Abstract

Concurrent data structures or CDS such as concurrent stacks, queues, sets etc. have become very popular in the past few years partly due to the rise of multi-core systems. Such concurrent CDSs offer great performance benefits over their sequential counterparts.

But one of the greatest challenges with CDSs has been developing correct structures and then proving correctness of these structures. We believe that techniques that help prove correctness of these CDSs can also guide in developing new CDSs.

An intuitive & popular techniques to prove correctness of CDSs is using Linearization Points or LPs. A LP is an (atomic) event in the execution interval of each method such that the execution of the entire method seems to have taken place in the instant of that event.

One of the main challenges with the LP based approach is to identify the correct LPs of a CDS. Identifying the correct LPs can be deceptively wrong in many cases. In fact in many cases, the LP identified or even worse the CDS itself could be wrong.

To address these issues, several automatic tools for verifying linearizability have been developed. But we believe that these tools don’t provide insight to a programmer to develop the correct concurrent programs or identify the LPs.

Considering the complexity of developing a CDS and verifying its correctness, we address the most basic problem of this domain in this paper: given the set of LPs of a CDS, how to show its correctness? We assume that we are given a CDS and its LPs. We have developed a hand-crafted technique of proving correctness of the CDSs by validating its LPs. As observed earlier, identifying the correct LPs is very tricky and erroneous. But since our technique is hand-crafted, we believe that the process of proving correctness might provide insight to identify the correct LPs, if the currently chosen LP is incorrect. We also believe that this technique might also offer the programmer some insight to develop more efficient variants of the CDS.

We believe that our proof technique can be applied to prove the correctness of several commonly used CDSs developed in literature such as Lock-free Linked based Sets Skiplists etc. Our technique will also show correctness of CDSs in which the LPs of method might lie outside the methods such as lazy-list based set. To show the efficacy of this technique, we show the correctness of lazy-list and hoh-locking-list based set.

Keywords: linearizability; concurrent data structure; linearization points; correctness;

1 Introduction

Concurrent data structures or CDS such as concurrent stacks, queues, sets etc. have become very popular in the past few years partly due to the rise of multi-core systems. To increase parallelism many of these CDSs have been designed in a way to reduce blocking of the concurrently executing threads due to synchronization by locks. They either use locks in highly fine-grained manner or use no locks at all.
Such concurrent CDSs offer great performance benefits over their sequential counterparts. But one of the greatest challenges with CDSs is developing correct structures and then proving their correctness either through automatic verification or through hand-written proofs [4]. We believe that the techniques which help prove correctness of CDSs can also guide in developing new CDSs.

To prove a concurrent data structure to be correct, linearizability proposed by Herlihy & Wing [9] is the standard correctness criterion used. They consider a history generated by the CDS which is collection of method invocation and response events. Each invocation of a method call has a subsequent response which can be interleaved with invocation, responses from other concurrent methods. A history is linearizable if (1) The invocation and response events can be reordered to get a valid sequential history. (2) The generated sequential history satisfies the object’s sequential specification. (3) If a response event precedes an invocation event in the original history, then this should be preserved in the sequential reordering.

A concurrent object is linearizable if each of their histories is linearizable. Linearizability ensures that every concurrent execution simulates the behavior of some sequential execution while not actually executing sequentially and hence leveraging on the performance.

One of the intuitive techniques to prove correctness of CDSs is using Linearization Points or LPs. A LP is an (atomic) event in the execution interval of each method such that the execution of the entire method seems to have taken place in the instant of that event.

Several techniques have been proposed for proving linearizability: both hand-written based and through automatic verification. Many of these techniques consider lazy linked-list based concurrent set implementation, denoted as lazy-list, proposed by Heller at al [6]. This is one of the popular CDSs used for proving correctness due to the intricacies of LPs of its methods in their execution. The LP of an unsuccessful contains method can sometimes be outside the code of its methods and depend on an concurrently executing add method. This is illustrated in Figure 4 of SubSection 4.1.

Vafeiadis et al. [16] hand-crafted one of the earliest proofs of linearizability for lazy-list using the rely-guarantee approach [10] which can be generalized to other CDSs as well. O’Hearn et al. [14] have developed a generic methodology for linearizability by identifying new property known as Hindsight lemma. Their technique is non-constructive in nature. Both these techniques don’t depend on the notion of LPs.

Recently Lev-Ari et al. [11,12] proposed a constructive methodology for proving correctness of CDSs. They have developed a very interesting notion of base-points and base-conditions to prove linearizability. Their methodology manually identifies the base conditions, commuting steps, and base point preserving steps and gives a roadmap for proving correctness by writing semi-formal proofs. Their seminal technique, does not depend on the notion of LPs, can help practitioners and researchers from other fields to develop correct CDSs.

In spite of several such techniques having been proposed for proving linearizability, LPs continue to remain most popular guiding tool for developing efficient CDSs and illustrating correctness of these CDSs among practitioners. LPs are popular since they seem intuitive and more importantly are constructive in nature. In fact, we believe using the notion of LPs, new CDS can be designed as well.

But one of the main challenges with the LP based approach is to identify the correct LPs of a CDS. Identifying the correct LPs can be deceptively wrong in many cases. For instance, it is not obvious to a novice developer that the LP of an unsuccessful contains method of lazy-list could be outside the contains method. In fact in many cases, the LP identified or even worse the CDS could be wrong.

The problem of proving correctness of CDS using LPs has been quite well explored in the verification community in the past few years. Several efficient automatic proving tools and techniques have been developed [1,3,13,15,19,20] to address this issue. In fact, many of these tools can also show correctness even without the information of LPs. But very little can be
gleaned from these techniques to identify the correct LPs of a CDS by a programmer. Nor do they provide any insight to a programmer to develop new CDSs which are correct. The objective of the most of these techniques has been to efficiently automate proving correctness of already developed CDSs.

Considering the complexity of developing a CDS and verifying its correctness, we address the most basic problem of this domain in this paper: given the set of LPs of a CDS, how to show its correctness? We assume that we are given a CDS and its LPs. We have developed a hand-crafted technique of proving correctness of the CDS by validating it LPs. We believe that our technique can be applied to prove the correctness of several commonly used CDSs developed in literature such as Lock-free Linked based Sets [17], hoh-locking-list [2, 8], lazy-list [6, 8], Skiplists [18] etc. Our technique will also work for CDSs in which the LPs of a method might lie outside the method such as lazy-list. To show the efficacy of this technique, we show the correctness of lazy-list and hand-over-hand locking list (hoh-locking-list) [2, 8].

As observed earlier, identifying the correct LPs is very tricky and erroneous. But since our technique is hand-crafted, we believe that the process of proving correctness might provide insight to identify the correct LPs, if the currently chosen LP is incorrect. We also believe that this technique might also offer the programmer some insight to develop more efficient variants of the CDS.

Our technique is inspired from the notion of rely-guarantee approach [10] and Vafeiadis et al. [16]. For the technique to work, we make some assumptions about the CDS and its LPs. We describe the main idea here and the details in the later sections.

**Main Idea: Proving Correctness of LPs.** In this technique, we consider executions corresponding to the histories. For a history \( H \), an execution \( E^H \) is a totally ordered sequence of atomic events which are executed by the threads invoking the methods of the history. Thus an execution starts from an initial global state and then goes from one global state to the other as it executes atomic events.

With each global state, we associate the notion of abstract data-structure or AbDS. This represent the state of the CDS if it had executed sequentially. Vafeiadis et al. [16] denote it as abstract set or AbS in the context of the lazy-list.

We assume that each method of the CDS has a unique atomic event as the LP within its execution. Further, we assume that only a (subset) of LP events can change the AbS. We have formalized these assumptions in SubSection 3.1.

With these assumptions in place, to show the correctness of a history \( H \), we first construct a sequential history \( CS(H) \): we order all the methods of \( H \) by their LPs (which all are atomic and hence totally ordered). Then based on this method ordering, we invoke the methods (using a single thread) with the same parameters on the CDS sequentially. The resulting history generated is sequential. The details of this construction is described in SubSection 3.3.

Since \( CS(H) \) is generated sequentially, it can be seen that it satisfies the sequential-specification of the CDS. All the method invocations of \( CS(H) \) respect the method ordering of \( H \). If we can show that all the response events in \( H \) and \( CS(H) \) are the same then \( H \) is linearizable.

The proof of this equivalence naturally depends on the properties of the CDS being considered. We have identified a lemma (Lemma 8 of SubSection 3.4) as a part of our proof technique, which if shown to be true for all the methods of the CDS, implies linearizability of the CDS. In this lemma, we consider the pre-state of the LP of a method \( m_i \) in a history \( H \). As the name suggests, pre-state is the global state of the CDS just before the LP event. This lemma requires that the AbDS in the pre-state to be the result of some sequential execution of the methods of the CDS. Similarly, the AbDS in the post-state of the LP must be as a result of some sequential execution the methods with \( m_i \) being the final method in the sequence. We show that if the CDS ensures these conditions then it is linearizable.

The lemma that we have identified is generic. We show that any CDS for which this lemma is true and satisfies our assumptions on the LPs, is linearizable. Thus, we would like to view
we use delimiters to differentiate them. In most cases, we ignore these invocation and response parameters unless they are required for the context and denote the method as \( m \). In such a case, we simply denote \( m, \text{inv}, m, \text{resp} \) as the inv and resp events.

Global States, Execution and Histories. We define the global state or state of the system as the collection of local and shared variables across all the threads in the system. The system starts with an initial global state. Each event changes possibly the global state of the system leading to a new global state. The events read, write, rmw on shared/local memory objects change the global state. The inv \& resp events on higher level shared-memory objects do not change the contents of the global state. Although we denote the resulting state with a new label \( H \) is the total order among these events. A history corresponding to an execution consists only of method inv \& resp events (in other words, a history views the methods as black boxes without going inside the internals). Similar to an execution, a history \( H \) can be formally denoted as \( \langle \text{evts}, <_H \rangle \) where \( \text{evts} \) are of type inv \& resp and \( <_H \) defines a total order among these events. With this definition, it can be seen that a history

\[
\langle \text{evts}, <_H \rangle
\]

Roadmap. In Section 2, we describe the system model. In Section 3, we describe the proof technique. In Section 4, we illustrate this technique by giving outline of the proof for lazy-list and hoh-locking-list. Finally, we conclude in Section 5.

2 System Model & Preliminaries

In this paper, we assume that our system consists of finite set of \( p \) processors, accessed by a finite set of \( n \) threads that run in a completely asynchronous manner and communicate using shared objects. The threads communicate with each other by invoking higher-level methods on the shared objects and obtaining the corresponding responses. Consequently, we make no assumption about the relative speeds of the threads. We also assume that none of these processors and threads fail. We refer to a shared objects as a concurrent data-structure or CDS.

Events \& Methods. We assume that the threads execute atomic events. Similar to Lev-Ari et. al.’s work, \([11,12]\) we assume that these events by different threads are (1) atomic read, write on shared/local memory objects; (2) atomic read-modify-write or rmw operations such compare \& swap etc. on shared memory objects (3) method invocation or inv event \& response or resp event on CDSs.

A thread executing a method \( m_i \), starts with the inv event, say \( \text{inv}_i \), executes the events in the \( m_i \) until the final resp event \( \text{resp}_i \). The resp event \( \text{resp}_i \) of \( m_i \) is said to match the inv event \( \text{inv}_i \). On the other hand, if the inv event \( \text{inv}_i \) does not have a resp event \( \text{resp}_i \) in the execution, then we say that both the inv event \( \text{inv}_i \) and the method \( m_i \) are pending.

The method inv \& resp events are typically associated with invocation and response parameters. The invocation parameters are passed as input while response parameters are obtained as output to and from the CDS respectively. For instance, the invocation event of the enqueue method on a queue object \( Q \) is denoted as \( \text{inv}(Q, \text{enq}(v)) \) while the resp event of a dequeue method can be denoted as \( \text{resp}(Q, \text{deq}(v)) \). We combine the inv and resp events to represent a method as follows: \( m_i (\text{inv-params}, \text{resp-params}) \) where \( \text{inv}(m_i (\text{inv-params})) \) and \( \text{resp}(m_i (\text{resp-params})) \) represent the inv, resp events respectively. For instance, we represent enqueue as \( \text{enq}(v, ok) \), or a successful add to a set as \( \text{add}(k, T) \). If there are multiple invocation or response parameters, we use delimiters to differentiate them. In most cases, we ignore these invocation and response parameters unless they are required for the context and denote the method as \( m_i \). In such a case, we simply denote \( m_i, \text{inv}, m_i, \text{resp} \) as the inv and resp events.
uniquely characterizes an execution and vice-versa. Thus we use these terms interchangeably in our discussion. For a history \( H \), we denote the corresponding execution as \( E^H \).

We denote the set of methods invoked by threads in a history \( H \) (and the corresponding execution \( E^H \)) by \( H.mths \) (or \( E^H.mths \)). Similarly, if a method \( m_x \) is invoked by a thread in a history \( H \) (\( E^H \)), we refer to it as \( H.m_x \) \((E^H.m_x)\). Although all the events of an execution are totally ordered in \( E^H \), the methods are only partially ordered. We say that a method \( m_x \) is ordered before method \( m_y \) in real-time if the resp event of \( m_x \) precedes the invocation event of \( m_y \), i.e. \((m_x.rsp <_H m_y.inv)\). We denote the set of all real-time orders between in the methods of \( H \) by \( \prec_H \).

Next, we relate executions (histories) with global states. An execution takes the system through a series of global states with each event of the execution stating from the initial state takes the global state from one to the next. We associate the state of an execution (or history) to be global state after the last event of the execution. We denote this final global state \( S \) of an execution \( E \) as \( S = E.state \) (or \( H.state \)). We refer to the set of all the global states that a system goes through in the course of an execution as \( E.allStates \) (or \( H.allStates \)). It can be seen that for \( E, E.state \in E.allStates \). Figure 1 shows a concurrent execution \( E^H \) and its corresponding history \( H \). In the figure, the curved line represents an event and the vertical line is a state. The open(][ & close(]) square brackets simply demarcate the methods of a thread and have no specific meaning in the figure.

Given an event \( e \) of an execution \( E \), we denote global state just before the \( e \) as the pre-state of \( e \) and denote it as \( PreE[e] \). Similarly, we denote the state immediately after \( e \) as the post-state of \( e \) or \( PostE[e] \). Thus if an event \( e \) is in \( E.evts \) then both \( PreE[e] \) and \( PostE[e] \) are in \( E.allStates \).

![Figure 1](image-url)

The notion of pre & post states can be extended to methods as well. We denote the pre-state of a method \( m \) or \( PreM[m] \) as the global state just before the invocation event of \( m \) whereas the post-state of \( m \) or \( PreM[m] \) as the global state just after the return event of \( m \). Figure 2 illustrates the global states immediately before and after \( m_i.LP \) which are denoted as \( PreE[E^H.m_i.LP] \) and \( PostE[E^H.m_i.LP] \) respectively in the execution \( E^H \).

**Notations on Histories.** We now define a few notations on histories which can be extended to the corresponding executions. We say two histories \( H1 \) and \( H2 \) are *equivalent* if the set of events in \( H1 \) are the same as \( H2 \), i.e., \( H1.evts = H2.evts \) and denote it as \( H1 \approx H2 \). We say history \( H1 \) is a *sub-history* of \( H2 \) if all the events of \( H1 \) are also in \( H2 \) in the same order, i.e., \((H1.evts \subseteq H2.evts) \land (<_H1 \subseteq <_H2))\). Let a thread \( T_i \) invoke some methods on a few CDSs (shared memory objects) in a history \( H \) and \( d \) be a CDS whose methods have been invoked by
Figure 2: Figure (a) illustrates an example of a concurrent execution $E^H$. Then, $m_i LP$ is the LP event of the method $m_i$. The global state immediately after this event is represented as Post-state of ($E^H . m_i LP$). Figure (b) represents sequential execution $E^S$ corresponding to (a) with post-state of method $m_i$ as the state after its resp event.

threads in $H$. Using the notation of [9], we denote $H | T_i$ to be the sub-history of all the events of $T_i$ in $H$. Similarly, we denote $H | d$ to be the sub-history of all the events involving $d$.

We assume that a history $H$ as well-formed if a thread $T_i$ does not invoke the next method on a CDS until it obtains the matching response for the previous invocation. We assume that all the executions & histories considered in this paper are well-formed. Note that since an execution is well-formed, there can be at most only one pending invocation for each thread.

We say the history $H$ is complete if for every inv event there is a matching resp event, i.e., there are no pending methods in $H$. The history $H$ is said to be sequential if every inv event, except possibly the last, is immediately followed by the matching resp event. In other words, all the methods of $H$ are totally ordered by real-time and hence $\prec_H$ is a total order. Note that a complete history is not sequential and the vice-versa. It can be seen that in a well-formed history $H$, for every thread $T_i$, we have that $H | T_i$ is sequential. Figure 3 shows a the execution of a sequential history $S$.

Figure 3: An illustration of a sequential execution $E^S$.

Sequential Specification. We next discuss about sequential-specification [9] of CDSs. The sequential-specification of a CDS $d$ is defined as the set of (all possible) sequential histories involving the methods of $d$. Since all the histories in the sequential-specification of $d$ are sequential, this set captures the behavior of $d$ under sequential execution which is believed to be correct. A sequential history $S$ is said to be legal if for every CDS $d$ whose method is invoked in $S$, $S | d$ is in the sequential-specification of $d$.

Safety: A safety property is defined over histories (and the corresponding executions) of shared objects and generally states which executions of the shared objects are acceptable to
any application. The safety property that we consider is linearizability [9]. A history \( H \) is said to be linearizable if (1) there exists a completion \( H' \) of \( H \) in which some pending inv events are completed with a matching response and some other pending inv events are discarded; (2) there exists a sequential history \( S \) such that \( S \) is equivalent to \( H' \), i.e., \( H' \approx S \); (3) \( S \) respects the real-time order of \( H \), i.e., \( \prec_H \subseteq \prec_S \); (4) \( S \) is legal. Another way to say that history \( H \) is linearizable if it is possible to assign an atomic event as a linearization point or LP inside the execution interval of each method such that the result of each of these methods is the same as it would be in a sequential history \( S \) in which the methods are ordered by their LPs [8]. In this document, we show how to prove the correctness of LPs of the various methods of a data-structure.

3 Generic Proof Technique

In this section, we develop a generic framework for proving the correctness of a CDS based on LP events of the methods. Our technique of proving is based on hand-crafting and is not automated. We assume that the developer of the CDS has also identified the LPs of the methods. We assume that the LPs satisfy a few properties that we outline in the course of this section.

