} Cost_{\text{MTF}}(\sigma) &= \sum_{i=1}^{n} \sum_{t=1}^{f_i} |w(op_{i,t})| + O(\dot{C}_\sigma) \\ &= \sum_{i=1}^{n} f_i(i-1) + \sum_{i=1}^{n} \sum_{t=1}^{f_i} L(op_{i,t}) + O(\dot{C}_\sigma), \end{aligned} \tag{6.1}$$

where $O(\dot{C}_\sigma)$ is the total point contention of all operations over the execution of σ by MTF. The second term in (6.1) counts all accesses to lower-frequency items that occur between consecutive accesses to higher-frequency keys. In the worst case, each lower-frequency item

k_j (for $j > i$) can contribute at most once to each $\mathcal{W}(op_{i,t})$ of k_i, yielding

$$\sum_{i=1}^{n} \sum_{t=1}^{f_i} L(op_{i,t}) \;\leq\; \sum_{i=1}^{n} \sum_{j>i} f_j.$$

Thus,

$$\begin{aligned} Cost_{\text{MTF}}(\sigma) &\leq \sum_{i=1}^{n} f_i(i-1) + \sum_{i=1}^{n} \sum_{j>i} f_j + O(\dot{C}_\sigma) \\ &= 2 \sum_{i=1}^{n} f_i(i-1) + O(\dot{C}_\sigma) \\ &\leq 2 \cdot Cost_{\text{OPT}}(\sigma) + O(\dot{C}_\sigma). \end{aligned}$$

Therefore, under an adversarial scheduler and assuming every key is accessed at least once in the lookup set σ,

$$\boxed{Cost_{\text{MTF}}(\sigma) \leq 2 \cdot Cost_{\text{OPT}}(\sigma) + O(\dot{C}_\sigma)}.$$

Thus, our list is statically optimal within a constant factor of 2, with an additive $O(\dot{C}_\sigma)$ term accounting for contention. In the absence of contention (i.e., $O(\dot{C}_\sigma) = 0$), the bound simplifies to

$$Cost_{\text{MTF}}(\sigma) \leq 2 \cdot Cost_{\text{OPT}}(\sigma).$$

Notice that every sequential execution can be represented by the adversary in our model. Thus, when contention is absent, our algorithm achieves the same 2-competitive ratio as the sequential move-to-front list with respect to its static optimum [14].

Chapter 7

Space Analysis

We bound the number of nodes that can be reached from $Head$ by following links or backlinks in a configuration C.

Definition 73. *A **pointer-walk** from a node x in a configuration C is a sequence of nodes $(x = v_0, v_1, \ldots, v_k)$ such that for every $0 \leq i < k$, $v_i.next = v_{i+1}$ or $v_i.backlink = v_{i+1}$ holds in C.*

Let n be the number of distinct keys in the list in configuration C. We bound the number of nodes that can be reached by a pointer-walk from $Head$ in C. We first consider nodes that are unmarked in C and then consider nodes that are marked in C.

Nodes unmarked in C.

Let x be the newest node claimed by some operation op' at step ℓ such that $C_\ell \preceq C$. By Lemma 21, in configuration C either $Head = x$ or $Head.next = x$.

Case 1. $Head = x$ **in** C**.**

By Lemma 68 and Lemma 66, at most $O(n + \dot{c}(op'))$ unmarked (safe or expiring) nodes can be reached from x in any configuration $C_\ell \preceq C$. Since $\dot{c}(op') \leq \dot{c} = \max_{op} \dot{c}(op)$, at most $O(n + \dot{c})$ unmarked nodes can be reached from $Head$ in C.

Case 2. $Head \neq x$ **in** C**.**

In this case, $Head.next = x$. There exists a configuration $C' \preceq C$ in which $Head = x$. Applying the argument of **Case 1** to C' shows that at most $O(n + \dot{c})$ unmarked nodes can be reached from x. Since the sentinel contributes only one additional node, at most $O(n + \dot{c})$ unmarked nodes can be reached from $Head$ in C.