In Section 4, we illustrate this technique by showing the correctness at a high level of two structures (1) lazy-list based concurrent set implementation [6] denoted as lazy-list in SubSection 4.1 and hand-over-hand locking based concurrent set implementation denoted as hoh-locking-list in SubSection 4.2.

3.1 Linearization Points Details

Intuitively, LP is an (atomic) event in the execution interval of each method such that the execution of the entire method seems to have taken place in the instant of that event. As discussed in Section 2, the LP of each method is such that the result of execution of each of these methods is the same as it would be in a sequential history \( S \) in which the methods are ordered by their LPs [8].

Given, the set of LPs of all the methods of a concurrent data-structure, we show how the correctness of these LPs can be verified. We show this by proving the correctness of the CDS assuming that it is linearizable and the LPs are chosen correctly in the first place.

Consider a method \( m_i \)(inv-params \( \uparrow \), rsp-params \( \downarrow \)) of a CDS \( d \). Then the precise LP of \( m_i \) depends on \( \text{rsp-params} \downarrow \). For instance in the lazy-list [6], the LP of \( \text{contains}(k, \text{true}) \) method is different from \( \text{contains}(k, \text{false}) \). Furthermore, the LP of a method also depends on the execution. For instance, considering the contains method of the lazy-list again, the LP of \( \text{contains}(k, \text{false}) \) depends on whether there is an \( \text{add}(k, \text{true}) \) method concurrently executing with it or not. The details of the LPs of the lazy-list are described in the original paper by Heller et. al [6] and also in in SubSection 4.1. Another important point to consider is that the method \( m_i \) in an execution can go through several possible LP events before returning a value. We then assume that the final LP event executed decides the return parameters of the method. Let us illustrate this again with the case of contains method of the lazy-list CDS. Consider an execution \( E^H \) having the contains method \( m_i \) concurrently executing with \( \text{add}(k, \text{true}) \) method. In this case, the LP of \( m_i \) depends on the LP of \( \text{add}(k, \text{true}) \) if \( m_i \) returns false. Suppose \( m_i \) executes the event, say \( e_x \), that corresponds to the LP of \( \text{contains}(k, \text{false}) \). Then later, the contains method also executes the event, say \( e_y \) corresponding to the LP of \( \text{contains}(k, \text{true}) \) which is reading of a shared memory variable \( n.\text{marked} \) of node \( n \). If \( n.\text{marked} \) is false then the contains method \( m_i \) returns true and \( e_y \) is the LP. Otherwise, \( m_i \) returns false and \( e_x \) is LP. Thus \( m_i \) executes both \( e_x \) and \( e_y \). Either of them can be the LP depending on the system state.

We denote the LP event of \( m_i \) in a history \( H \) as \( E^H.m_i.(\text{inv-params} \uparrow, \text{rsp-params} \downarrow).LP \) or \( E^H.m_i.LP \) (depending on the context). The global state in the execution \( E^H \) immediately
before and after $m_i.LP$ is denoted as $PreE[E^H.m_i.LP]$ and $PostE[E^H.m_i.LP]$ respectively.

3.2 Abstract Data-Structure & LP Assumptions

To prove correctness of a CDS $d$, we associate with it an abstract data-structure or $AbDS$. The $AbDS$ captures the behavior of CDS if it had executed sequentially. Since sequential executions are assumed to be correct, it is assumed that $AbDS$ is correct. In fact, the sequential-specification of $d$ can be defined using $AbDS$ since in any global state the internal state of $AbDS$ is the result of sequential execution. Thus, we can say that CDS $d$ refines $AbDS$ [5].

The exact definition of $AbDS$ depends on the actual CDS being implemented. In the case of lazy-list, $AbDS$ is the set of unmarked nodes reachable from the head while the CDS is the set of all the nodes in the system. Vafeiadis et. al [16] while proving the correctness of the lazy-list refer to $AbDS$ as abstract set or $AbS$. In the case of hoh-locking-list, $AbDS$ is the set nodes reachable from the head while the CDS is the set of all nodes similar to lazy-list. Normally the CDS maintains more information (such as sentinel nodes) than $AbDS$ to implement the desired behavior. For a given global state $S$, we use the notation $S.AbDS$ and $S.CDS$ to refer to the contents of these structures in $S$.

Now we state a few assumptions about the CDS and its LPs that we require for our proof technique to work.

**Assumption 1** In any sequential execution, any method of the CDS can be invoked in any global state and yet get a response.

Intuitively, Assumption 1 states that if threads execute the methods of the CDS sequentially then every method invocation will have a matching response. No method blocks in the sequential execution. Such methods are called as total [8, Chap 10].

**Assumption 2** Every sequential history $S$ generated by the CDS is legal.

Assumption 2 says that sequential execution of the CDS is correct and does not result in any errors. We next make the following assumptions based on the LPs.

**Assumption 3** Consider a method $m_i$(inv-params ↑, rsp-params ↓) of the CDS in a concurrent execution $E^H$. Then $m_i$ has a unique LP which is an atomic event within the inv and resp events of $m_i$ in $E^H$. The LP event can be identified based on the inv-params ↑, rsp-params ↓ and the execution $E^H$.

**Assumption 4** Consider an execution $E^H$ of a CDS $d$. Then only the LP events of the methods can change the contents $AbDS$ of the given CDS $d$.

Assumptions 3 & 4 when combined imply that there is only event in each method that can change the $AbDS$. As per Assumption 4, only the LPs can change the contents of $AbDS$. But this does not imply that all the LPs change the $AbDS$. It implies that if an event changes $AbDS$ then it must be a LP event. For instance in the case of lazy-list, the LPs of $add(k, false)$, $remove(k, false)$ and the LPs of the contains methods do not change the $AbDS$.

We believe that the assumptions made by us are generic and are satisfied by many of the commonly used CDSs such as Lock-free Linked based Sets [17], hoh-locking-list [2, 8], lazy-list [6,8], Skiplists [18] etc. In fact, these assumptions are similar in spirit to the definition of Valid LP by Zhu et al [20].

It can be seen that the Assumptions 3 & 4 characterize the LP events. Any event that does not satisfy these assumptions is most likely not a LP (please refer to the discussion section Section 5 more on this).
3.3 Constructing Sequential History

To prove linearizability of a CDS \(d\) which satisfies the Assumptions 1, 2, 3, 4 we have to show that every history generated by \(d\) is linearizable. To show this, we consider an arbitrary history \(H\) generated by \(d\). First we complete \(H\), to form \(\overline{H}\) if \(H\) is incomplete. We then construct a sequential history denoted as \(CS(H)\) (constructed sequential history). \(H\) is linearizable if (1) \(CS(H)\) is equivalent to a completion of \(H\); (2) \(CS(H)\) respects the real-time order of \(H\) and (3) \(CS(H)\) is legal.

We now show how to construct \(\overline{H}\), \(CS(H)\). We then analyze some properties of \(CS(H)\).

**Completion of \(H\).** Suppose \(H\) is not complete. This implies \(H\) contains some incomplete methods. Note that since these methods are incomplete, they could have executed multiple possible LP events. Based on these LP events, we must complete them by adding appropriate resp event or ignore them. We construct the completion \(\overline{H}\) and \(E^{\overline{H}}\) as follows:

1. Among all the incomplete methods of \(E^{H}\) we ignore those methods, say \(m_i\), such that: (a) \(m_i\) did not execute a single LP event in \(E^{H}\); (b) the LP event executed by \(m_i\) did not change the AbDS.
2. The remaining incomplete methods must have executed an LP event in \(E^{H}\) which changed the AbDS. Note from Assumptions 3 & 4, we get that each method has only one event which can change the AbDS and that event is the LP event. We build an ordered set consisting of all these incomplete methods which is denoted as partial-set. The methods in partial-set are ordered by their LPs.
3. To build \(\overline{H}\), for each incomplete method \(m_i\) in partial-set considered in order, we append the appropriate resp event to \(H\) based on the LP event of \(m_i\) executed. Since the methods in partial-set are ordered by their LP events, the appended resp events are also ordered by their LP events. Here, we assumed that once a method executes a LP event that changes the AbDS, its resp event can be determined.
4. To construct \(E^{\overline{H}}\), for each incomplete method \(m_i\) in partial-set considered in order, we sequentially append all the remaining events of \(m_i\) (after its LP) to \(E^{H}\). All the appended events are ordered by the LPs of their respective methods.

From this construction, one can see that if \(\overline{H}\) is linearizable then \(H\) is also linearizable. Formally, \((\overline{H}\) is linearizable) \(\implies\) (\(H\) is linearizable)).

For simplicity of presentation, we assume that all the concurrent histories & executions that we consider in the rest of this document are complete unless stated otherwise. Given any history that is incomplete, we can complete it by the transformation mentioned here. Next, we show how to construct a \(CS(H)\) for a complete history \(H\).

**Construction of \(CS(H)\).** Given a complete history \(H\) consisting of method inv & rsp events of a CDS \(d\), we construct \(CS(H)\) as follows: We have a single (hypothetical) thread invoking each method of \(H\) (with the same parameters) on \(d\) in the order of their inv events. Only after getting the response for the currently invoked method, the thread invokes the next method. From Assumption 1, which says that the methods are total, we get that for every method invocation \(d\) will issue a response.

Thus we can see that the output of these method invocations is the sequential history \(CS(H)\). From Assumption 2, we get that \(CS(H)\) is legal. The histories \(H\) and \(CS(H)\) have the same inv events for all the methods. But, the resp events could possibly be different. Hence, they may not be equivalent to each other unless we prove otherwise.

In the sequential history \(CS(H)\) all the methods are totally ordered. So we can enumerate all its methods as: \(m_1\text{(inv-params, rsp-params)} m_2\text{(inv-params, rsp-params)} \ldots m_n\text{(inv-params, rsp-params)}\). On the other hand, the methods in a concurrent history \(H\) are not ordered. From
our model, we have that all the events of the execution $E^H$ are ordered. In Assumption 3, we have assumed that each complete method has a unique LP event which is atomic. All the methods of $H$ and $E^H$ are complete. Hence, we can order the LPs of all the methods in $E^H$. Based on LP ordering, we can enumerate the corresponding methods of the concurrent history $H$ as $m_1(\text{inv-params}, \text{rsp-params}), m_2(\text{inv-params}, \text{rsp-params}), \ldots, m_n(\text{inv-params}, \text{rsp-params})$. Note that this enumeration has nothing to do with the ordering of the inv and resp events of the methods in $H$.

Thus from the construction of $CS(H)$, we get that for any method $m_i$, $H.\text{inv}(m_i(\text{inv-params})) = CS(H).\text{inv}(m_i(\text{inv-params}))$ but the same need not be true for the resp events.

For showing $H$ to be linearizable, we further need to show $CS(H)$ is equivalent to $H$ and respects the real-time order $H$. Now, suppose $CS(H)$ is equivalent to $H$. Then from the construction of $CS(H)$, it can be seen that $CS(H)$ satisfies the real-time order of $H$. The following lemma proves it.

**Lemma 5** Consider a history $H$ be a history generated by a CDS $d$. Let $CS(H)$ be the constructed sequential history. If $H$ is equivalent to $CS(H)$ then $CS(H)$ respects the real-time order of $H$. Formally, $(\forall H : (H \approx CS(H))) \implies (\langle r^t_1 \lessdot r^t \rangle)$.

**Proof.** This lemma follows from the construction of $CS(H)$. Here we are given that for every method $m_i$, $H.m_i.\text{inv} = CS(H).m_i.\text{inv}$ and $H.m_i.\text{rsp} = CS(H).m_i.\text{rsp}$.

Now suppose two methods, $m_i, m_j$ are ordered by real-time. This implies that $m_i.\text{rsp} <_H m_j.\text{inv}$. Hence, we get that $m_i.\text{inv} <_H m_i.\text{rsp} <_H m_j.\text{inv}$ which means that $m_i$ is invoked before $m_j$ in $H$. Thus, from the construction of $CS(H)$, we get that $m_i$ is invoked before $m_j$ in $CS(H)$ as well. Since $CS(H)$ is sequential, we get that $m_i.\text{rsp} <_{CS(H)} m_j.\text{inv}$. Thus $CS(H)$ respects the real-time order of $H$. □

Now it remains to prove that $H$ is equivalent to $CS(H)$ for showing linearizability of $H$. But this proof depends on the properties of the CDS $d$ being implemented and is specific to $d$. Now we give a generic outline for proving the equivalence between $H$ and $CS(H)$ for any CDS. As mentioned earlier, later in Section 4, we illustrate this technique by showing at a high level the correctness of lazy-list & hoh-locking-list.

### 3.4 Details of the Generic Proof Technique

As discussed above, to prove the correctness of a concurrent (& complete) history $H$ representing an execution of a CDS $d$, it is sufficient to show that $H$ is equivalent to $CS(H)$. To show this, we have developed a generic proof technique.

It can be obviously seen that to prove the correctness, this proof depends on the properties of the CDS $d$ being considered. To this end, we have identified a CDS-specific lemma which captures the properties required of the CDS $d$. Proving this CDS-specific lemma for each CDS would imply equivalence of $H$ between $CS(H)$ and hence linearizability of the CDS.

In the following lemmas, we assume that all the histories and execution considered here are generated from the CDS $d$. The CDS $d$ satisfies the Assumptions 1, 2, 3, 4. Since we are only considering CDS $d$, we refer to its abstract data-structure as $AbDS$ and refer to its state in a global state $S$ as $S.AbDS$.

In the following lemmas, as described in SubSection 3.3, we enumerate all the methods of a sequential history $S$ as: $m_1, m_2 ... m_n$. We enumerate all the all the methods of the concurrent history $H$ as $m_1, m_2 ... m_n$ based on the order of their LPs.

**Lemma 6** The $AbDS$ of $d$ in the global state after the resp event of a method $m_x$ is the same as the $AbDS$ before the inv event of the consecutive method $m_{x+1}$ in an execution $E^S$ of a sequential history $S$. Formally, $(\forall m_x \in E^S.mths : PostM[E^S.m_x].AbDS = PreM[E^S.m_{x+1}].AbDS)$.
Proof. From the definition of Sequential Execution.

Lemma 7 Consider a concurrent execution $E^H$ of the methods of $d$. Then, the contents of AbDS in the post-state of LP of $m_y$ is the same as the AbDS in pre-state of the next LP belonging to $m_{x+1}$. Formally, $(\forall m_x \in E^H \text{mths} : \text{Post}E[E^H.m_x.LP].\text{AbDS} = \text{Pre}E[E^H.m_{x+1}.LP].\text{AbDS})$.

Proof. From the assumption 4, we know that any event between the post-state of $m_y.LP$ and the pre-state of $m_{x+1}.LP$ will not change the AbDS. Hence we get this lemma.

Now, we describe a lemma which is CDS specific. This lemma can be considered to be analogous to an abstract class in C++. Based on the CDS involved, this has to be appropriately proved.

CDS-Specific Lemma 8 Consider a concurrent history $H$ and a sequential history $S$. Let $m_x,m_y$ be methods in $H$ and $S$ respectively. Suppose the following are true (1) The AbDS in the pre-state of $m_x$’s LP in $H$ is the same as the AbDS in the pre-state of $m_y$ in $S$; (2) The inv events of $m_x$ and $m_y$ are the same. Then (1) the resp event of $m_x$ in $H$ must be same as resp event of $m_y$ in $S$; (2) The AbDS in the post-state of $m_x$’s LP in $H$ must be the same as the AbDS in the post-state of $m_y$ in $S$. Formally, $(\forall m_x \in E^H \text{mths}, \forall m_y \in E^S \text{mths} : (\text{Pre}E[E^H.m_x.LP].\text{AbDS} = \text{Pre}M[E^S.m_y].\text{AbDS}) \land (E^H.m_x.inv = E^S.m_y.inv) \implies (\text{Post}E[E^H.m_x.LP].\text{AbDS} = \text{Post}M[E^S.m_y].\text{AbDS}) \land (E^H.m_x.resp = E^S.m_y.resp))$.

Readers familiar with the work of Zhu et. al [20] can see that this lemma is similar to Theorem 1 on showing linearizability of CDS $d$. In SubSection 4.1 and in SubSection 4.2 we prove this lemma specifically for lazy-list and hoh-locking-list.

Next, in the following lemmas we consider the methods of $H$ and $CS(H)$. As observed in SubSection 3.3, for any method $m_x$ in $CS(H)$ there is a corresponding method $m_y$ in $H$ having the same inv event, i.e., $H.m_x.inv = CS(H).m_x.inv$. We use this observation in the following lemma.

Lemma 9 For any method $m_x$ in $H, CS(H)$ the AbDS in the pre-state of the LP of $m_x$ in $H$ is the same as the AbDS in the pre-state of $m_x$ in $CS(H)$. Formally, $(\forall m_x \in E^H \text{mths}, E^{CS(H)} \text{mths} : \text{Pre}E[E^H.m_x.LP].\text{AbDS} = \text{Pre}M[E^{CS(H)}.m_x].\text{AbDS})$.

Proof. We prove by Induction on events which are the linearization points of the methods.

Base Step: Before the 1st LP event, the initial AbDS remains same because all the events in the concurrent execution before the 1st LP do not change AbDS.

Induction Hypothesis: Let us assume that for $k$ LP events, we know that
\[
\text{Pre}E[E^H.m_k.LP].\text{AbDS} = \text{Pre}M[E^{CS(H)}.m_k].\text{AbDS}.
\]

Induction Step: We have to prove that \text{Pre}E[E^H.m_{k+1}.LP].\text{AbDS} = \text{Pre}M[E^{CS(H)}.m_{k+1}].\text{AbDS} holds true.

We know from Induction Hypothesis that for $k$th method,
\[
\text{Pre}E[E^H.m_k.LP].\text{AbDS} = \text{Pre}M[E^{CS(H)}.m_k].\text{AbDS}
\]

From the construction of $CS(H)$, we get that $H.m_x.inv = CS(H).m_x.inv$. Combining this with Lemma 8 we have,
\[
(H.m_x.inv = CS(H).m_x.inv) \land (\text{Pre}E[E^H.m_k.LP].\text{AbDS} = \text{Pre}M[E^{CS(H)}.m_k].\text{AbDS})
\]
\[
\frac{\text{Lemma 8}}{(\text{Post}E[E^H.m_k.LP].\text{AbDS} = \text{Post}M[E^{CS(H)}.m_k].\text{AbDS})}
\]

(1)
From the Lemma 6, we have,

\[ \text{PostM}[E^{CS(H)}_{m_k}].AbDS \xrightarrow{\text{Lemma 6}} \text{PreM}[E^{CS(H)}_{m_{k+1}}].AbDS \]  \hspace{1cm} (2)

From the equation 1 we have,

\[ \text{PostE}[E^H_{m_k}.LP].AbDS = \text{PostM}[E^{CS(H)}_{m_k}].AbDS \]  \hspace{1cm} (3)

By combining the equation 3 and 2 we have,

\[ \text{PostE}[E^H_{m_k}.LP].AbDS = \text{PreM}[E^{CS(H)}_{m_{k+1}}].AbDS \]  \hspace{1cm} (4)

And from the Lemma 7 we have,

\[ \text{PostE}[E^H_{m_k}.LP].AbDS \xrightarrow{\text{Lemma 7}} \text{PreE}[E^H_{m_{k+1}}.LP].AbDS \]  \hspace{1cm} (5)

So, by combining equations 5 and 4 we get,

\[ \text{PreE}[E^H_{m_{k+1}}.LP].AbDS = \text{PreM}[E^{CS(H)}_{m_{k+1}}].AbDS \]  \hspace{1cm} (6)

This holds for all \( m_i \) in \( E^H \). Hence the lemma.

\[ \square \]

**Lemma 10** The return values for all the methods in \( H \& CS(H) \) are the same. Formally, \( \langle \forall m_x \in E^H.mths, E^{CS(H)}.mths : E^H.m_x.resp = E^{CS(H)}.m_x.resp \rangle \).

**Proof.** From the construction of \( CS(H) \), we get that for any method \( m_x \) in \( H, CS(H) \) the invocation parameters are the same. From Lemma 9, we get that the pre-states of all these methods are the same. Combining this result with Lemma 8, we get that the responses parameters for all these methods are also the same.

\[ \square \]

**Theorem 11** All histories \( H \) generated by the CDS \( d \) are linearizable.

**Proof.** From Lemma 10, we get that for all the methods \( m_x \), the responses in \( H \) and \( CS(H) \) are the same. This implies that \( H \) and \( CS(H) \) are equivalent to each other. Combining this with Lemma 5, we get that \( CS(H) \) respects the real-time order of \( H \). We had already observed from Assumption 2 that \( CS(H) \) is legal. Hence \( H \) is linearizable.

\[ \square \]

**Analysis of the Proof Technique:** Theorem 11 shows that proving CDS specific Lemma 8 implies that the CDS \( d \) under consideration is linearizable. Lemma 8 states that the contents of the AbDS in the pre-state of the LP event of a method \( m_x \) should be the same as the result of sequential execution of (some of) the methods of \( d \). Thus if the contents of AbDS in the pre-state of the LP event (satisfying the assumptions 3 & 4) cannot be produced by some sequential execution of the methods of \( d \) then it is most likely the case that either the LP or the design of the \( d \) is incorrect.

Further Lemma 8 requires that after the execution of the LP, the AbDS in the post-state must again be same as the sequential execution of some methods of \( d \) with the final method being \( m_x \). If this is not the case, then it implies that some other events of the method are also modifying the AbDS and hence indicating some error in the analysis.

Extending this thought, we also believe that the intuition gained in proving this lemma for \( d \) might give the programmers new insights in the working of the CDS which can result in designing new variants of it having some desirable properties.
4 Data-Structure Specific Proofs

In this section, we prove the data structure specific Lemma 8 described in the Section 3. In the SubSection 4.1, we give the proof for the Algorithm 1-5 of lazy list satisfies the requirements of the ds-specific lemma 8 which implies that it is linearizable. Similarly, in the SubSection 4.2, we provide the proof outline for the Algorithm 6-9 of hand-over-hand locking also satisfies the requirements of the ds-specific lemma 8 and it is linearizable.

4.1 Lazy List

In this section, we define the lazy list data structure. It is implemented as a set of nodes - concurrent set which is dynamically being modified by a fixed set of concurrent threads. In this setting, threads may perform insertion or deletion of nodes to the set. We describe lazy list based set algorithm based on Heller et.al. [6]. This is a linked list of nodes of type Node and it has four fields. The val field is a unique identifier of the node. The nodes are sorted in order of the val field. The marked field is of type boolean which indicates whether that node is logically present in the list or not. The next field is a reference to the next node in the list. The lock field is for ensuring access to a shared node which happens in a mutually exclusive manner. We say a thread acquires a lock and releases the lock when it executes a lock.acquire() and lock.release() method call respectively. We assume the next and marked of the node are atomic. This ensures that operations on these variables happen atomically. In the context of a particular application, the node structure can be easily modified to carry useful data (like weights etc).

```java
class Node{
    int val;
    Node next;
    boolean marked;
    Lock lock;
    Node(int key){
        val = key;
        marked = false;
        next = null;
        lock = new Lock();
    }
};
```

4.1.1 Methods Exported & Sequential Specification

In this section, we describe the methods exported by the lazy list data structure.

1. The Add(n) method adds a node n to the list, returns true if the node is not already present in the list else returns false.

2. The Remove(n) method removes a node n from the list, if it is present and returns true. If the node is not present, it returns false.

3. The Contains(n) returns true, if the list contains the node n; otherwise returns false.

Table 1 shows the sequential specification of the lazy-list. As the name suggests, it shows the behaviour of the list when all the methods are invoked sequentially. The Pre-state of each method is the shared state before inv event and the Post-state is also the shared state just after the resp event of a method (after executing it sequentially), as depicted in the Figure 1.
### 4 DATA-STRUCTURE SPECIFIC PROOFS

#### Table 1: Sequential Specification of the Lazy list

| Method   | Return Value | Pre-state(S: Pre-state of the method) | Post-state(S': Post-State of the method) |
|----------|--------------|---------------------------------------|----------------------------------------|
| Add(n)   | true         | S : (n \notin S.AbS)                  | S' : (n \notin S'.AbS)                  |
| Add(n)   | false        | S : (n \in S.AbS)                     | S' : (n \in S'.AbS)                    |
| Remove(n) | true        | S : (n \in S.AbS)                     | S' : (n \notin S'.AbS)                  |
| Remove(n) | false        | S : (n \notin S.AbS)                  | S' : (n \notin S'.AbS)                  |
| Contains(n) | true  | S : (n \in S.AbS)                     | S' : (n \in S'.AbS)                    |
| Contains(n) | false | S : (n \notin S.AbS)                  | S' : (n \notin S'.AbS)                  |

#### 4.1.2 Working of Lazy List Methods

In this section, we describe the implementation of the lazy list based set algorithm based on Heller et.al. [6] and the working of the various methods.

**Notations used in PseudoCode:**

\(↓, ↑\) denote input and output arguments to each method respectively. The shared memory is accessed only by invoking explicit \textit{read()} and \textit{write()} methods. The \textit{flag} is a local variable which returns the status of each operation. We use nodes \(n_1, n_2, n\) to represent \textit{node} references.

**Algorithm 1 Validate Method:** Takes two nodes, \(n_1, n_2\), each of type \textit{node} as input and validates for presence of nodes in the list and returns \textit{true} or \textit{false}

1:  procedure \textit{Validate} (\(n_1 ↓, n_2 ↓, \textit{flag} ↑\))
2:      if (\(\text{read}(n_1.\text{marked}) = \text{false} \land \text{read}(n_2.\text{marked}) = \text{false} \land \text{read}(n_1.\text{next}) = n_2 \)) then
3:          \(\text{flag} ← \text{true};\)
4:      else
5:          \(\text{flag} ← \text{false};\)
6:      end if
7:  return;
8: end procedure

**Algorithm 2 Locate Method:** Takes \textit{key} as input and returns the corresponding pair of neighboring node \(\langle n_1, n_2 \rangle\). Initially \(n_1, n_2\) are set to \textit{null}.

9:  procedure \textit{Locate} (\(\textit{key} ↓, n_1 ↑, n_2 ↑\))
10:    while (\textit{true}) do
11:        \(n_1 ← \text{read}(\textit{Head});\)
12:        \(n_2 ← \text{read}(n_1.\text{next});\)
13:        while (\(\text{read}(n_2.\text{val}) < \text{key}\)) do
14:            \(n_1 ← n_2;\)
15:            \(n_2 ← \text{read}(n_2.\text{next});\)
16:        end while
17:        \(\text{lock.acquire}(n_1);\)
18:        \(\text{lock.acquire}(n_2);\)
19:        if (\(\text{Validate}(n_1 ↓, n_2 ↓, \textit{flag} ↑)\)) then
20:            return;
21:        else
22:            \(\text{lock.release}(n_1);\)
23:            \(\text{lock.release}(n_2);\)
24:        end if
25:    end while
26: end procedure

#### 4.1.3 Working of the methods

**Working of the Add (\(n\)) method:** When a thread wants to add a node \(n\) to the list, it traverses the list from \textit{Head} without acquiring any locks until it finds a node with its key greater
Algorithm 3 Add Method: *key* gets added to the set if it is not already part of the set. Returns *true* on successful add and returns *false* otherwise.

```plaintext
procedure Add (key ↓, flag ↑)
Locate(key ↓, n1 ↑, n2 ↑);
if (read(n2.val) ≠ key) then
  write(n3, new node(key));
  write(n1.next, n2);
  flag ← true;
else
  flag ← false;
end if
lock.release(n1);
lock.release(n2);
return;
end procedure
```

Algorithm 4 Remove Method: *key* gets removed from the set if it is already part of the set. Returns *true* on successful remove otherwise returns *false*.

```plaintext
procedure Remove (key ↓, flag ↑)
Locate(key ↓, n1 ↑, n2 ↑);
if (read(n2.val) = key) then
  write(n2.marked, true);
  write(n1.next, n2.next);
  flag ← true;
else
  flag ← false;
end if
lock.release(n1);
lock.release(n2);
return;
end procedure
```

Algorithm 5 Contains Method: Returns *true* if *key* is part of the set and returns *false* otherwise.

```plaintext
procedure Contains (key ↓, flag ↑)
n ← read(Head);
while (read(n.val) < key) do
  n ← read(n.next);
end while
if (read(n.val) ≠ key) ∨ (read(n.marked)) then
  flag ← false;
else
  flag ← true;
end if
return;
end procedure
```

than or equal to *n*, say *ncurr* and it’s predecessor node, say *npred*. It acquires locks on the nodes *npred* and *ncurr* itself. It validates to check if *ncurr* is reachable from *npred*, and if both the nodes have not been deleted (marked). The algorithm maintains an invariant that all the unmarked nodes are reachable from *Head*. If the validation succeeds, the thread adds the *node(key)* between *npred* and *ncurr* in the list and returns true after unlocking the nodes. If it fails, the thread starts the traversal again after unlocking the locked nodes. This is described in Algorithm 3.

**Working of the Remove () method:** Each node of list has a boolean *marked* field. The removal of a node *n* happens in two steps: (1) The node *n*’s marked field is first set to *true*. This is referred to as logical removal. This ensures that if any node is being added or removed concurrently corresponding to that node, then Add method will fail in the validation process after checking the marked field. (2) Then, the pointers are changed so that *n* is removed from the list. This is referred to as physical deletion which involves changing the pointer of the predecessor of the marked node to its successor so that the deleted node is no longer reachable from the *Head* in the list. To achieve this, Remove(*n*) method proceeds similar to the Add(*n*).
The thread iterates through the list until it identifies the node \( n \) to be deleted. Then after \( n \) and its predecessor have been locked, logical removal occurs by setting the marked field to true. This is described in Algorithm 4.

**Working of the Contains \((n)\) method:** Method \( \text{Contains}(n) \) traverses the list without acquiring any locks. This method returns true if the node it was searching for is present and unmarked in the list, otherwise returns false. This is described in Algorithm 5.

### 4.1.4 The Linearization Points of the Lazy list methods

Here, we list the linearization points (LPs) of each method. Note that each method of the list can return either true or false. So, we define the LP for six methods:

1. \( \text{Add}(key, true) \): \( \text{write}(n_1.next, n_3) \) in Line 32 of \( \text{Add} \) method.
2. \( \text{Add}(key, false) \): \( \text{read}(n_2.val) \) in Line 29 of \( \text{Add} \) method.
3. \( \text{Remove}(key, true) \): \( \text{write}(n_2.marked, true) \) in Line 44 of \( \text{Remove} \) method.
4. \( \text{Remove}(key, false) \): \( \text{read}(n_2.val) \) in Line 43 of \( \text{Remove} \) method.
5. \( \text{Contains}(key, true) \): \( \text{read}(n.marked) \) in Line 59 of \( \text{Contains} \) method.
6. \( \text{Contains}(key, false) \): LP is the last among the following lines executed. There are three cases here:
   
   (a) \( \text{read}(n.val) \neq key \) in Line 59 of \( \text{Contains} \) method is the LP, in case of no concurrent \( \text{Add}(key, true) \).
   
   (b) \( \text{read}(n.marked) \) in Line 59 of \( \text{Contains} \) method is the LP, in case of no concurrent \( \text{Add}(key, true) \) (like the case of Step 5).
   
   (c) in case of concurrent \( \text{Add}(key, true) \) by another thread, we add a dummy event just before Line 32 of \( \text{add}(key, true) \). This dummy event is the LP of \( \text{Contains} \) method if: (i) if in the post-state of \( \text{read}(n.val) \) event in Line 59 of \( \text{Contains} \) method, \( n.val \neq key \) and \( \text{write}(n_1.next, n_3) \) (with \( n_3.val = key \)) in Line 32 of \( \text{Add} \) method executes before this \( \text{read}(n.val) \). (ii) if in the post-state of \( \text{read}(n.marked) \) event in Line 59 of \( \text{Contains} \) method, \( n.marked = true \) and \( \text{write}(n_1.next, n_3) \) (with \( n_3.val = key \)) in Line 32 of \( \text{Add} \) method executes before this \( \text{read}(n.marked) \). An example is illustrated in Figure 4.

### 4.1.5 Proof of Concurrent Lazy Linked List

In this subsection, we describe the lemmas to prove the correctness of the concurrent lazy list structure. We say a node \( n \) is a public node if it has an incoming link, which makes it reachable from the head of the linked list. We assume that Head and Tail node are public nodes.

**Observation 12** Consider a global state \( S \) which has a node \( n \). Then in any future state \( S' \) of \( S \), \( n \) is node in \( S' \) as well. Formally, \( (\forall S, S' : (n \in S.\text{nodes}) \land (S \sqsubseteq S') \Rightarrow (n \in S'.\text{nodes})) \).

With this observation, we assume that nodes once created do not get deleted (ignoring garbage collection).

**Observation 13** Consider a global state \( S \) which has a node \( n \) and it is initialized to \( n.val \).
Figure 4: An illustration of a concurrent set based linked list where the LP of the 
Contains method does not lie in the code of the method. (a) Thread \( T_3 \) begins executing \( \text{Contains}(7) \) by traversing the list until it finds a node with key greater than or equal to 7 (Line 59). At the same time, thread \( T_2 \) starts the process of deletion of node 7. (b) depicts that \( T_2 \) successfully performs deletion of 7. (c) After this, Thread \( T_1 \) tries to add a new node with key 7 and upon not encountering it in the list already; adds it successfully. Here thread \( T_3 \) has become slow and is still pointing to the deleted node 7. It now executes Line 59 and returns false; even though the node with key 7 is present in the list, thus resulting in a illegal sequentialisation. The correct LP order is obtained by linearising \( \text{Contains} \) just before the LP of the Add method. (d) shows the correct sequential history: \( T_2.\text{Remove}(7, \text{true}) <_H T_3.\text{Contains}(7, \text{false}) <_H T_1.\text{Add}(7, \text{true}) \).

13.1 Then in any future state \( S' \), where node \( n \) exists, the value of \( n \) does not change. Formally, 
\[
\langle \forall S, S' : (n \in S.\text{nodes}) \land (S \supset S') \land (n \in S'.\text{nodes}) \Rightarrow (S.n.val = S'.n.val) \rangle.
\]

13.2 Then in any past state \( S'' \), where node \( n \) existed, the value of \( n \) was the same. Formally, 
\[
\langle \forall S, S'' : (n \in S.\text{nodes}) \land (S'' \supset S) \land (n \in S''.\text{nodes}) \Rightarrow (S.n.val = S''.n.val) \rangle.
\]

Observation 14 Consider a global state \( S \) which has a node \( n \) and it is marked. Then in any future state \( S' \) the node \( n \) stays marked. Formally, \( \langle \forall S, S' : (n \in S.\text{nodes}) \land (S.n.\text{marked}) \land (S \supset S') \Rightarrow (S'.n.\text{marked}) \rangle. \)

Observation 15 Consider a global state \( S \) which has a node \( n \) which is marked. Then in any future state \( S' \), \( n.\text{next} \) remains unchanged. Formally, \( \langle \forall S, S' : (n \in S.\text{nodes}) \land (S.n.\text{marked}) \land (S \supset S') \Rightarrow (S'.n.\text{next} = S.n.\text{next}) \rangle. \)

Definition 16 \( S.\text{AbS} \equiv \{ n | (n \in S.\text{nodes}) \land (S.\text{Head} \rightarrow^* S.n) \land (\neg S.n.\text{marked}) \} \).

This definition of \( \text{AbS} \) captures the set of all nodes of \( \text{AbS} \) for the global state \( S \). It consists of all the nodes that are reachable from \( \text{Head} \) of the list (public) and are not marked for deletion.

Observation 17 Consider a global state \( S \) which is the post-state of return event of the method \( \text{Locate}(\text{key}) \) invoked in the Add or Remove methods. Suppose the Locate method returns \( \langle n_1, n_2 \rangle \). Then in the state \( S \), we have,

17.1 \( \langle (n_1, n_2) \in S.\text{nodes} \rangle. \)

17.2 \( \langle (S.\text{lock.acquire}(n_1) = \text{true}) \land (S.\text{lock.acquire}(n_2) = \text{true}) \rangle \)

17.3 \( \langle S.n_1.\text{next} = S.n_2 \rangle \)

17.4 \( \langle \neg (S.n_1.\text{marked}) \land \neg (S.n_2.\text{marked}) \rangle \)

Lemma 18 Consider the global state \( S \) which is the post-state of return event of the method \( \text{Locate}(\text{key}) \) invoked in the Add or Remove methods. Say, the Locate method returns \( (n_1, n_2) \). Then in the state \( S \), we have that \( (S.n_1.\text{val} < \text{key} \leq S.n_2.\text{val}) \).
Proof. Line 11 of Locate method initialises $S.n_1$ to Head and $S.n_2 = S.n_1.next$ by Line 12. The last time Line 14 in the while loop was executed, we know that $S.n_{1.val} < S.n_{2.val}$. The value of node does not change, from Observation 13. So, before execution of Line 17, we know that $S.n_{2.val} ≥ key$ and $S.n_{1.val} < S.n_{2.val}$. These nodes $\in S.nodes$ and $S.n_1.val < key ≤ S.n_2.val$. Also, putting together Observation 17.2, 17.3 and 13 that node $n_1$ and $n_2$ are locked (do not change), hence, the lemma holds when Locate returns.