By Lemma 35, any unmarked node reached by a pointer-walk from $Head$ in C is also reached

by a forward traversal. Hence, the number of unmarked nodes reached by a pointer-walk from $Head$ in C is $O(n + \dot{c})$.

Nodes marked in C.

Let y be a marked node in configuration C such that the call MarkAndRemove$(*, y)$ that marked y has completed. By Lemma 52, during this call there exist nodes x' and y' such that $y'.backlink = x'$, and at some earlier step x' was active with $x'.next = y'$.

Assume, for contradiction, that y is reached from $Head$ in C by a pointer-walk ($Head = v_0, v_1, \ldots, v_k$). Let $I = [y'.serial, x'.serial]$, and let j be the smallest index $0 \leq j \leq k$ such that $v_j.serial \in I$. Then $v_{j-1}.serial \notin I$, and the step from v_{j-1} to v_j follows either a *next* pointer or a *backlink* in C.

By Property 2 **(ii)** of Lemma 30, exactly one of the following holds:

Case 1. $v_{j-1}.serial > x'.serial$ and $v_{j-1}.next = v_j$.

Case 2. $v_{j-1}.serial < y'.serial$ and $v_{j-1}.backlink = v_j$.

In **Case 1**, we have $v_{j-1}.serial > x'.serial > v_j.serial > y'.serial$ and both $v_{j-1}.next = v_j.next$ and $x'.next = y'.next$ hold, contradicting Property 2 **(iv)** of Lemma 30.

In **Case 2**, by Lemma 27, $v_j.next = v_{j-1}$ held at an earlier step, yielding $v_j.serial > x'.serial > v_{j-1}.serial > y'.serial$ and again $v_j.next = v_{j-1}.next$ together with $x'.next = y'.next$, contradicting Property 2 **(iv)** of Lemma 30.

Thus, no such index j exists, and y cannot be reached from $Head$ by a pointer-walk in C.

Therefore, the only marked nodes that can be reached by a pointer-walk from $Head$ in C are those marked by *pending* calls to MarkAndRemove, of which there are at most $O(\dot{c})$.

Final Bound.

A pointer-walk beginning from $Head$ in configuration C can reach at most $O(n+\dot{c})$ unmarked nodes and $O(\dot{c})$ marked nodes. Hence, the total number of nodes that can be reached by such a pointer-walk is

$$O(n + \dot{c}).$$

Chapter 8

Summary of Analytical Results

Amortized Step Complexity

Each SEARCH, INSERT, or DELETE operation completes in amortized

$$O(|w(op)| + \dot{c}(op))$$

steps when the accessed key k is in the set immediately before ℓ, where $w(op)$ denotes the concurrent working set of k at op's linearization point ℓ, and $\dot{c}(op)$ is op's point contention. If k is absent, the operation takes $O(n + \dot{c})$ steps, where n is the number of keys in the set immediately before ℓ.

Static Optimality

Compared to an optimal static list serving the same set of accesses σ, our structure is 2-competitive up to an additive contention term:

$$\mathrm{Cost}_{\mathrm{MTF}}(\sigma) \leq 2\,\mathrm{Cost}_{\mathrm{OPT}}(\sigma) + O(\dot{C}_\sigma),$$

irrespective of scheduling, where $\dot{C}_\sigma$ represents the cumulative point contention over the execution of σ by MTF.

Table Summary

Property	**Bound**	**Description**
Per-operation (op) cost (successful)	$O(\lvert w(op)\rvert + \dot{c}(op))$	Adaptive to the concurrent working set size plus contention
Per-operation (op) cost (unsuccessful)	$O(n + \dot{c}(op))$	Linear in the number of keys plus contention
Static optimality (global adversarial model)	$O(\mathrm{Cost}_{\mathrm{OPT}}(\sigma) + \dot{C}_\sigma)$	Within a constant factor of the optimal offline fixed list for access sequence σ, up to additive contention
Nodes reached from $Head$	$O(n + \dot{c})$	Linear shared-memory footprint in the number of keys plus contention

Table 8.1: Summary of analytical results for our concurrent move-to-front list.