Observation 19 Consider a global state $S$ which has a node $n$ that is marked. Then there will surely be some previous state $S'$ ($S' \subseteq S$) such that $S'$ is the state after return of Locate $n.val$ method.

Observation 20 Consider the global state $S$ which has a node $n$. If $S.n$ is unmarked and $S.n.next$ is marked, then $n$ and $n.next$ are surely locked in the state $S$.

Lemma 21 Consider a global state $S$ which is the post-state of return event of the Locate(key) method (invoked by the Add or Remove methods). Say, the Locate method returns $⟨n_1,n_2⟩$. Then in the state $S$, we have that the successor node of $n_2$ (if it exists) is unmarked i.e. $¬(S.n_{2.next}.marked)$.

Proof. We prove the lemma by using induction on the return events of the Locate method in $E^H$.

Base condition: Initially, before the first return of the Locate, we know that $(Head.key < Tail.key)$ and Head.next is Tail and Tail.marked is set to false and $(Head,Tail) \in S.nodes$. In this case, locate will return $⟨Head,Tail⟩$ such that the successor of Tail does not exist.

Induction Hypothesis: Say, upto the first $k$ return events of Locate, the successor of $n_2$ (if it exists) is unmarked.

Induction Step: So, by the observing the code, the $(k + 1)^st$ event which can be the return of the Locate method can only be at Line 20.

We prove by contradiction. Suppose when thread $T_1$ returns $⟨n_1,n_2⟩$ after invoking Locate method in state $S$, $n_{2.next}$ is marked. By Observation 17, it is known that, $(n_1,n_2) \in S.nodes$, $n_1,n_2$ are locked, $n_{1.next} = n_2$ and $(n_1,n_2)$ are unmarked. Suppose another thread say $T_2$ is trying to remove the node $n_{2.next}$. From the Observation 19, it needs to invoke the Locate method. Again, we know from the Observation 17 that when Locate method returns, it must have acquired lock on $n_2$ and $n_{2.next}$. However, since $n_2$ is already locked, it cannot proceed until $T_1$ has released its lock on $n_2$. Hence the node $n_{2.next}$ cannot be marked. This contradicts our initial assumption.

Observation 22 Consider a global state $S$ which has two non-consecutive nodes $n_p$, $n_q$ where $n_p$ is unmarked and $n_q$ is marked. Then we have that in any future state $S'$, $n_p$ cannot point to $n_q$.

Formally, $¬(S.n_p.marked) \land (S.n_q.marked) \land (S.n_{p.next} ≠ n_q) \land (S \subseteq S') \implies (S'.n_p.next ≠ S'.n_q)$.

Lemma 23 In any global state $S$, consider three nodes $p$, $q$ & $r$ such that $p.next = q$ and $q.next = r$ and only $q$ is marked. Then in a future state $S'$ ($S \subseteq S'$) where $p.next = q$ and $p$ is still unmarked, $r$ will surely be unmarked.

Proof. We prove the lemma by contradiction. Suppose in state $S'$, node $r$ is marked and $p.next = q$ and $q.next = r$. From Observation 14, we know that $q$ will remain marked. From the Observation 19 we know that any node is marked only after invoking the Locate method. Say, the node $q$ was marked by the thread $T_1$ by invoking the Remove method. As we know from the Lemma 21 that when $T_1$.Locate returns $⟨q,q.next = r⟩$, the successor of $q$ (i.e. $r$) is unmarked, which contradicts our intial assumption. Hence the lemma holds.
Lemma 24 For any node $n$ in a global state $S$, we have that $(\forall n \in S\text{.nodes} \land n\text{.next} \neq \text{null} : S.n\text{.val} < S.n\text{.next}\text{.val})$.

Proof. We prove the lemma by inducting on all events in $E^H$ that change the next field of a node $n$.

Base condition: Initially, before the first event that changes the next field, we know that $(\text{Head.key} < \text{Tail.key}) \land (\text{Head.Tail}) \in S\text{.nodes}$.

Induction Hypothesis: Say, in any state $S$ up to first $k$ events that change the next field of any node, $\forall n \in S\text{.nodes} \land S.n\text{.next} \neq \text{null} : S.n\text{.val} < S.n\text{.next}\text{.val}$.

Induction Step: So, by observing the code, the $(k + 1)^{st}$ event which can change the next field can be only one among the following:

1. Line 31 of Add method: Let $S_1$ be the state after the Line 29. We know that when Locate (Line 28) returns by the Observation 17, $S_1.n_1$ & $S_1.n_2$ are not marked, $S_1.n_1$ & $S_1.n_2$ are locked, $S_1.n_1\.next = S_1.n_2$. By the Lemma 18 we have $(S_1.n_1\.val \leq S_1.n_2\.val)$. Also we know from Observation 13 that node value does not change, once initialised. To reach Line 31, $n_2\.val \neq \text{key}$ in the Line 29 must evaluate to true. Therefore, $(S_1.n_1\.val < \text{key} < S_1.n_2\.val)$. So, a new node $n_3$ is created in the Line 30 with the value $\text{key}$ and then a link is added between $n_3\.next$ and $n_2$ in the Line 31. So this implies $n_3\.val < n_2\.val$ even after execution of line 31 of Add method.

2. Line 32 of Add method: Let $S_1$ and $S_2$ be the states after the Line 28 and Line 32 respectively. By observing the code, we notice that the Line 32 (next field changing event) can be executed only after the Locate method returns. From Lemma 18, we know that when Locate returns then $S_1.n_1\.val < \text{key} < S_1.n_2\.val$. To reach Line 32 of Add method, Line 29 should ensure that $S_1.n_2\.val \neq \text{key}$. This implies that $S_1.n_1\.val < \text{key} < S_1.n_2\.val$. From Observation 17.3, we know that $S_1.n_1\.next = S_1.n_2$. Also, the atomic event at Line 32 sets $S_2.n_1\.next = S_2.n_3$ where $S_2.n_3\.val = \text{key}$.

Thus from $S_2.n_1\.val < (S_2.n_3\.val = \text{key}) < S_2.n_2\.val$ and $S_2.n_1\.next = S_2.n_3$, we get $S_2.n_1\.val < S_2.n_1\.next\.val$. Since $(n_1, n_2) \in S\text{.nodes}$ and hence, $S.n_1\.val < S.n_1\.next\.val$.

3. Line 45 of Remove method: Let $S_1$ and $S_2$ be the states after the Line 42 and Line 44 respectively. By observing the code, we notice that the Line 45 (next field changing event) can be executed only after the Locate method returns. From Lemma 18, we know that when Locate returns then $S_1.n_1\.val < \text{key} < S_1.n_2\.val$. To reach Line 45 of Remove method, Line 43 should ensure that $S_1.n_2\.val = \text{key}$. Also we know from Observation 13 that node value does not change, once initialised. This implies that $S_1.n_1\.val = (\text{key} = S_2.n_2\.val)$. From Observation 17.3, we know that $S_2.n_1\.next = n_2$. Also, the atomic event at line 50 sets $S_2.n_1\.next = S_2.n_2\.next$.

We know from Induction hypothesis, $S_2.n_2\.val < S_2.n_2\.next\.val$. Thus from $S_2.n_1\.val < S_2.n_2\.val$ and $S_2.n_1\.next = S_2.n_2\.next$, we get $S_2.n_1\.val < S_2.n_1\.next\.val$. Since $(n_1, n_2) \in S\text{.nodes}$ and hence, $S.n_1\.val < S.n_1\.next\.val$.

Corollary 25 There cannot exist two nodes with the same key in $S\text{.AbS}$ of a particular global state $S$.

Lemma 26 In a global state $S$, any non-marked public node $n$ is reachable from Head. Formally, $(\forall S, n : (n \in S\text{.nodes}) \land (\neg S\text{.marked}) \implies (S\text{.Head} \rightarrow^* S.n))$.
Proof. We prove by Induction on events that change the next field of the node (as these affect reachability), which are Line 31 & 32 of Add method and Line 45 of Remove method. It can be seen by observing the code that Locate and Contains method do not have any update events.

Base step: Initially, before the first event that changes the next field of any node, we know that \((\text{Head}, \text{Tail}) \in \text{S.nodes} \land \neg (\text{Head}.\text{marked}) \land \neg (\text{Tail}.\text{marked}) \land (\text{Head} \rightarrow \text{Tail})\).

Induction Hypothesis: Say, the first \(k\) events that changed the next field of any node in the system did not make any unmarked node unreachable from the Head.

Induction Step: As seen by observing the code, the \((k + 1)^{\text{st}}\) event can be one of the following events that change the next field of a node:

1. **Line 30 & 31 of Add method**: Let \(S_1\) be the state after the Line 28. Line 30 of the Add method creates a new node \(n_3\) with value \(\text{key}\). Line 31 then sets \(S_1.n_3.\text{next} = S_1.n_2\). Since this event does not change the next field of any node reachable from the Head of the list, the lemma is not violated.

2. **Line 32 of Add method**: By observing the code, we notice that the Line 31 (next field changing event) can be executed only after the Locate method returns. Let \(S_1\) and \(S_2\) be the states after the Line 29 and Line 32 respectively. From Observation 17.3, we know that when Locate returns then \(S_1.n_1.\text{marked} = S_1.n_2.\text{marked} = \text{false}\). From Line 30 & 31 of Add method, \((S_1.n_1.\text{next} = S_1.n_3) \land (S_1.n_3.\text{next} = S_1.n_2) \land (\neg S_1.n_3.\text{marked})\). It is to be noted that (From Observation 17.2), \(n_1 \& n_2\) are locked, hence no other thread can change \(S_1.n_1.\text{marked} \& S_1.n_2.\text{marked}\). Also from Observation 13, a node’s key field does not change after initialization. Before executing Line 32, \(S_1.n_1.\text{marked} = \text{false}\) and \(S_1.n_1\) is reachable from \(\text{Head}\). After Line 32, we know that from \(S_2.n_1\), unmarked node \(S_2.n_3\) is also reachable. Formally, \((S_2.\text{Head} \rightarrow^* S_2.n_1) \land (\neg S_2.n_1.\text{marked}) \land (S_2.n_1 \rightarrow S_2.n_3) \land (\neg S_2.n_3.\text{marked}) \implies (S_2.\text{Head} \rightarrow^* S_2.n_3)\).

3. **Line 45 of Remove method**: Let \(S_1\) and \(S_2\) be the states after the execution of Line 43 and Line 45 respectively. By observing the code, we notice that the Line 45 (next field changing event) can be executed only after the Locate method returns. From Observation 17.2, we know that when Locate returns then \(S_1.n_1.\text{marked} = S_1.n_2.\text{marked} = \text{false}\). We know that \(S_1.n_1\) is reachable from \(\text{Head}\) and from Line 44 and 45 of Remove method, \(S_2.n_2.\text{marked} = \text{true}\) and later sets \(S_2.n_1.\text{next} = S_2.n_2.\text{next}\). It is to be noted that (From Observation 17.2), \(n_1 \& n_2\) are locked, hence no other thread can change \(S_1.n_1.\text{marked} \& S_1.n_2.\text{marked}\). This event does not affect reachability of any non-marked node. Also from Observation 13, a node’s key does not change after initialization. And from Observation 14, a marked node continues to remain marked. If \(S_2.n_2.\text{next}\) is unmarked (reachable), then it continues to remain unmarked & reachable. So this event does not violate the lemma.

Lemma 27 Consider the global state \(S\) such that for any unmarked node \(n\), if there exists a key strictly greater than \(n.\text{val}\) and strictly smaller than \(n.\text{next}.\text{val}\), then the node corresponding to the key does not belong to \(S.\text{AbS}\). Formally, \(\langle S, n, \text{key} : \neg(S.n.\text{marked}) \land (S.n.\text{val} < \text{key} < S.n.\text{next}.\text{val}) \implies \text{node(key) } \notin S.\text{AbS}\rangle\).

Proof. We prove by contradiction. Suppose there exists a key which is strictly greater than \(n.\text{val}\) and strictly smaller than \(n.\text{next}.\text{val}\) and then it belongs to \(S.\text{AbS}\). From the Observation 12, we know that node \(n\) is unmarked in a global state \(S\), so it is belongs to \(S.\text{nodes}\). But we know from Lemma 26 that any unmarked node should be reachable from Head. Also, from Definition 16, any unmarked node i.e. \(n\) in this case, is reachable from Head and belongs to \(S.\text{AbS}\). From the
Then we have the sequential specification of all methods as follows, the execution of the method and methods.

AbS in 44 of Remove method can change the AbS.

Observation 13, we know that the node’s key value does not change after initialization. So both the nodes n and n.next belong to S.AbS. From the Lemma 24 we know that n.val < n.next.val. So node n’ can not be present in between n and n.next. Which contradicts the initial assumption. Hence (∀S,n, key : ¬(S.n.marked) ∧ (S.n.val < key < S.n.next.val) ⇒ node(key) ∉ S.AbS).

Lemma 28 Only the events write(n_1.next, n_3) in 32 of Add method and write(n_2.marked, true) in 44 of Remove method can change the AbS.

Proof. It is to be noted that the Locate and Contains methods do not have any update events. By observing the code, it appears that the following (write) events of the Add and Remove method can change the AbS:

1. Line 30 & 31 of Add method: In Algorithm 3, let S_1.AbS be the initial state of the AbS, such that we know from Line 29 that key ∉ S_1.AbS. Line 30 of the Add method creates a node n_3 with value key, i.e. n_3.val = key. Now, Line 31 sets S_1.n_3.next = S_1.n_2. Since this event does not change the next field of any node reachable from the Head of the list, hence from Definition 16, S_1.AbS remains unchanged after these events.

2. Line 32 of Add method: Let S_1 and S_2 be the states after the Line 29 and Line 32 respectively. At line 29, true evaluation of the condition leads to the execution of S_1.n_1.next = S_1.n_3 at Line 32. Also, S_1.n_1 and S_1.n_2 are locked, therefore from Observation 17, Head → S_1.n_1. From line 31 & 32 we get: S_1.n_1 → S_1.n_3 → S_1.n_2. Hence, Head → S_1.n_1 → S_1.n_3 → S_1.n_2 follows. We have ¬ (S_2.n_3.marked) ∧ (Head → S_2.n_3). Thus from Definition 16, S_1.AbS changes to S_2.AbS = S_1.AbS ∪ n_3.

3. Line 44 of Remove method: Let S_1 be the state after the Line 44. By observing the code, we notice that the state before execution of Line 44 satisfies that key ∈ S.AbS. After execution of line 44, AbS changes such that key ∉ S.AbS. Note that this follows from Definition 16.

4. Line 45 of Remove method: Let S_1 be the state after the execution of Line 44. Till line 44 of the Remove method, S.AbS has changed such that S_1.n_3.val ∉ S.AbS. So even after the execution of Line 45 when S_1.n_1.next is set to S_1.n_2.next, S.AbS remains unchanged (from Definition 16).

Hence, only the events in Line 32 of Add method and in Line 44 of Remove method can change the AbS.

Corollary 29 Both these events write(n_1.next, n_3) in 32 of Add method and write(n_2.marked, true) in 44 of Remove method change the AbS are in fact the Linearization Points(LPs) of the respective methods.

Observation 30 Consider a sequential history S. Let S be a global state in S.allStates before the execution of the method and S’ be a global state just after the return of the method (S ⊨ S’). Then we have the sequential specification of all methods as follows,

30.1 For a given key, suppose node(key) ∉ S.AbS. In this state, suppose Add (key) method is (sequentially) executed. Then the Add method will return true and node(key) will be present in S’.AbS. Formally, (∀S : (node(key) ∉ S.AbS) ⇒ S.Add(key, true) ∧ (S ⊨ S’) ∧ (node(key) ∈ S’.AbS)).
30.2 For a given key, suppose node(key) ∈ S.AbS. In this state, suppose Add (key) method is (sequentially) executed. Then the Add method will return false and node(key) will continue to be present in S'.AbS. Formally, ⟨∀S : (node(key) ∈ S.AbS) seq-add S.Add(key, false) ∧ (S ⊂ S') ∧ (node(key) ∈ S'.AbS)⟩.

30.3 For a given key, suppose node(key) ∈ S.AbS. In this state, suppose Remove (key) method is (sequentially) executed. Then the Remove method will return true and node(key) will not be present in S'.AbS. Formally, ⟨∀S : (node(key) ∈ S.AbS) seq-remove S.Remove(key, true) ∧ (S ⊂ S') ∧ (node(key) ∉ S'.AbS)⟩.

30.4 For a given key, suppose node(key) ∉ S.AbS. In this state, suppose Remove (key) method is (sequentially) executed. Then the Remove method will return false and node(key) will continue to be not present in S'.AbS. Formally, ⟨∀S : (node(key) ∉ S.AbS) seq-remove S.Remove(key, false) ∧ (S ⊂ S') ∧ (node(key) ∉ S'.AbS)⟩.

30.5 For a given key, suppose node(key) ∈ S.AbS. In this state, suppose Contains (key) method is (sequentially) executed. Then the Contains method will return true and node(key) will continue to be present in S'.AbS. Formally, ⟨∀S : (node(key) ∈ S.AbS) seq-contains S.Contains(key, true) ∧ (S ⊂ S') ∧ (node(key) ∈ S'.AbS)⟩.

30.6 For a given key, suppose node(key) ∉ S.AbS. In this state, suppose Contains (key) method is (sequentially) executed. Then the Contains method will return false and node(key) will continue to be not present in S'.AbS. Formally, ⟨∀S : (node(key) ∉ S.AbS) seq-contains S.Contains(key, false) ∧ (S ⊂ S') ∧ (node(key) ∉ S'.AbS)⟩.

Lemma 31 If some Add (key) method returns true in E^H then,

31.1 The node(key) is not present in the pre-state of LP event of the method. Formally, ⟨Add(key, true) ⇒ (node(key) ∉ (PreE[E^H. Add(key, true). LP]. AbS))⟩.

31.2 The node(key) is present in the post-state of LP event of the method. Formally, ⟨Add(key, true) ⇒ (node(key) ∈ (PostE[E^H. Add(key, true). LP]. AbS))⟩.

Proof.

• 31.1: From Line 28, when Locate returns in state S_1 we know that (from Observation 17 & Lemma 26), nodes n_1 and n_2 are locked, (n_1, n_2) ∈ S_1.nodes and n_1.next = n_2. Also, S_1.n_1.val < key ≤ S_1.n_2.val from Lemma 18. If this method is to return true, Line 29, n_2.val ≠ key must evaluate to true. Also from Lemma 27, we conclude that node(key) does not belong to S_1.AbS. And since from Observation 13, no node changes its key value after initialization, node(key) ∉ S_2.AbS, where S_2 is the pre-state of the LP event of the method. Hence node(key) ∉ (PreE[E^H. Add(key, true). LP]. AbS).

• 31.2: From the Lemma 31.1 we get that node(key) is not present in the pre-state of the LP event. From Line 28, it is known that only LP event can change the S.AbS. Now after execution of the LP event i.e. write(n_1.next, n_3) in the Line 32, node(key) ∈ S’.AbS, where S’ is the post-state of the LP event of the method. Hence, ⟨Add(key, true) ⇒ (node(key) ∈ (PostE[E^H. Add(key, true). LP]. AbS))⟩.

Lemma 32 If some Add (key) method returns false in E^H, then
32.1 The node(key) is present in the pre-state of LP event of the method. Formally, \( \langle \text{Add}(\text{key}, \text{false}) \rangle \Rightarrow (\text{node}(\text{key}) \in (\text{PreE}[\text{E}^H.\text{Add}(\text{key}, \text{false}).\text{LP}].\text{AbS})) \).

32.2 The node(key) is present in the post-state of LP event of the method. Formally, \( \langle \text{Add}(\text{key}, \text{false}) \rangle \Rightarrow (\text{node}(\text{key}) \in (\text{PostE}[\text{E}^H.\text{Add}(\text{key}, \text{false}).\text{LP}].\text{AbS})) \).

Proof.

- **32.1:** From Line 28, when Locate returns in state S1 we know that (from Observation 17 & Lemma 26), nodes n1 and n2 are locked, \((n_1, n_2) \in S.nodest\) and \(n_1.next = n_2\). Also, \(n_1.val < \text{key} \leq n_2.val\) from Lemma 18. If this method is to return false, Line 29, \(n_2.val \neq \text{key}\) must evaluate to false. So node(key) which is \(n_2\) belongs to \(S_1.\text{AbS}\). And since from Observation 13, no node changes its key value after initialization and the fact that it is locked, node(key) \(\in S_2.\text{AbS}\), where \(S_2\) is the pre-state of the LP event of the method. Hence node(key) \(\in (\text{PreE}[\text{E}^H.\text{Add}(\text{key}, \text{false}).\text{LP}].\text{AbS})\).

- **32.2:** From the Lemma 32.1 we get that node(key) is present in the pre-state of the LP event. This LP event \(n_2.val \neq \text{key}\) in Line 29 does not change the \(S.\text{AbS}\), Now after execution of the LP event the node(key) also present in the \(S'.\text{AbS}\), where \(S'\) is the post-state of the LP event of the method. Hence, \(\langle \text{Add}(\text{key}, \text{false}) \rangle \Rightarrow (\text{node}(\text{key}) \in (\text{PostE}[\text{E}^H.\text{Add}(\text{key}, \text{false}).\text{LP}].\text{AbS}))\).

\[
\square
\]

**Lemma 33** If some Remove (key) method returns true in \(E^H\), then

33.1 The node(key) is present in the pre-state of LP event of the method. Formally, \(\langle \text{Remove}(\text{key}, \text{true}) \rangle \Rightarrow (\text{node}(\text{key}) \in (\text{PreE}[\text{E}^H.\text{Remove}(\text{key}, \text{true}).\text{LP}].\text{AbS}))\).

33.2 The node(key) is not present in the post-state of LP event of the method. Formally, \(\langle \text{Remove}(\text{key}, \text{true}) \rangle \Rightarrow (\text{node}(\text{key}) \notin (\text{PostE}[\text{E}^H.\text{Remove}(\text{key}, \text{true}).\text{LP}].\text{AbS}))\).

Proof.

- **33.1:** From Line 42, when Locate returns in state S1 we know that (from Observation 17 & Lemma 26), nodes n1 and n2 are locked, \((n_1, n_2) \in S.nodest\) and \(n_1.next = n_2\). Also, \(S_1.n_1.val < \text{key} \leq S_1.n_2.val\) from Lemma 18. If this method is to return true, Line 43, \(n_2.val = \text{key}\) must evaluate to true. So we know that node(key) which is \(n_2\) belongs to \(S_1.\text{AbS}\). And since from Observation 13, no node changes its key value after initialization, node(key) \(\in S_2.\text{AbS}\), where \(S_2\) is the pre-state of the LP event of the method. Hence node(key) \(\notin (\text{PreE}[\text{E}^H.\text{Remove}(\text{key}, \text{true}).\text{LP}].\text{AbS})\).

- **33.2:** From the Lemma 33.1 we get that node(key) is present in the pre-state of the LP event. This LP event write\((n_2.marked, \text{true})\) in the Line 44 changes the \(S.\text{AbS}\). Now after execution of the LP event the node(key) will not present in the \(S'.\text{AbS}\), where \(S'\) is the post-state of the LP event of the method. Hence, \(\langle \text{Remove}(\text{key}, \text{true}) \rangle \Rightarrow (\text{node}(\text{key}) \notin (\text{PostE}[\text{E}^H.\text{Remove}(\text{key}, \text{true}).\text{LP}].\text{AbS}))\).

\[
\square
\]

**Lemma 34** If some Remove (key) method returns false in \(E^H\), then

34.1 The node(key) is not present in the pre-state of LP event of the method. Formally, \(\langle \text{Remove}(\text{key}, \text{false}) \rangle \Rightarrow (\text{node}(\text{key}) \notin \text{PreE}[\text{E}^H.\text{Remove}(\text{key}, \text{false}).\text{LP}].\text{AbS})\).
34.2 The node(key) is not present in the post-state of LP event of the method. Formally, \( \langle \text{Remove(key, false)} \rangle \implies (\text{node(key}) \notin \text{PostE}[E^H.\text{Remove(key, false)}.LP].\text{AbS}] \).

Proof.

- **34.1:** From Line 42, when Locate returns in state \( S_1 \) we know that (from Observation 17 & Lemma 26), nodes \( n_1 \) and \( n_2 \) are locked, \((n_1, n_2) \in S_1.\text{nodes} \) and \( n_1.\text{next} = n_2 \). Also, \( S_1.n_1.\text{val} < \text{key} \leq S_1.n_2.\text{val} \) from Lemma 18. If this method is to return false, Line 43, \( n_2.\text{val} = \text{key} \) must evaluate to false. Also from Lemma 27, we conclude that \( \text{node(key)} \) does not belong to \( S_1.\text{AbS} \). And since from Observation 13, no node changes its key value after initialization, \( \text{node(key)} \in S_2.\text{AbS} \), where \( S_2 \) is the pre-state of the LP event of the method. Hence \( \text{node(key)} \notin \langle \text{PreE}[E^H.\text{Remove(key, false)}.LP].\text{AbS}] \).

- **34.2:** From the Lemma 34.1 we get that \( \text{node(key)} \) is not present in the pre-state of the LP event. This LP event \( \langle \text{read}(n_2.\text{val}) = \text{key} \rangle \) in the Line 43 does not change the \( S.\text{AbS} \). Now after execution of the LP event the \( \text{node(key)} \) will not present in the \( S’.\text{AbS} \), where \( S’ \) is the post-state of the LP event of the method. Hence, \( \langle \text{Remove(key, false)} \rangle \implies (\text{node(key}) \notin \langle \text{PostE}[E^H.\text{Remove(key, false)}.LP].\text{AbS}] \).

\[\square\]

**Lemma 35** Consider a global state \( S \) which has two consecutive nodes \( n_p, n_q \) which are marked. Then we say that marking event of \( n_p \) happened before marking event of \( n_q \). Formally, \( \forall S : (n_p, n_q \in S.\text{nodes}) \land (S.n_p.\text{marked}) \land (S.n_q.\text{marked}) \land (S.n_p.\text{next} = S.n_q) \implies (n_p.\text{marked} <_E n_q.\text{marked}) \).

**Proof.** We prove by contradiction. We assume that \( n_q \) was marked before \( n_p \). Let \( S’ \) be the post-state of marking of the node \( n_q \). It can be seen as in Figure 5 that the state \( S \) follows \( S’ \), i.e., \( S’ \sqsubseteq S \). This is because in state \( S \) both \( n_p \) & \( n_q \) are marked. So we know that in \( S’ \), \( n_p \) is unmarked and \( n_q \) is marked.

Now suppose in \( S’\): \( (n_p.\text{next} \neq n_q) \). So, \( (S’.n_p.\text{next} \neq S’.n_q) \land (\neg S’.n_p.\text{marked}) \) also in the state \( S \), we have that \( S.n_p.\text{next} = S.n_q \) and \( n_p \) and \( n_q \) are both marked. This contradicts the Observation 22 that \( S’.n_p.\text{next} \neq S’.n_q \). Hence in \( S’ \): \( n_p.\text{next} \) must point to \( n_q \).

Consider some state \( S’’ \) immediately before marking event of \( n_q \). We know that \( S’’.n_p.\text{next} = S’’.n_q \) (similar argument), and \( n_p, n_q \) are both unmarked (from Observation 17.2). Then in some state \( R \) after \( S’ \) and before \( S, n_p.\text{next} \neq n_q \). From Observation 22, unmarked node cannot point to marked node. Hence in state \( S \) also, we will have that \( S.n_p.\text{next} \neq S.n_q \). This contradicts the given statement that \( S.n_p.\text{next} = S.n_q \). Hence proved that in \( S’ \), \( n_p \) was marked before \( n_q \). \( \square \)

**Lemma 36** If some Contains (key) method returns true in \( E^H \), then

36.1 The node(key) is present in the pre-state of LP event of the method. Formally, \( \langle \text{Contains(key, true)} \rangle \implies (\text{node(key}) \in \text{PreE}[E^H.\text{Contains(key, true)}.LP].\text{AbS}] \).

---

24
concurrently with a Remove (key) method and node \( S.n \) then marking of

\[ \langle \text{Contains}(\text{key}, \text{true}) \rangle \implies (\text{node}(\text{key}) \in \text{PreE}[E^H.\text{Contains}(\text{key}, \text{true}).\text{LP}].\text{AbS}) \].

**Proof.**

- **36.1:** By observing the code, we realize that at the end of while loop at Line 58 of \text{Contains} method, \( n.val \geq \text{key} \). To return \text{true}, \( n.marked \) should be false in \( \langle \text{PreE}[E^H.\text{Contains}.\text{LP}].\text{AbS} \rangle \). But we know from Lemma 26 that any unmarked node should be reachable from head. Also, from Definition 16, any unmarked nodes that are reachable belong to \text{AbS} in that state. From the Observation 13 we know that the node’s key value does not change after initialization. Hence

\[ \text{node}(\text{key}) \in \langle \text{PreE}[E^H.\text{Contains}(\text{key}, \text{true}).\text{LP}].\text{AbS} \rangle \]

- **36.2:** From the Lemma 36.1 we get that \( \text{node}(\text{key}) \) is present in the pre-state of the LP event. This LP event \((\text{read}(n.val) \neq \text{key}) \lor (\text{read}(n.marked))\) in the Line 59 does not change the \( S.\text{AbS} \). Now after execution of the LP event the \( \text{node}(\text{key}) \) will be present in the \( S'.\text{AbS} \), where \( S' \) is the post-state of the LP event of the method. Hence, \( \langle \text{Contains}(\text{key}, \text{true}) \rangle \implies (\text{node}(\text{key}) \in \langle \text{PostE}[E^H.\text{Contains}(\text{key}, \text{true}).\text{LP}].\text{AbS} \rangle) \).

\[ \square \]

**Lemma 37** Consider a global state \( S \) which has a node \( n \). If \text{Contains} (key) method is running concurrently with a Remove (key) method and \( \text{node}(\text{key}) = n \) and \( n \) is marked in the state \( S \), then marking of \( S.n \) happened only after \text{Contains} (key) started.

**Proof.**

**Notations used in Lemma 38:**

\( \text{Contains}(\text{key}) \) executes the while loop to find out location of the node \( n_x \) where \( n_x.val \leq \text{key} \) and \( n_x \in \text{AbS} \). We denote execution of the last step \( n_x = \text{read}(n_{x-1}.next) \) which satisfies \( n_x.val \leq \text{key} \). Also note that \( n_{x-1} \) represents the execution of penultimate loop iteration in sequential scenario. Figure 6 depicts the global state used in the Lemma 38.

1. \( S_{x-1} \): Global state after the execution of \( \text{read}(n_{x-1}.next) \) at Line 57.
2. \( S'_{x-1} \): Global state after the execution of \( \text{read}(n_{x-1}.val) \) at Line 56.
3. \( S_x \): Global state after the execution of \( \text{read}(n_{x-1}.next) \) at Line 57.
4. \( S'_x \): Global state after the execution of \( \text{read}(n_x.val) \) at Line 59.
5. \( S''_x \): Global state after the execution of \( \text{read}(n_x.marked) \) at Line 59.

![Figure 6: The global state representation for Lemma 38](image)

**Lemma 38** If some \text{Contains} (key) method returns false in \( E^H \), then

- **38.1** The \( \text{node}(\text{key}) \) is not present in the pre-state of LP event of the method. Formally,

\[ \langle \text{Contains}(\text{key}, \text{false}) \rangle \implies (\text{node}(\text{key}) \notin \text{PreE}[E^H.\text{Contains}(\text{key}, \text{false}).\text{LP}].\text{AbS}) \].

\[ \square \]
38.2 The node(key) is also not present in the post-state of LP event of the method. Formally, \( \langle \text{Contains}(\text{key, false}) \implies (\text{node(key)} \notin \text{PreE}[E^H.\text{Contains(\text{key, false}).LP}.\text{AbS})] \rangle. \)

Proof.

- **38.1**: There are following cases:
  