Chapter 9

Future Directions

9.1 Competitive Analysis

A natural question concerns whether the additive contention term in our competitive ratio is unavoidable. When compared against a static optimal algorithm, this term appears unavoidable: any concurrent algorithm must pay for synchronization costs that the static optimal avoids by executing a read-only workload. The picture becomes less clear when comparing our list to a *dynamic optimal list* as it would likely incur synchronization overhead for restructing. Even if a dynamic optimal could strategically stall operations to reduce contention, such stalls would likely carry comparable costs, suggesting an intrinsic tradeoff between synchronization and adaptivity. More broadly, how our list performs relative to a dynamically optimal algorithm–under a reasonable concurrent model of competitive analysis–remains an open question. Applying Boyar et al.'s fully adversarial model to our list may be an interesting exercise [15].

9.2 Shared-Memory Caches

Another promising direction concerns the list's application to shared-memory caching. Our MTF list naturally extends to a shared-memory LRU cache: cache lookups and insertions map directly to Search and Insert operations, while each append moves the accessed key toward the head, preserving recency order. When the cache reaches capacity, a cooperative cleanup can be triggered before the next append: processes traverse backlinks starting from the tail (to which a shared pointer is maintained) to locate and remove the tail's predecessor, representing either the least recently used key or an obsolete operation node, by invoking MarkAndRemove. Making this cleanup routine efficient may require a doubly linked variant to make the tail's predecessor directly reachable, which can easily be achieved by cooperatively setting a node's backlink whenever a new node is appended to the head.

A more scalable LRU-like cache can be constructed by combining an LRU variant of our list with a lock-free hash table such as Michael's [33]. Each hash bucket can maintain its own instance of the list as a per-bucket chaining structure, backed by pre-allocated memory blocks. Eviction occurs locally by removing the tail node of each bucket, yielding a fully lock-free cache that preserves LRU-like ordering without global synchronization. This distributed design partitions contention across buckets and enables independent eviction with minimal coordination. Theoretical justification for this approach is provided by Ji et al. [28], who showed that with a well-chosen hash function, the asymptotic hit rate of a distributed LRU cache converges to that of a single global LRU cache when each local cache is sufficiently large.

9.3 Concurrent Structures with Adaptive Bounds

Our work began with the intent to formally capture self-adjustment in concurrent data structures. A concurrent list with a working set size bound per operation is an important first step, but an adaptive bound has yet to be proven for other concurrent data structures. Can the techniques used for the design and analysis of our list be used to develop a lock-free tree or skip-list with adaptive bounds?

Another open question relates to our definition of concurrent working set. As explained in Section 5.1, our definition of concurrent working set differs from the sequential definition applied directly to the linearization ordering. Can a concurrent set implementation be designed whose operation cost uses exactly the sequential working-set definition taken over the linearization ordering (plus contention)? More generally, what other adaptive bounds can be shown for concurrent structures?

9.4 Wait-Free Variants

Finally, can a wait-free variant of our list be achieved by aggregating operations up an ordering tree and linearizing them cooperatively at the root, as in recent wait-free constructions [10, 35]? Asbell and Ruppert [10] achieve a wait-free deque by collecting enqueue operations into batches represented by persistent red-black trees at each level of an ordering tree, which are then applied atomically to a persistent red-black tree in the root that represents the state of the deque. Future work could explore whether combining Asbell and Ruppert's batching techniques with the persistent-list of Kaplan and Tarjan [30] yields an efficient wait-free self-adjusting data structure.

Chapter 10

Conclusion

This thesis introduces the first concurrent data structure with an adaptive bound: a lock-free move-to-front (MTF) list whose cost per operation matches the locality measure acheived by its sequential counterpart. We prove that every operation *op* on our list runs in amortized $O(|w(op)| + \dot{c}(op))$ steps, where $|w(op)|$ denotes the operation's *concurrent working-set size* and $\dot{c}(op)$ denotes its point contention. We further show the list is 2-competitive with the optimal offline static list under any adversarial concurrent schedule of the same set of operations.