  1. **Case 1**: key is not present in the Pre-State of read(n.x.val \( \neq \) key) event at Line 59 of Contains method, which is the LP of contains(key, false). We assume that there is no concurrent add from \( S_1 \) until \( S_x'. \)

```plaintext
\[ S_{x-1}.n_{x-1}.val \geq key \quad \text{(Line 56 of the Contains method)} \quad (7) \]

\[ S_{x-1}.n_{x-1}.val < key \quad \text{(Line 57 of the Contains method)} \quad (8) \]

\[ S_x.n_x.val > key \quad \text{(Line 59 of the Contains method)} \quad (9) \]

\[ S_{x-1}.n_{x-1}.val < S_{x-1}.n_{x-1}.next.val \quad \text{(from Lemma 24)} \quad (10) \]

\[ S_x.n_x.next = S_x.n_x \quad \text{(Line 57 of the contains method)} \quad (11) \]

\[ S_{x-1}.n_{x-1}.val < S_{x-1}.n_x.val \quad \text{(from Equation 10 & 11 & Observation 13)} \quad (12) \]

Combining the equations 8, 9 & 12 we have,

\[ (S_{x-1}.n_{x-1}.val < key) \implies (node(key) \notin S_{x-1}.AbS) \quad (13) \]

Now since no concurrent add on key happens between \( S_1 \) until \( S_x' \) we have that,

\[ \langle node(key) \notin S_x'.AbS \rangle \quad (14) \]

- **Given**: \((S_{x-1}.Head \rightarrow * S_{x-1}.n_{x-1-1}) \land (S_{x-1}.n_{x-1-1}.marked = true)\)

To Prove: node(key) \( \notin S_x'.AbS\)

From given, we have that,

\[ (n_{x-1} \notin S_{x-1}.AbS) \quad (15) \]
Let $n_i$ be the first unmarked node belonging to $S'_i$.AbS while traversing the linked list of $n_1, \ldots, n_i, n_{i+1}, n_{i+2}, \ldots, n_{x-1}, n_x, \ldots$ nodes. Therefore,

$$ n_i \in S'_i.\text{AbS} \quad (16) $$

In the worst case, $n_i$ could be the Head node $n_1$.

We know that, $(n_{i+1} \text{ to } n_{x-1}) \notin (S'_{i-1}.\text{AbS} \text{ to } S'_{x-1}.\text{AbS}) \quad (17)$

In the linked list of $n_1, \ldots, n_i, n_{i+1}, n_{i+2}, \ldots, n_{x-1}, n_x, \ldots$ nodes, where $n_{i+1}, n_{i+2}, \ldots, n_{x-1}$ are marked and consecutive, we can conclude (from Lemma 35) that,

$$(S'_{i+2}.n_{i+1}.\text{next} = S'_{i+2}.n_{i+2}) \land (S'_{i+2}.n_{i+1}.\text{marked}) \land (S'_{i+2}.n_{i+2}.\text{marked}) \Rightarrow (n_{i+1}.\text{marking} < E n_{i+2}.\text{marking}) \quad (18)$$

In state $S'_i$, we know that $n_{i}.\text{next} = n_{i+1}$. Depending upon the status of node $n_{i+1}$ in $S'_i$, we have two possible situations:

i. $S'_i.n_{i+1}.\text{unmarked}$

Since we know that in $S'_{i+1}: n_{i+1}.\text{marked}$. Thus we have that,

$$\text{Contains}.\text{read}(n_i) < E \text{Remove}.\text{marking}(n_{i+1}) < E \text{Remove}.\text{marking}(n_{i+2}) \quad (19)$$

ii. $S'_i.n_{i+1}.\text{marked}$

We know that in $S'_{i+1}: n_{i+1}.\text{next} = n_{i+2}$. From Equation 18, we can conclude that in $S'_i: n_{i+2}$ is unmarked. From Lemma 23,

$$\text{Remove1}.\text{unlock}(n_{i+1}) < E \text{Remove2}.\text{lock}(n_{i+2}) < E \text{Remove2}.\text{marking}(n_{i+2}) \quad (20)$$

Hence we can conclude that,

$$\text{Contains}.\text{read}(n_i) < E n_{i+1}.\text{marking} < E n_{i+2}.\text{marking} \quad (21)$$

Now consider a state $S_k$ in which $n_{x-1}$ is unmarked. From the Lemma 35 we have

$$n_{x-1}.\text{marked} < E n_x.\text{marked} \quad (22)$$

From the Observation 22 and from the Equation 22 we have,

$$\exists S_k: (S_k.n_{x-1}.\text{marked} = \text{false}) \xrightarrow{\text{Observation} \ 22} S_k.n_x.\text{marked} = \text{false} \quad (23)$$

Let us call the state immediately after the marking of $n_{x-1}$ as $S'_k$ as below:

Let us call the state immediately after the marking of $n_{x-1}$ as $S'_k$ as below:

\[
\begin{array}{c|c|c|c|c}
\text{S}_k & \text{Add} & \text{S}'_k & \text{Remove} & \text{S}'_{k-1} \\
(n_{x-1}.\text{marked}) \land \neg(n_{x}.\text{marked}) & n_{x-1}.\text{marking} & (n_{x-1}.\text{marked}) \land \neg(n_{x}.\text{marked}) & n_x.\text{marking} & (n_{x-1}.\text{marked}) \land (n_{x}.\text{marked}) \land n_{x}.\text{next} \neq n_i) \\
\end{array}
\]

Figure 8: Contains(key, false) with no successful concurrent Add on key. $S'_{x-1}.n_{x-1}.\text{marked} = \text{true}$ and $S'_{x-1}.n_x.\text{marked} = \text{true}$ and node(key) $\notin S'_k.\text{AbS}$ at Line 59
Combining Observation 15 and 14, we know that,
\[ S'_{k'.n_{x-1}.next} = S'_{k.n_x} \] (24)

Also since \( n_{x-1}.marking \) is the only event between \( S_k \) and \( S'_k \), we can say that,
\[ S_{k.n_{x-1}.next} = S_{k.n_x} \] (25)

Also by observing the code of Contains method, we have the following:
\[ S'_{x-1.n_{x-1}.val} < key \quad \text{(Line 57 of the Contains method)} \] (26)
\[ S_{x.n_{x}.val} \geq key \quad \text{(Line 56 of the Contains method)} \] (27)
\[ S'_{x.n_{x}.val} > key \quad \text{(Line 59 of the Contains method)} \] (28)
\[ \neg (S_{k.n_{x-1}.marked}) \land \neg (S_{k.n_x.marked}) \quad \text{(by the Lemma 35)} \] (29)

Combining the equations 23, 25, 26 & 28, 29 and Observation 13 and 14,
\[ (S_{k.n_{x-1}.val} < key < S_{k.n_x.val}) \land (\neg S_{k.n_{x-1}.marked})\land \]
\[ (\neg S_{k.n_x.marked}) \land (S_{k.n_{x-1}.next} = S_{k.n_x}) \quad \text{\Rightarrow} \]
\[ (node(key) \notin S_k.AbS) \] (30)

Now since no concurrent Add happens between \( S_1 \) and \( S'_x \) we have that,
\[ node(key) \notin S''_x.AbS \] (31)

2. **Case 2**: \emph{key is present, but marked in the Pre-State of read(n.marked) event at Line 59 of Contains method, which is the LP of Contains (key, false).} We assume that there is no concurrent Add from \( S_1 \) until \( S'_x \).

![Figure 9: LP of Contains(key, false) with no successful concurrent Add is at read(n.val = key) at Line 59](image)

(a) **Given**: \( S'_{x-1.n_{x-1}.marked} = false \land S'_{x-1.n_x.marked} = true \)

**To Prove**: \( node(key) \notin S''_x.AbS \)

\[ S'_{x-1.n_{x-1}.val} < key \quad \text{(Line 57 of the Contains method)} \] (32)
\[ S_{x.n_{x}.val} \geq key \quad \text{(Line 56 of the Contains method)} \] (33)
\[ S'_{x.n_{x}.val} = key \quad \text{(Line 59 of the Contains method)} \] (34)
\[ S'_{x-1.n_x.marked} = true \quad \text{(Given)} \] (35)
\[ S''_{x}.n_{x}.marked = \text{true} \quad \text{(From Observation 14)} \]  
\[ S'_{x-1}.n_{x-1}.val < S'_{x-1}.n_{x-1}.next.val \quad \text{(from Lemma 24)} \]  
\[ S'_{x-1}.n_{x-1}.next = S'_{x-1}.n_{x} \quad \text{(Line 57 of the Contains method)} \]  
\[ S'_{x-1}.n_{x-1}.val < S'_{x-1}.n_{x}.val \quad \text{(from Equation 37 & 38)} \]

Combining the equations 32,34 & 39 and Observation 13,
\[ (S'_{x-1}.n_{x-1}.val < (\text{key} = S'_{x-1}.n_{x}.val)) \land (\text{key} \neq S'_{x-1}.n_{x-1}.val) \land (S'_{x-1}.n_{x}.marked) \implies \text{(node(key) \notin S''_{x-1}.AbS)} \]  

Now since no concurrent Add happens between \( S_{1} \) and \( S''_{x} \) we have that,
\[ \text{node(key)} \notin S''_{x}.AbS \]  

(b) **Given:** \( S'_{x-1}.n_{x-1}.marked = \text{true} \land S'_{x-1}.n_{x}.marked = \text{true} \)  
**To Prove:** \( \text{node(key)} \notin S''_{x}.AbS \)  
From given, we have that,
\[ (n_{x-1} \notin S'_{x-1}.AbS) \land (n_{x} \notin S'_{x-1}.AbS) \]  
From \( S'_{x-1}.n_{x-1} \) we backtrack the nodes until we find the first node \( n_{i} \) belonging to \( S'_{x-1}.AbS \). Therefore,
\[ n_{i} \in S'_{x-1}.AbS \]  
In the worst case, \( S'_{x-1}.n_{i} \) could be the Head node.

We know that,  
\[ (n_{i+1} \text{ to } n_{x}) \notin (S'_{x-1}.AbS) \]  
In the linked list of \( n_{1}, n_{i+1}, n_{i+2}, \ldots, n_{x-1}, n_{x} \) nodes, where \( n_{i+1}, n_{i+2}, \ldots, n_{x} \) are marked and consecutive, we can conclude (from Lemma 35) that,

\[ \text{Contains.read}(n_{1}) <_{E} \text{Contains.read}(n_{i}) <_{E} \] 
\[ \text{Remove.unlock}(n_{i+1}) <_{E} n_{i+2}.\text{marking} <_{E} \] 
\[ n_{i+3}.\text{marking} \ldots <_{E} n_{x-1}.\text{marking} <_{E} n_{x}.\text{marking} \] 
This implies that marking of \( n_{i+1} \) to \( n_{x} \) completes after \( \text{Contains(key, false)} \) started.

\[ \text{Contains.read}(n_{1}) <_{E} n_{x-1}.\text{marking} \]  

Now consider a state \( S'_{k+1} \) in which \( n_{x-1} \) was observed to be unmarked. Let us call the state immediately after the marking of \( n_{x} \) as \( S'_{k+1} \) as follows:

\[
\begin{array}{c}
\text{\( S'_{k+1} \)}
\end{array}
\]

Figure 10: \( \text{Contains(key, false)} \) with no successful concurrent Add. \( S'_{k+1}.n_{x}.marked = \text{true} \), \( \text{node(key)} \notin S''_{x}.AbS \) at Line 59  
LP of \( \text{Contains(key, false)} \) with no successful concurrent Add is at \( \text{read(n.val = key)} \) at Line 59  

Since a marked node remains marked (from Observation 14),
\[ S'_{k+1}.n_{x}.marked \implies S''_{x}.n_{x}.marked \]
Also by observing the code of Contains method, we have the following:

\[ S'_{x-1}.n_{x-1}.val < key \quad \text{(Line 57 of the Contains method)} \] (48)

\[ S_x.n_x.val \geq key \quad \text{(Line 56 of the Contains method)} \] (49)

\[ S'_x.n_x.val = key \quad \text{(Line 59 of the Contains method)} \] (50)

Combining the equations 50,47 & and from Observation 13 & 14,

\[ (S'_{k+1}.n_x.val = key) \land (S'_{k+1}.n_x.marked) \Rightarrow (node(key) \notin S'_{k+1}.AbS) \] (51)

Now since no concurrent Add happens between \( S_1 \) and \( S''_x \), we have that,

\[ node(key) \notin S''_x.AbS \] (52)

3. **Case 3**: key is not present in the Pre-State of the LP of Contains (key, false) method. LP is a dummy event inserted just before the LP of the Add. We assume that there exists a concurrent Add from \( S_1 \) until \( S'_x \).

![Figure 11: LP of Contains(key, false) with successful concurrent Add is at read(n.val = key)](image)

**To prove**: \( node(key) \notin S_{dummy}.AbS \)

From Lemma 31, we know that if Add returns true, then \( node(key) \) does not belong to the \( AbS \) in the pre-state of the LP of Add method. We add a dummy event just before this LP event of add method as in Figure 11.

\[ node(key) \notin S_{dummy}.AbS \] (53)

4. **Case 4**: key is present, but marked in the Pre-State of the LP of Contains (key, false) method. LP is a dummy event inserted just before the LP of the Add. We assume that there exists a concurrent Add from \( S_1 \) until \( S'_x \).

![Figure 12: LP of Contains(key, false) with successful concurrent Add is at read(n.val = key) at Line 59](image)

**To prove**: \( node(key) \notin S_{dummy}.AbS \)

From Lemma 31, we know that if Add returns true, then \( node(key) \) does not belong to the \( AbS \) in the pre-state of the LP of Add method. We add a dummy event just before this LP event of Add method as in Figure 12.

\[ node(key) \notin S_{dummy}.AbS \] (54)
Lemma 39 Consider a concurrent history $H$ and a sequential history $S$. Let $m_x, m_y$ be methods in $H$ and $S$ respectively. Suppose the following are true (1) The AbS in the pre-state of $m_x$’s LP in $H$ is the same as the AbS in the pre-state of $m_y$ in $S$; (2) The inv events of $m_x$ and $m_y$ are the same. Then (1) the resp event of $m_x$ in $H$ must be same as resp event of $m_y$ in $S$; (2) The AbS in the post-state of $m_x$’s LP in $H$ must be the same as the AbS in the post-state of $m_y$ in $S$. Formally, $\langle \forall m_x \in E^H.\text{mths}, \forall m_y \in E^S.\text{mths} : \langle \text{PreE}[E^H.m_x.\text{LP}].\text{AbS} = \text{PreM}[E^S.m_y].\text{AbS} \rangle \land (E^H.m_x.\text{inv} = E^S.m_y.\text{inv}) \implies \langle \text{PostE}[E^H.m_x.\text{LP}].\text{AbS} = \text{PostM}[E^S.m_y].\text{AbS} \rangle \land (E^H.m_x.\text{resp} = E^S.m_y.\text{resp}) \rangle$.

Proof.

Let us prove by contradiction. So we assume that,

\[
(\langle \text{PreE}[E^H.m_x.\text{LP}].\text{AbS} = \text{PreM}[E^S.m_y].\text{AbS} \rangle \land (E^H.m_x.\text{inv} = E^S.m_y.\text{inv}) \implies (E^H.m_x.\text{resp} \neq E^S.m_y.\text{resp}))
\] (55)

We have the following cases that $E^H.m_x.\text{inv}$ is invocation of either of these methods:

1. $m_x.\text{inv}$ is Add (key) Method:
   - $m_x.\text{resp} = \text{true}$: Given that the method $m_x.\text{resp}$ which is Add (key) returns true, we know that from the Lemma 31, $\langle \text{node(key)} \notin \text{PreE}[E^H.\text{Add(key, true)\text{.LP}}].\text{AbS} \rangle$. But since from assumption in equation 55, $(E^H.m_x.\text{resp} \neq E^S.m_y.\text{resp})$, $E^S.m_y.\text{resp}$ is false. However, from the Observation 30.1, if $\langle \text{node(key)} \notin \text{pre-state of LP of\text{ Add method}} \rangle$, then the Add(key,true) method must return true in $E^S$. This is a contradiction.
   - $m_x.\text{resp} = \text{false}$: Given that the method $m_x.\text{resp}$ which is Add (key) returns false, we know that from the Lemma 32, $\langle \text{node(key)} \in \text{PreE}[E^H.\text{Add(key, false)\text{.LP}}].\text{AbS} \rangle$. But since from assumption in equation 55, $(E^H.m_x.\text{resp} \neq E^S.m_y.\text{resp})$, $E^S.m_y.\text{resp}$ is false. However, from the Observation 30.2, if $\langle \text{node(key)} \in \text{pre-state of LP of\text{ Add method}} \rangle$, then the Add(key,false) method must return false in $E^S$. This is a contradiction.

2. $m_x.\text{inv}$ is Remove (key) Method:
   - $m_x.\text{resp} = \text{true}$: Given that the method $m_x.\text{resp}$ which is Remove (key) returns true, we know that from the Lemma 33, $\langle \text{node(key)} \in \text{PreE}[E^H.\text{Remove(key, true)\text{.LP}}].\text{AbS} \rangle$. But since from assumption in equation 55, $(E^H.m_x.\text{resp} \neq E^S.m_y.\text{resp})$, $E^S.m_y.\text{resp}$ is false. However, from the Observation 30.3, if $\langle \text{node(key)} \notin \text{pre-state of LP of\text{ Remove method}} \rangle$, then the Remove(key,true) method must return true in $E^S$. This is a contradiction.
   - $m_x.\text{resp} = \text{false}$: Given that the method $m_x.\text{resp}$ which is Remove (key) returns false, we know that from the Lemma 34, $\langle \text{node(key)} \notin \text{PreE}[E^H.\text{Remove(key, false)\text{.LP}}].\text{AbS} \rangle$. But since from assumption in equation 55, $(E^H.m_x.\text{resp} \neq E^S.m_y.\text{resp})$, $E^S.m_y.\text{resp}$
is false. However, from the Observation 30.4, if node(key) \notin pre-state of LP of Remove method, then the Remove(key, false) method must return false in E^S. This is a contradiction.

3. m_x.inv is Contains (key) Method:

- \( m_x.resp = true \): Given that the method \( m_x.resp \) which is \( \text{Contains}(key) \) returns true, we know that from the Lemma 36, node(key) \in PreE[H.Contains(key, true).LP].AbS. But since from assumption in equation 55, \( (E^H.m_x.resp \neq E^S.m_y.resp) \), \( E^S.m_y.resp \) is false. However, from the Observation 30.5, if node(key) \in pre-state of LP of Contains method, then the Contains(key, true) method must return true in E^S. This is a contradiction.

- \( m_x.resp = false \): Given that the method \( m_x.resp \) which is \( \text{Contains}(key) \) returns false, we know that from the Lemma 38, node(key) \notin PreE[H.Contains(key, false).LP].AbS. But since from assumption in equation 55, \( (E^H.m_x.resp \neq E^S.m_y.resp) \), \( E^S.m_y.resp \) is false. However, from the Observation 30.6, if node(key) \notin pre-state of LP of Contains method, then the Contains(key, false) method must return false in E^S. This is a contradiction.

Thus we conclude that the \( \text{resp} \) event of \( m_x \) in H must be same as \( \text{resp} \) event of \( m_y \) in S. Formally, \( \langle E^H.m_x.resp = E^S.m_y.resp \rangle \).

**Lemma 40** All histories H generated by the Lazy List are linearizable.

**Proof.** Lemma follows based on the Lemma 39, Lemma 9 and Lemma 10.

---

4.2 Hand-over-Hand Locking List

In this section we define the fine list data structure. It is implemented as a collection of a set of nodes. This is a linked list of node shown in the Node is a class and it has three fields, the val field is the key value of the node. The nodes are sorted in order of the val field. This helps efficiently detect when a node is absent in the list. The next field is a reference to the next node in the list. The lock field is for ensuring access to a shared node happens in a mutually exclusion manner. We say a thread acquires a lock and releases the lock when it executes a lock.acquire() and lock.release() method call respectively. Each thread acquires lock in a hoh-locking-list order. We assume the next field of the node is atomic.