These results demonstrate that adaptive, locality-aware behavior can coexist with strong non-blocking guarantees in shared-memory systems. The analytical framework developed here, including the concept of a *concurrent working set* and the combined use of a *global adversarial scheduler* and a static optimal for competitive analysis, offers a foundation for designing and analyzing concurrent and adaptive versions of more advanced structures such as trees and heaps.

Chapter 11

References

[1] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed Computing*, 27(6):393–417, 2014.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 401–411, 1994. doi:10.1109/SFCS.1994.365676.

[3] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. The splay-list: A distribution-adaptive concurrent skip-list. *Distributed Computing*, 36(3):395–418, 2023.

[4] Susanne Albers. Improved randomized on-line algorithms for the list update problem. *SIAM Journal on Computing*, 27(3):682–693, 1998.

[5] Susanne Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1):3–26, 2003.

[6] Susanne Albers, Lene M Favrholdt, and Oliver Giel. On paging with locality of reference. In *Proceedings of the thiry-fourth annual ACM Symposium on Theory of Computing*, pages 258–267, 2002.

[7] Susanne Albers and Sonja Lauer. On list update with locality of reference. In *International Colloquium on Automata, Languages, and Programming*, pages 96–107. Springer, 2008.

[8] Susanne Albers and Michael Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica*, 21:312–329, 1998.

[9] Spyros Angelopoulos, Reza Dorrigiv, and Alejandro López-Ortiz. List update with

locality of reference. In *LATIN 2008: Theoretical Informatics*, pages 399–410, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[10] Shalom M Asbell and Eric Ruppert. A wait-free deque with polylogarithmic step complexity. In *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[11] Hagit Attiya, Arie Fouren, and Jeremy Ko. Lower bounds on the amortized time complexity of shared objects. *Theory of Computing Systems*, 68(5):1372–1426, 2024.

[12] Baruch Awerbuch, Shay Kutten, and David Peleg. Competitive distributed job scheduling. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 571–580, 1992.

[13] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. In *Proceedings of the twenty-fourth annual ACM Symposium on Theory of Computing*, pages 39–50, 1992.

[14] Jon L Bentley and Catherine C McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, 1985.

[15] Joan Boyar, Faith Ellen, and Kim S Larsen. The scheduler is very powerful in competitive analysis of distributed list accessing. *arXiv preprint arXiv:1807.06820*, 2018.

[16] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 329–342, 2014.

[17] Peter J Denning. Working set analytics. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2021.

[18] Reza Dorrigiv and Alejandro López-Ortiz. A new perspective on list update: Probabilistic locality and working set. In *International Workshop on Approximation and Online Algorithms*, pages 150–163. Springer, 2011.

[19] Ran El-Yaniv. *There are infinitely many competitive-optimal online list accessing algorithms*. Hebrew University of Jerusalem, 1996.

[20] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM Symposium on Principles of Distributed Computing*,

pages 50–59, 2004.

[21] Jing Fu and Kunlong Zhang. Practical concurrent self-organizing lists. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 384–391. IEEE, 2017.

[22] Joel Gibson and Vincent Gramoli. Why non-blocking operations should be selfish. In *International Symposium on Distributed Computing*, pages 200–214. Springer, 2015.

[23] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

[24] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 206–215, 2004.

[25] Maurice P Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the seventh annual ACM Symposium on Principles of Distributed Computing*, pages 276–290, 1988.

[26] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[27] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A–Z*, 2025.

[28] Kaiyi Ji, Guocong Quan, and Jian Tan. Asymptotic miss ratio of LRU caching with consistent hashing. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 450–458. IEEE, 2018.

[29] Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 251–266. Springer, 2013.

[30] Haim Kaplan and Robert E Tarjan. Persistent lists with catenation via recursive slowdown. In *Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing*, pages 93–102, 1995.

[31] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Search-*

ing. Addison-Wesley, 2nd edition, 1998.

[32] Arm Limited. *Arm® Architecture Reference Manual, for A-profile Architecture (Armv8 / Armv9)*, 2021. Chapter A2. URL: `https://developer.arm.com/documentation/ddi0487/latest/`.

[33] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.

[34] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[35] Hossein Naderibeni and Eric Ruppert. A wait-free queue with polylogarithmic step complexity. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 124–134, 2023.

[36] Ronald Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.

[37] Eric Ruppert. Analysing the average time complexity of lock-free data structures. In *Workshop on Complexity and Analysis of Distributed Algorithms*, 2016.

[38] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update rules. In *Proceedings of the sixteenth annual ACM Symposium on Theory of Computing*, pages 488–492, 1984.

[39] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[40] Longfei Tan, Zhao Han, Chunguang Chen, Yinghua He, and Kunlong Zhang. A non-blocking self-organizing linked list algorithm. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 71–76, 2012. `doi:10.1109/PDCAT.2012.27`.

[41] Robert Endre Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[42] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the*

fourteenth annual ACM Symposium on Principles of Distributed Computing, pages 214–222, 1995.

[43] Yuanhao Wei, Guy E Blelloch, Panagiota Fatourou, and Eric Ruppert. Practically and theoretically efficient garbage collection for multiversioning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 66–78, 2023.

Appendix

A A Worst-Case Execution of Fu et al.'s Lock-Free MTF List

We now construct an execution that demonstrates a worst-case scenario for Fu et al.'s [21] lock-free move-to-front (MTF) list. Intuitively, this sequence causes unbounded chains of invalid (`INV`) nodes to accumulate, and concurrent searches repeatedly undo one another's clean-up work, leading to unbounded total work.

Let a *p–Sequence* of a key x denote the following sequence of operations on Fu et al.'s list in a system with p processes:

1. INSERT(x) is ENLISTed onto the list by some process, such that the head points to a `DAT` node $found$ with $found.value = x$ and $found.result =$ `FND` (**Algorithm 4**, lines 1 - 4).

2. p concurrent CONTAINS(x) operations are invoked and their home nodes are ENLISTed onto the list (**Algorithm 2**, lines 1–2).

3. Let p' be the last process to ENLIST its home node $home'$. All other processes set their monitor to their home node (**Algorithm 5**, line 2) and pause.

4. p' traverses the list (**Algorithm 5**, lines 1–23) until reaching $found$ ($found$ is set to *monitor* at line 18, and the loop at lines 5–23 exits at the next execution of lines 6–7), collecting every node on its path (including $found$) into its friend list (at some iteration of line 17, by lines 8, 12, and 15–20).

5. p' sets $home'.result \leftarrow$ `FND` (**Algorithm 5**, lines 26-29), updates all its friends to `FND`, marks them `INV`, and terminates (**Algorithm 5**, lines 31 - 37).

6. All other processes wake, find $monitor.result =$ `FND` (**Algorithm 5**, lines 6–7), and terminate.

The result of this sequence is a chain of p nodes marked `INV` immediately following $home'$.

These nodes will not be removed by future searches for x, since $home'$ precedes them and $home'.result = \texttt{FND}$, which terminates the search loop.

A worst–case sequence can now be constructed as follows:

1. An INSERT(x) is ENLISTed so that *head* now points to a DAT node $found$ with $found.value = x$ and $found.result = \texttt{FND}$ (**Algorithm 4**, lines 1–4).

2. We create a chain of r consecutive INV nodes on a different key y by performing any number of **p–Sequences**. This chain begins at *head* and ends at $found$.

3. A process p_x performs CONTAINS(x) and ENLISTs its DAT node (**Algorithm 2**, lines 1–2). Another process p_z performs r concurrent CONTAINS operations on distinct keys $z_1, \ldots, z_r$ (all different from each other and from x and y). For the ith such operation ($1 \leq i \leq r$), p_x and p_z execute in the following order:

   (a) p_x reaches a node *curr* in the chain of INV nodes (**Algorithm 5**, lines 3–4), reads *curr.next* in preparation for deletion (lines 8–9), and pauses.

   (b) p_z continues, reaches *curr*, and removes a chain of $r - i + 1$ INV nodes from *curr* up to $found$ (**Algorithm 5**, lines 8–11, repeated $r - i + 1$ times), then finishes.