```java
class Node{
    int val; // actual key of node
    Node next; // next Node in list
    Lock lock; //synchronizes individual Node
    /*
    For the sentinel Node the @param key should be min or max
    int value and for the usual Node @param key val be the
    actual element in list
    */
    Node(int key){
        val = key;
        next = null;
        lock = new Lock();
    }
};
```
We define concurrent set $S$, which is dynamically being modified by a fixed set of concurrent threads. In this setting, threads may perform insertion or deletion of nodes to the set. We used hoh-locking-list based set algorithm based on [8, Chap 9]. We assume that all the nodes have unique identification key.

| Method       | Return Value | Pre-state($S$: global state)                                                                 | Post-state($S'$: future state of $S$ such that $S \subseteq S'$) |
|--------------|--------------|--------------------------------------------------------------------------------------------|-----------------------------------------------------------------|
| HoHAdd($n$)  | true         | $S$: $(n \notin S.AbS)$                                                                     | $S'$: $(n \in S'.AbS)$                                           |
| HoHAdd($n$)  | false        | $S$: $(n \in S.AbS)$                                                                       | $S'$: $(n \notin S'.AbS)$                                        |
| HoHRemove($n$) | true     | $S$: $(n \in S.AbS)$                                                                       | $S'$: $(n \notin S'.AbS)$                                        |
| HoHRemove($n$) | false    | $S$: $(n \notin S.AbS)$                                                                     | $S'$: $(n \notin S'.AbS)$                                        |
| HoHContains($n$) | true     | $S$: $(n \in S.AbS)$                                                                       | $S'$: $(n \in S'.AbS)$                                           |
| HoHContains($n$) | false    | $S$: $(n \notin S.AbS)$                                                                     | $S'$: $(n \notin S'.AbS)$                                        |

### 4.2.1 Methods Exported & Sequential Specification

In this section, we describe the methods exported by the hoh-locking-list data structure.

1. The **HoHAdd**($n$) method adds a node $n$ to the list, returns *true* if the node is not present in the list else it returns *false*. This follows directly from our assumption that all the nodes are assigned distinct keys.

2. The **HoHRemove**($n$) method deletes a node $n$ from the list, if it is present and returns *true*. If the node is not in the list earlier, it returns *false*.

3. The **HoHContains**($n$) returns *true*, if the list contains the node $n$; otherwise returns *false*.

Table 2 shows the sequential specification, as the name suggests shows the behaviour of the list when all the methods are invoked sequentially. We defined each method formally in any given global state $S$ before the execution of the method and future state $S'$ after executing it sequentially. The *Pre-state* is the shared state before *inv* event and the *Post-state* is also the shared state just after the *resp* event of a method, which is depicted in the Figure 1.

All the fields in the structure are declared atomic. This ensures that operations on these variables happen atomically. In the context of a particular application, the node structure can be easily modified to carry useful data (like weights etc).

#### Notations used in PseudoCode:

\(\downarrow\), \(\uparrow\) denote input and output arguments to each method respectively. The shared memory is accessed only by invoking explicit *read()* and *write()* methods. The *flag* is a local variable which returns the status of each operation. We use nodes $n_1$, $n_2$, $n$ to represent node references.

### 4.2.2 Working of the methods of hoh-locking-list

We define all methods like hoh-locking-list used in the [8, Chap 9] with some modification. We add a new **HoHLocate** method, which helps to locate the location of the *key* in the list like lazy list defined in the SubSection 4.1. This **HoHLocate** method takes *key* as input and returns the corresponding pair of neighboring node \( (n_1, n_2) \) and both these nodes are locked and reachable from the Head. Initially $n_1$ and $n_2$ are set to *null*.

**Working of the HoHAdd (key) method:** When a thread wants to add a node to the list,
**Algorithm 6** HoHLocate Method: Takes key as input and returns the corresponding pair of neighboring node \((n_1, n_2)\). Initially \(n_1\) and \(n_2\) are set to null.

66: procedure HoHLocate (key ↓, \(n_1 \uparrow, n_2 \uparrow\))
67:     lock.acquire(Head);
68:     node \(n_1 = \) Head;
69:     node \(n_2 = n_1\).next;
70:     lock.acquire(\(n_2\));
71:     while (read(\(n_2\).val) < key) do
72:         lock.release(\(n_1\));
73:         \(n_1 \leftarrow n_2\);
74:         \(n_2 \leftarrow n_2\).next
75:     lock.acquire(\(n_2\));
76: end while
77: end procedure

**Algorithm 7** HoHContains Method: Returns **true** if key is part of the set and returns **false** otherwise.

78: procedure HoHContains (key ↓, flag ↑)
79:     HoHLocate(key ↓, \(n_1 \uparrow, n_2 \uparrow\));
80:     if (read(\(n_2\).val) = key) then
81:         flag ← true;
82:     else
83:         flag ← false;
84:     end if
85:     lock.release(\(n_1\));
86:     lock.release(\(n_2\));
87:     return;
88: end procedure

**Algorithm 8** HoHAdd Method: key gets added to the list if it is not already part of the list. Returns **true** on successful add and returns **false** otherwise.

89: procedure HoHAdd (key ↓, flag ↑)
90:     HoHLocate(key ↓, \(n_1 \uparrow, n_2 \uparrow\));
91:     if (read(\(n_2\).val) ≠ key) then
92:         write(\(n_3\), new node(key));
93:         write(\(n_3\).next, \(n_2\));
94:         flag ← true;
95:     else
96:         flag ← false;
97:     end if
98:     lock.release(\(n_1\));
99:     lock.release(\(n_2\));
100:    return;
101: end procedure

**Algorithm 9** HoHRemove Method: key gets removed from the list if it is already part of the list. Returns **true** on successful remove otherwise returns **false**.

103: procedure HoHRemove (key ↓, flag ↑)
104:     HoHLocate(key ↓, \(n_1 \uparrow, n_2 \uparrow\));
105:     if (read(\(n_2\).val) = key) then
106:         write(\(n_1\).next, \(n_2\).next);
107:         flag ← true;
108:     else
109:         flag ← false;
110:     end if
111:     lock.release(\(n_1\));
112:     lock.release(\(n_2\));
113:     return;
114: end procedure

it invokes HoHLocate in the Line 90. The HoHLocate traverses the list from Head by acquiring locks both predecessor and successor nodes until it finds a node with its key greater than or equal to key, say \(n_{curr}\) and it’s predecessor node, say \(n_{pred}\). When HoHLocate method returns, both the nodes are locked. Then it checks if \(read(n_{curr}.val) ≠ key\) is true(Line 91), then the thread adds the new node(key) between \(n_{pred}\) and \(n_{curr}\) in the list from the Line 92-94 and
returns true after unlocking the nodes. If the key is already present in the list, it returns false by unlocking the locked nodes. This is described in Algorithm 8.

**Working of the HoHRemove (key) method:** When a thread wants to delete a node from the list, it invokes HoHLocate in the Line 104. The HoHLocate traverses the list from Head by acquiring locks both predecessor and successor nodes until it finds a node with its key greater than or equal to key, say ncurr and its predecessor node, say npred. When HoHLocate method returns, both the nodes are locked. Then it checks if \((\text{read}(n_2.val) = \text{key})\) is true (Line 105), if it is then the thread removes the ncurr by changing the next pointer of npred to ncurr.next in the Line 106. If the key is not present in the list, it returns false by unlocking the locked nodes. This is described in Algorithm 9.

**Working of the HoHContains () method:** When a thread wants to search a node in the list, it invokes HoHLocate in the Line 79. The HoHLocate traverses the list from Head by acquiring locks both predecessor and successor nodes until it finds a node with its key greater than or equal to key, say ncurr and its predecessor node, say npred. When HoHLocate method returns, both the nodes are locked. Then it checks if \((\text{read}(n_2.val) = \text{key})\) is true (Line 80), if it is then the thread returns true in the Line 81. If the key is not present in the list, it returns false in the Line 83. This is described in Algorithm 7.

### 4.2.3 The LPs of the hoh-locking-list

Here, we list the linearization points (LPs) of each method of hoh-locking-list. Each method of the list can return either true or false. So, we define the LP for six methods:

1. **HoHAdd(key, true):** write\((n_1.next, n_3)\) in Line 94 of HoHAdd method.
2. **HoHAdd(key, false):** read\((n_2.val)\) in Line 91 of HoHAdd method.
3. **HoHRemove(key, true):** write\((n_1.next, n_2.next)\) in Line 106 of HoHRemove method.
4. **HoHRemove(key, false):** \((\text{read}(n_2.val))\) in Line 105 of HoHRemove method.
5. **HoHContains(key, true):** read\((n.val)\) in Line 80 of HoHContains method.
6. **HoHContains(key, false):** read\((n.val)\) in Line 80 of HoHContains method.

### 4.2.4 HoH-Locking-List Proof

In this subsection, we describe the lemmas to prove the correctness of concurrent hoh-locking-list structure.

Having defined a few notions on \(S\), we now define the notion of an abstract set, AbS for a global state \(S\) which we will use for guiding us in correctness of our methods and it is defined below:

**Definition 41** \(S.AbS \equiv \{ n | (n \in S.nodes) \land (S.Head \rightarrow^{*} S.n) \}\).

This definition of AbS captures the set of all nodes of AbS for the global state \(S\). It consists of all the nodes that are reachable from \(S.Head\).

**Observation 42** Consider a global state \(S\) which has a node \(n\). Then in any future state \(S'\) of \(S\), \(n\) is node in \(S'\) as well. Formally, \(\forall S, S': (n \in S.nodes) \land (S \sqsubseteq S') \Rightarrow (n \in S.nodes)\).

With this observation, we assume that nodes once created do not get deleted (ignoring garbage collection).
Observation 43 Consider a global state $S$ which has a node $n$ and it is initialized with key val. Then in any future state $S'$ the value of $n$ does not change. Formally, $\forall S, S': (n \in S.n\text{odes}) \land (S \sqsubseteq S') \Rightarrow (n \in S.n\text{odes}) \land (S.n.val = S'.n.val)$.

Corollary 44 There cannot exist two nodes with the same key in the $S.AbS$ of a particular global state $S$.

Observation 45 Consider a global state $S$ which is the post-state of return event of the method $HoHLocate(key)$ invoked in the $HoHAdd$ or $HoHRemove$ or $HoHContains$ methods. Suppose the $HoHLocate$ method returns $(n_1, n_2)$. Then in the state $S$, we have,

45.1 $\langle (n_1, n_2 \in S.n\text{odes}) \rangle$

45.2 $\langle (S.lock.acquire(n_1) = true) \land (S.lock.acquire(n_2) = true) \rangle$

45.3 $\langle S.n_1.next = S.n_2 \rangle$

Lemma 46 Consider the global state $S$ which is the post-state of return event of the method $HoHLocate(key)$ invoked in the $HoHAdd$ or $HoHRemove$ or $HoHContains$ methods. Suppose the $HoHLocate$ method returns references as $(n_1, n_2)$. Then in the state $S$, we have that $(S.n_1.val < key \leq S.n_2.val)$ for all nodes whose next $\neq$ null.

Proof. Line 67 of $HoHLocate$ method locks the $Head$, in Line 68 initialises $S.n_1$ to $Head$ and $S.n_2 = S.n_1.next$ in Line 69. In the last iteration of the while loop in the Line 71 the $S.n_1.val < S.n_2.val$ and from the Observation 43 we know that the node key does not change. So, before execution of Line 76, the $S.n_2.val \geq key$ and $S.n_1.val < S.n_2.val$ and $S.n_1, S.n_2$ are locked. Both nodes are belongs to $S.n\text{odes}$ and $S.n_1.val < key \leq S.n_2.val$. Also, from the Observations 45.2, 45.3 and 43 the nodes $n_1$ and $n_2$ are locked (do not change), and both are reachable from $Head$, hence, the lemma holds even when $HoHLocate$ returns.

\[\Box\]

Lemma 47 For a node $n$ in any global state $S$, we have that $\langle \forall n \in S.n\text{odes} \land n.next \neq null : S.n.val < S.n.next.val \rangle$.

Proof. We prove by induction on all events in $E_H$ that change the next field of the node.

Base condition: Initially, before the first event that changes the next field, we know that $(Head.key < Tail.key) \land (Head, Tail \in S.n\text{odes})$.

Induction Hypothesis: Say, upto $k$ events that change the next field of any node, $\forall n \in S.n\text{odes} \land n.next \neq null : S.n.val < S.n.next.val$.

Induction Step: So, by observation of the code, the $(k + 1)^{st}$ event which can change the next field can be only one of the following:

1. Line 93 of $HoHAdd$ method: Let $S_1$ be the state after the Line 91. We know that when $HoHLocate$ (Line 90) returns by the Observation 45, $S_1.n_1$ & $S_1.n_2$ are locked, $S_1.n_1.next = S_1.n_2$. By the Lemma 46 we have $(S_1.n_1.val \leq S_1.n_2.val)$. Also we know from Observation 43 that node value does not change, once initialised. To reach Line 93, $n_2.val \neq key$ in the Line 91 must evaluate to true. Therefore, $(S_1.n_1.val < key < S_1.n_2.val)$. So, a new node $n_3$ is created in the Line 92 with the value $key$ and then a link is added between $n_3.next$ and $n_2$ in the Line 93. So this implies $n_3.val < n_2.val$ even after execution of line 93 of $HoHAdd$ method.

\[\Box\]
2. Line 94 of HoHAdd method: By observing the code, we notice that the Line 94 (next field changing event) can be executed only after the HoHLocate method returns. From Lemma 46, we know that when HoHLocate returns then \( n_1.val < \text{key} \leq n_2.val \). To reach Line 94 of HoHAdd method, Line 91 should ensure that \( n_2.val \neq \text{key} \). This implies that \( n_1.val < \text{key} < n_2.val \). From Observation 45.3, we know that \( n_1.next = n_2 \). Also, the atomic event at Line 94 sets \( n_1.next = n_3 \) where \( n_3.val = \text{key} \).

Thus from \( n_1.val < n_3.val < n_2.val \) and \( n_1.next = n_3 \), we get \( n_1.val < n_1.next.val \). Since \((n_1, n_2) \in S\text{nodes}\) and hence, \( S.n_1.val < S.n_1.next.val \).

3. Line 106 of HoHRemove method:

Let \( S_1 \) and \( S_2 \) be the states after the Line 105 and Line 106 respectively. By observing the code, we notice that the Line 106 (next field changing event) can be executed only after the HoHLocate method returns. From Lemma 46, we know that when HoHLocate returns then \( S_1.n_1.val < \text{key} \leq S_1.n_2.val \). To reach Line 106 of HoHRemove method, Line 105 should ensure that \( S_1.n_2.val = \text{key} \). Also we know from Observation 43 that node value does not change, once initialised. This implies that \( S_2.n_1.val < (\text{key} = S_2.n_2.val) \). From Observation 45.3, we know that \( S_2.n_1.next = n_2 \). Also, the atomic event at line 106 sets \( S_2.n_1.next = S_2.n_2.next \).

We know from Induction hypothesis, \( S_2.n_2.val < S_2.n_2.next.val \). Thus from \( S_2.n_1.val < S_2.n_2.val \) and \( S_2.n_1.next = S_2.n_2.next \), we get \( S_2.n_1.val < S_2.n_1.next.val \). Since \((n_1, n_2) \in S\text{nodes}\) and hence, \( S.n_1.val < S.n_1.next.val \).

\( \Box \)

**Corollary 48** There cannot exist two nodes with the same key in the AbS of a particular global state \( S \).

**Corollary 49** Consider the global state \( S \) such that for a node \( n \), if there exists a key strictly greater than \( n.val \) and strictly smaller than \( n.next.val \), then the node corresponding to the key does not belong to \( S.AbS \). Formally, \( \{S, n, \text{key} : (S.n.val < \text{key} < S.n.next.val) \implies \text{node(}\text{key}) \notin S.AbS\} \).

**Lemma 50** In a global state \( S \), for any node \( n \), if it is in the list, then \( n \) is reachable from Head. Formally, \( \{S, n : (n \in S\text{nodes}) \implies (S.Head \rightarrow^* S.n)\} \).

**Proof.** We prove by Induction on events that change the next field of the node (as these affect reachability), which are Line 93 & 94 of HoHAdd method and Line 106 of HoHRemove method. It can be seen by observing the code that HoHLocate and HoHContains method do not have any update events.

**Base step:** Initially, before the first event that changes the next field of any node, we know that \( ((\text{Head}, \text{Tail} \in S\text{nodes}) \land (\text{Head} \rightarrow^* \text{Tail})) \).

**Induction Hypothesis:** We assume that the \( k^{th} \) event that changes the next field of some node reachable from the Head.

**Induction Step:** By observing the code, the \((k + 1)^{st}\) event can be one of the following events that change the next field of a node:

1. **Line 92 & 93 of HoHAdd method:** Let \( S_1 \) be the state after the Line 90. Line 92 of the HoHAdd method creates a new node \( n_3 \) with value \( \text{key} \). Line 93 then sets \( S_1.n_3.next = S_1.n_2 \). Since this event does not change the next field of any node reachable from the Head of the list, the lemma is not violated.
2. Line 94 of HoHAdd method: By observing the code, we notice that the Line 93 (next field changing event) can be executed only after the HoHLocate method returns. Let $S_1$ and $S_2$ be the states after the Line 91 and Line 94 respectively. From Observation 45.3, we know that when HoHLocate returns then $S_1.n_1.next = S_1.n_2$. From Line 92 & 93 of HoHAdd method, $(S_1.n_1.next = S_1.n_2) \land (S_1.n_3.next = S_1.n_2)$. It is to be noted that (From Observation 45.2), $S_1.n_1$ & $S_1.n_2$ are locked, hence no other thread can change the next field. Also from Observation 43, a node’s key field does not change after initialization. Before executing Line 94, $S_1.n_1$ is reachable from Head. After Line 94, node $S_2.n_3$ is also reachable from $S_1.n_1$. Thus, we know that $S_2.n_3$ is also reachable from Head. Formally, $(S_2.Head \rightarrow \ast S_2.n_1) \land (S_2.n_1 \rightarrow S_2.n_3) \implies (S_2.Head \rightarrow \ast S_2.n_3)$.

3. Line 106 of HoHRemove method: Let $S_1$ and $S_2$ be the states after the execution of Line 105 and Line 106 respectively. By observing the code, we notice that the Line 106 (next field changing event) can be executed only after the HoHLocate method returns. From Observation 45.2, we know that when HoHLocate returns then $S_1.n_1$ & $S_1.n_2$ are locked and $S_1.n_1$ is reachable from Head and from Line 106 of HoHRemove method $S_1.n_1.next = S_1.n_2.next$. As $S_1.n_1$ & $S_1.n_2$ are locked, no other thread can change $S_2.n_1.next$ and $S_2.n_2.next$. Also from Observation 43, a node’s key does not change after initialization. If $S_2.n_2.next$ is reachable from Head, then it continues to remain reachable. So this event does not violate the lemma.