   (c) p_x resumes, swings *pred.next* to *curr.next*, undoing $r - i$ of p_z's deletions (at **Algorithm 5**, line 10). It then moves to *curr.next* (line 11): if this node is still an INV node, the process repeats from (a).

In this execution, the r searches performed by p_z each clean a progressively shorter chain of invalid nodes. The total work they perform is

$$r + (r - 1) + (r - 2) + \cdots + 1 = \Omega(r^2).$$

Constructing a **p–Sequence** of size r takes $O(r)$ operations, and the r searches by p_z also contribute $O(r)$ operations. The initial INSERT(x) and the CONTAINS(x) by p_x add only one operation each. Thus the total number of operations in this sequence is $O(r)$, while the list performs $\Omega(r^2)$ steps.

Therefore, the amortized step complexity of Fu et al.'s list is proportional to the total number of operations performed on the list.

B A Worst–Case Execution of Boyar et al.'s Restricted Lock–Free MTF List

Lemma 74. *The amortized cost of a* SEARCH(k) *operation op on Boyar et al.'s [15] restricted MTF list is:*

$$\Omega\big(|w(op)| + p\big).$$

Proof. We consider a sequential execution of a worst–case sequence of repeated accesses to the keys

$$(k_n, k_{n-1}, \ldots, k_1,\ k_n, k_{n-1}, \ldots, k_1,\ \ldots),$$

where k_i denotes the key initially at position i in the list.

Let $\text{rank}_j(k_i)$ denote the position of the key k_i in the list before the j–th operation op_j on k_i in the execution. During each access to a key, the algorithm moves the key to the front of the list (lines 39–52, invoked from line 20) and never reorders any other nodes.

The *concurrent working set size* of an operation *op* on the key k linearized at step ℓ, denoted $|w(op)|$, is the number of distinct keys in the set at ℓ with an operation that was linearized between ℓ_k, the most recent linearization point among completed operations on k, and ℓ (see section 5.1). Formally, these are all keys k' in the set at ℓ that were accessed by operations with a linearization point ℓ' such that $\ell_k \preceq \ell' \prec \ell$.

Because our worst-case execution is sequential, operations are linearized in their real-time order. Furthermore, since the set of keys in Boyar et al.'s list is fixed, the concurrent working set size of an operation op_j on key k_i in our sequential execution equals the number of distinct keys k' accessed after the previous operation op_{j-1} on k_i and before op_j, plus one to include the access to k_i by op_j. Each such key k' is moved to the front of the list during its access, while no other keys are reordered. Thus, immediately before op_j, every disinct key k' precedes k_i in the list, implying

$$\text{rank}_j(k_i) = (\#\text{ of distinct keys accessed after } op_{j-1} \text{ and before } op_j) + 1 \ \geq\ |w(op_j)|.$$

Consequently, the traversal cost at lines 11–31 is $\Omega(w(op_j))$ for the j–th access of the key k_i, because it stops when it reaches the node containing k_i (see line 12 and lines 17–23). The traversal does not terminate early via the return statement at line 29, because this requires a concurrent process to write the current operation's result (see the check at line 27), and

our execution is sequential.

When the node containing k_i is found by op_j (the CAS at line 18 always succeeds in this contention–free execution), the call to MOVE–TO–FRONT(g') at line 20 scans all p entries of the process announcement array $A[\cdot]$ (lines 44–46). Even without contention, this scan performs $\Omega(p)$ `read` steps. Hence, the operation op_j on k_i incurs an additional $\Omega(p)$ cost independent of $|w(op_j)|$.

For any prefix of m SEARCH operations in our sequence, every lookup has the same working set size of $n-1$. Hence, the total cost of those m SEARCH operations is:

$$\sum_{op} \Omega(|w(op)|) + \Omega(mp) = \sum_{i=1}^{m} \Omega(n-1) + \Omega(mp) = \Omega(m(n-1)) + \Omega(mp) = \Omega(m(n-1+p)).$$

Using the aggregate method, the amortized cost of a SEARCH(k_i) operation op linearized at the step ℓ is

$$\Omega(n-1+p) = \Omega(|w(op)| + p).$$

□