Hence eventually, $\forall S_2, n : (n \in S_2.nodes) \implies (S_2.Head \rightarrow \ast S_2.n)$. □

**Lemma 51** Only the events write($n_1.next, n_3$) in 94 of HoHAdd method and write($n_1.next, n_2.next$) in 106 of HoHRemove method can change the AbS.

**Proof.** It is to be noted that the HoHLocate and HoHContains methods do not have any update events. By observing the code, it appears that the following (write) events of the HoHAdd and HoHRemove method can change the AbS:

1. Line 92 & 93 of HoHAdd method: In Algorithm 8, let $S_1.AbS$ be the initial state of the AbS, such that we know from Line 91 that $key \notin S_1.AbS$. Line 92 of the HoHAdd method creates a node $n_3$ with value $key$, i.e. $n_3.val = key$. Now, Line 93 sets $S_1.n_3.next = S_1.n_2$. Since this event does not change the next field of any node reachable from the Head of the list, hence from Definition 41, $S_1.AbS$ remains unchanged after these events.

2. Line 94 of HoHAdd method: Let $S_1$ and $S_2$ be the states after the Line 91 and Line 94 respectively. At line 91, true evaluation of the condition leads to the execution of $S_1.n_1.next = S_1.n_3$ at Line 94. Also, $S_1.n_1$ and $S_1.n_2$ are locked, therefore from Observation 45, Head $\rightarrow \ast S_1.n_1$. From line 93 & 94 we get: $S_1.n_1 \rightarrow S_1.n_3 \rightarrow S_1.n_2$. Hence, Head $\rightarrow S_1.n_1 \rightarrow S_1.n_3 \rightarrow S_1.n_2$ follows. We have (Head $\rightarrow S_2.n_3$). Thus from Definition 41, $S_1.AbS$ changes to $S_2.AbS = S_1.AbS \cup n_3$.

3. Line 106 of HoHRemove method: Let $S_1$ be the state after the Line 106. By observing the code, we notice that the state before execution of Line 106 satisfies that $key \in S_1.AbS$. After execution of line 106, AbS changes such that $key \notin S_1.AbS$. In Line 106 $S_1.n_1.next$ is set to $S_1.n_2.next$, $S_1.AbS$ remains unchanged follows from Definition 41.

Hence, only the events write($n_1.next, n_3$) in 94 of HoHAdd method and write($n_1.next, n_2.next$) in 106 of HoHRemove method can change the AbS. □
Corollary 52 Both these events write\((n_1 .\text{next}, n_3)\) in Line 94 of HoHAdd method and write\((n_1 .\text{next}, n_2 .\text{next})\) in Line 106 of HoHRemove method can change the AbS are also be the Linearization Points (LPs) of the respective methods.

Observation 53 Consider a sequential history S. Let S be a global state in S.allStates before the execution of the method and S’ be a global state just after the return of the method \((S \sqsubseteq S’)\). Then we have the sequential specification of all methods as follows,

53.1 For a given key, suppose node\((\text{key})\) \notin S.AbS. In this state, suppose HoHAdd \((\text{key})\) method is \((\text{sequentially})\) executed. Then the HoHAdd method will return true and node\((\text{key})\) will be present in S’.AbS. Formally, \(\forall S : (\text{node(\text{key})} \notin S.AbS) \xrightarrow{\text{seq-add}} S.HoHAdd(\text{key}, \text{true}) \land (S \sqsubseteq S’) \land (\text{node(\text{key})} \in S’.AbS)).\)

53.2 For a given key, suppose node\((\text{key})\) \in S.AbS. In this state, suppose HoHAdd \((\text{key})\) method is \((\text{sequentially})\) executed. Then the HoHAdd method will return false and node\((\text{key})\) will continue to be present in S’.AbS. Formally, \(\forall S : (\text{node(\text{key})} \in S.AbS) \xrightarrow{\text{seq-add}} S.HoHAdd(\text{key}, \text{false}) \land (S \sqsubseteq S’) \land (\text{node(\text{key})} \in S’.AbS)).\)

53.3 For a given key, suppose node\((\text{key})\) \in S.AbS. In this state, suppose HoHRemove \((\text{key})\) method is \((\text{sequentially})\) executed. Then the HoHRemove method will return true and node\((\text{key})\) will not be present in S’.AbS. Formally, \(\forall S : (\text{node(\text{key})} \in S.AbS) \xrightarrow{\text{seq-remove}} S.HoHRemove(\text{key}, \text{true}) \land (S \sqsubseteq S’) \land (\text{node(\text{key})} \notin S’.AbS)).\)

53.4 For a given key, suppose node\((\text{key})\) \notin S.AbS. In this state, suppose HoHRemove \((\text{key})\) method is \((\text{sequentially})\) executed. Then the HoHRemove method will return false and node\((\text{key})\) will continue to be not present in S’.AbS. Formally, \(\forall S : (\text{node(\text{key})} \notin S.AbS) \xrightarrow{\text{seq-remove}} S.HoHRemove(\text{key}, \text{false}) \land (S \sqsubseteq S’) \land (\text{node(\text{key})} \notin S’.AbS)).\)

53.5 For a given key, suppose node\((\text{key})\) \in S.AbS. In this state, suppose HoHContains \((\text{key})\) method is \((\text{sequentially})\) executed. Then the HoHContains method will return true and node\((\text{key})\) will continue to be present in S’.AbS. Formally, \(\forall S : (\text{node(\text{key})} \in S.AbS) \xrightarrow{\text{seq-contains}} S.HoHContains(\text{key}, \text{true}) \land (S \sqsubseteq S’) \land (\text{node(\text{key})} \in S’.AbS)).\)

53.6 For a given key, suppose node\((\text{key})\) \notin S.AbS. In this state, suppose HoHContains \((\text{key})\) method is \((\text{sequentially})\) executed. Then the HoHContains method will return false and node\((\text{key})\) will continue to be not present in S’.AbS. Formally, \(\forall S : (\text{node(\text{key})} \notin S.AbS) \xrightarrow{\text{seq-contains}} S.HoHContains(\text{key}, \text{false}) \land (S \sqsubseteq S’) \land (\text{node(\text{key})} \notin S’.AbS)).\)

Lemma 54 If some HoHAdd method returns true in E^H then

54.1 The node\((\text{key})\) is not present in the pre-state of LP event of the method. Formally, \(\langle \text{HoHAdd(\text{key}, \text{true})} \rangle \implies (\text{node(\text{key})} \notin (\text{PreE}[E^H .HoHAdd(\text{key}, \text{true}).LP].AbS)).\)

54.2 The node\((\text{key})\) is present in the post-state of LP event of the method. Formally, \(\langle \text{HoHAdd(\text{key}, \text{true})} \rangle \implies (\text{node(\text{key})} \in (\text{PostE}[E^H .HoHAdd(\text{key}, \text{true}).LP].AbS)).\)

Proof.

- **54.1:** From Line 90, when HoHLocate returns we know that from the Observation 45, nodes n_1 and n_2 are locked and \((n_1, n_2 \in S.nodes)\). Also, \(n_1 .\text{val} < \text{key} \leq n_2 .\text{val}\) from Lemma 46. Now in Line 91, \(n_2 .\text{val} \neq \text{key}\) is evaluated to true. Also from Corollary 49, we conclude that node\((\text{key})\) not in the state after HoHLocate returns. And from Observation 43, no node changes its key value after initialization. So, node\((\text{key})\) \notin S.AbS, where S is the pre-state of the LP event of the method. Hence, \(\langle \text{HoHAdd(\text{key}, \text{true})} \rangle \implies (\text{node(\text{key})} \notin (\text{PreE}[E^H .HoHAdd(\text{key}, \text{true}).LP].AbS)).\)
4 DATA-STRUCTURE SPECIFIC PROOFS

- **54.2:** From the Lemma 54.1 we get that node(key) is not present in the pre-state of the LP event. From Lemma 51, it is known that only LP event can change the S.AbS. Now after execution of the LP event i.e. write(n1.next, n2) in the Line 94, node(key) ∈ S’.AbS, where S’ is the post-state of the LP event of the method. Hence, ⟨HoHAdd(key, true) ⟩ ➝ (node(key) ∈ (PostE[E^H.HoHAdd(key, true).LP].AbS)).

**Lemma 55** If some HoHAdd method returns false in E^H then

55.1 The node(key) is present in the pre-state of LP event of the method. Formally, ⟨HoHAdd(key, false) ⟩ ➝ (node(key) ∈ (PreE[E^H.HoHAdd(key, false).LP].AbS)).

55.2 The node(key) is present in the post-state of LP event of the method. Formally, ⟨HoHAdd(key, false) ⟩ ➝ (node(key) ∈ (PostE[E^H.HoHAdd(key, false).LP].AbS)).

**Proof.**

- **55.1:** From Line 90, when HoHLocate returns we know that from the Observation 45, nodes n1 and n2 are locked and (n1, n2 ∈ S.nodes). Also, n1.val < key ≤ n2.val from Lemma 46. Now in Line 91, n2.val ≠ key is evaluated to false, means node (key) present. Also from Corollary 49, we conclude that node(key) not in the state after HoHLocate returns. And from Observation 43, no node changes its key value after initialization. So, node(key) ∈ S.AbS, where S is the pre-state of the LP event of the method. Hence, ⟨HoHAdd(key, false) ⟩ ➝ (node(key) ∈ (PreE[E^H.HoHAdd(key, false).LP].AbS)).

- **55.2:** From the Lemma 55.1 we get that node(key) is present in the pre-state of the LP event. This LP event n2.val ≠ key in Line 91 does not change the S.AbS, Now after execution of the LP event the node(key) also present in the S’.AbS, where S’ is the post-state of the LP event of the method. Hence, ⟨HoHAdd(key, false) ⟩ ➝ (node(key) ∈ (PostE[E^H.HoHAdd(key, false).LP].AbS)).

**Lemma 56** If some HoHRemove method returns true in E^H then

56.1 The node(key) is present in the pre-state of LP event of the method. Formally, ⟨HoHRemove(key, true) ⟩ ➝ (node(key) ∈ (PreE[E^H.HoHRemove(key, true).LP].AbS)).

56.2 The node(key) is not present in the post-state of LP event of the method. Formally, ⟨HoHAdd(key, true) ⟩ ➝ (node(key) ∈ (PostE[E^H.HoHAdd(key, true).LP].AbS)).

**Proof.**

- **56.1:** From Line 104, when HoHLocate returns we know that from the Observation 45, nodes n1 and n2 are locked and (n1, n2 ∈ S.nodes). Also, n1.val < key ≤ n2.val from Lemma 46. Now in Line 105, n2.val = key is evaluated to true, means node (key) is present. So, before execution of the LP event write(n1.next, n2.next) in the Line 106 node (key) is also present in the S.AbS and from the Observation 43, no node changes its key value after initialization. So, node(key) ∈ S.AbS, where S is the pre-state of the LP event of the method. Hence, ⟨HoHRemove(key, true) ⟩ ➝ (node(key) ∈ (PreE[E^H.HoHRemove(key, true).LP].AbS)).
4 DATA-STRUCTURE SPECIFIC PROOFS

• 56.2: From the Lemma 56.1 we get that node(key) is present in the pre-state of the LP event. This LP event \( \text{write}(n_1 .next, n_2.next) \) in the Line 106 changes the \( S.AbS \). Now after execution of the LP event the node(key) will not present in the \( S'.AbS \), where \( S' \) is the post-state of the LP event of the method. Hence, \( \langle \text{HoHRemove}(key, true) \rangle \Rightarrow (\text{node}(key) \notin (\text{PostE}[E^H . \text{HoHRemove}(key, true)].LP].AbS)) \).

\[ \square \]

Lemma 57 If some HoHRemove method returns false in \( E^H \) then

57.1 The node(key) is not present in the pre-state of LP event of the method. Formally, \( \langle \text{HoHRemove}(key, false) \rangle \Rightarrow (\text{node}(key) \notin (\text{PreE}[E^H . \text{HoHRemove}(key, false)].LP].AbS)) \).

57.2 The node(key) is not present in the post-state of LP event of the method. Formally, \( \langle \text{HoHRemove}(key, false) \rangle \Rightarrow (\text{node}(key) \notin (\text{PostE}[E^H . \text{HoHAdd}(key, false)].LP].AbS)) \).

Proof.

• 57.1: From Line 104, when HoHLocate returns we know that from the Observation 45, nodes \( n_1 \) and \( n_2 \) are locked and \( (n_1, n_2 \in S.nodes) \). Also, \( n_1.val < key \leq n_2.val \) from Lemma 46. Now in Line 105, \( n_2.val = key \) (the LP event \( \text{read}(n_2.val) \)) is evaluated to false, means node (key) is not present. So, before execution of the LP the node (key) is not present in the \( S.AbS \), where \( S \) is the pre-state of the LP event of the method. Hence, \( \langle \text{HoHRemove}(key, false) \rangle \Rightarrow (\text{node}(key) \in (\text{PreE}[E^H . \text{HoHRemove}(key, false)].LP].AbS)) \).

• 57.2: From the Lemma 57.1 we get that node(key) is not present in the pre-state of the LP event. This LP event \( \langle \text{read}(n_2.val) = key \rangle \) in the Line 43 does not change the \( S.AbS \). Now after execution of the LP event the node(key) will not present in the \( S'.AbS \), where \( S' \) is the post-state of the LP event of the method. Hence, \( \langle \text{HoHRemove}(key, false) \rangle \Rightarrow (\text{node}(key) \notin (\text{PostE}[E^H . \text{HoHRemove}(key, false)].LP].AbS)) \).

\[ \square \]

Lemma 58 If some HoHContains method returns true in \( E^H \) then

58.1 The node(key) is present in the pre-state of LP event of the method. Formally, \( \langle \text{HoHContains}(key, true) \rangle \Rightarrow (\text{node}(key) \in (\text{PreE}[E^H . \text{HoHContains}(key, true)].LP].AbS)) \).

58.2 The node(key) is present in the post-state of LP event of the method. Formally, \( \langle \text{HoHContains}(key, true) \rangle \Rightarrow (\text{node}(key) \in (\text{PostE}[E^H . \text{HoHContains}(key, true)].LP].AbS)) \).

Proof.

• 58.1: From Line 79, when HoHLocate returns we know from the Observation 45 that, nodes \( n_1 \) and \( n_2 \) are locked and \( (n_1, n_2 \in S.nodes) \). Also, \( n_1.val < key \leq n_2.val \) from Lemma 46. Now in Line 80, \( n_2.val = key \) (the LP event \( \text{read}(n_2.val) \)) is evaluated to true and this LP event does not change the \( S.AbS \). From Observation 43, no node changes its key value after initialization. So, \( \text{node}(key) \in S.AbS \), where \( S \) is the pre-state of the LP event of the method. Hence, \( \langle \text{HoHContains}(key, true) \rangle \Rightarrow (\text{node}(key) \in (\text{PreE}[E^H . \text{HoHContains}(key, true)].LP].AbS)) \).
• **58.2:** From the Lemma 58.1 we get that node(key) is present in the pre-state of the LP event. This LP event (read(n2.val) = key) in the Line 80 does not change the S.Abs. Now after execution of the LP event the node(key) will be present in the S’.Abs, where S’ is the post-state of the LP event of the method. Hence, \((HoHContains(key, true) \implies (node(key) \notin (PostE[E^H.HoHContains(key, true)].LP].Abs))\).

\[ \square \]

**Lemma 59** If some HoHContains method returns false in \(E^H\) then

59.1 The node(key) is not present in the pre-state of LP event of the method. Formally, \(<HoHContains(key, false) \implies (node(key) \notin (PreE[E^H.HoHContains(key, false)].LP].Abs))\).

59.2 The node(key) is not present in the post-state of LP event of the method. Formally, \(<HoHContains(key, false) \implies (node(key) \notin (PostE[E^H.HoHContains(key, false)].LP].Abs))\).

**Proof.** Similar argument as Lemma 58.

\[ \square \]

**Lemma 60** Consider a concurrent history \(H\) and a sequential history \(S\). Let \(m_x, m_y\) be methods in \(H\) and \(S\) respectively. Suppose the following are true (1) The AbS in the pre-state of \(m_x\)’s LP in \(H\) is the same as the AbS in the pre-state of \(m_y\) in \(S\); (2) The inv events of \(m_x\) and \(m_y\) are the same. Then (1) the resp event of \(m_x\) in \(H\) must be same as resp event of \(m_y\) in \(S\); (2) The AbS in the post-state of \(m_x\)’s LP in \(H\) must be the same as the AbS in the post-state of \(m_y\) in \(S\). Formally, \(<\forall m_x \in E^H.mths, \forall m_y \in E^S.mths : (PreE[E^H.x.LP].AbS = PreM[E^S.y].AbS) \land (E^H.x.inv = E^S.y.inv) \implies (PostE[E^H.x.LP].AbS = PostM[E^S.y].AbS) \land (E^H.x.resp = E^S.y.resp)\>.

**Proof.** Let us prove by contradiction. So we assume that,

\[ ((PreE[E^H.m_x.LP].AbS = PreM[E^S.m_y].AbS) \land (E^H.m_x.inv = E^S.m_y.inv) \implies (E^H.m_x.resp \neq E^S.m_y.resp)) \quad (56) \]

We have the following cases that \(E^H.m_x.inv\) is invocation of either of these methods:

1. \(m_x.inv\) is HoHAdd (key) Method:
   - \(m_x.resp = true\): Given that the method \(m_x.resp\) which is HoHAdd (key) returns true, we know that from the Lemma 54, node(key) \(\notin PreE[E^H.Add(key, true)].LP].AbS\). But since from assumption equation 56, \((E^H.m_x.resp \neq E^S.m_y.resp)\), \(E^S.m_y.resp\) is false. However, from the Observation 53.1, if node(key) \(\notin\) pre-state of LP of HoHAdd method, then the HoHAdd(key, true) method must return true in \(E^S\). This is a contradiction.
   - \(m_x.resp = false\): Given that the method \(m_x.resp\) which is HoHAdd (key) returns false, we know that from the Lemma 55, node(key) \(\in PreE[E^H.HoHAdd(key, false)].LP].AbS\). But since from assumption in equation 56, \((E^H.m_x.resp \neq E^S.m_y.resp)\), \(E^S.m_y.resp\) is false. However, from the Observation 53.2, if node(key) \(\in\) pre-state of LP of HoHAdd method, then the HoHAdd(key, false) method must return false in \(E^S\). This is a contradiction.

2. \(m_x.inv\) is HoHRemove (key) Method:
• $m_x.resp = \text{true}$: Given that the method $m_x.resp$ which is $HoHRemove(\text{key})$ returns true, we know that from the Lemma 56, $node(key) \in PreE[E^H.HoHRemove(key, true).LP].AbS$. But since from assumption in equation 56, $(E^H.m_x.resp \neq E^S.m_y.resp)$, $E^S.m_y.resp$ is false. However, from the Observation 53.3, if $node(key) \notin$ pre-state of LP of $HoHRemove$ method, then the $HoHRemove(key, true)$ method must return true in $E^S$. This is a contradiction.

• $m_x.resp = \text{false}$: Given that the method $m_x.resp$ which is $HoHRemove(\text{key})$ returns false, we know that from the Lemma 57, $node(key) \notin PreE[E^H.HoHRemove(key, false).LP].AbS$. But since from assumption in equation 56, $(E^H.m_x.resp \neq E^S.m_y.resp)$, $E^S.m_y.resp$ is false. However, from the Observation 53.4, if $node(key) \in$ pre-state of LP of $HoHRemove$ method, then the $HoHRemove(key, false)$ method must return false in $E^S$. This is a contradiction.

3. $m_x.inv$ is $HoHContains(\text{key})$ Method:

• $m_x.resp = \text{true}$: Given that the method $m_x.resp$ which is $HoHContains(\text{key})$ returns true, we know that from the Lemma 58, $node(key) \in PreE[E^H.HoHContains(key, true).LP].AbS$. But since from assumption in equation 56, $(E^H.m_x.resp \neq E^S.m_y.resp)$, $E^S.m_y.resp$ is false. However, from the Observation 53.5, if $node(key) \in$ pre-state of LP of $HoHContains$ method, then the $HoHContains(key, true)$ method must return true in $E^S$. This is a contradiction.

• $m_x.resp = \text{false}$: Given that the method $m_x.resp$ which is $HoHContains(\text{key})$ returns false, we know that from the Lemma 59, $node(key) \notin PreE[E^H.HoHContains(key, false).LP].AbS$. But since from assumption in equation 56, $(E^H.m_x.resp \neq E^S.m_y.resp)$, $E^S.m_y.resp$ is false. However, from the Observation 53.6, if $node(key) \notin$ pre-state of LP of $HoHContains$ method, then the $HoHContains(key, false)$ method must return false in $E^S$. This is a contradiction.

Thus we conclude that the resp event of $m_x$ in $H$ must be same as resp event of $m_y$ in $S$. Formally, $(E^H.m_x.resp = E^S.m_y.resp)$.

\[\square\]

**Lemma 61** All histories $H$ generated by the hoh-locking-list are linearizable.

**Proof.** Proof follows based on the Lemma 60, Lemma 9 and Lemma 10. \[\square\]

5 Discussion & Conclusion

CDSs offer great performance benefits over their sequential counterparts. But one of the greatest challenges with CDSs is developing correct structures and then proving their correctness either through automatic verification or through hand-written proofs [4]. We believe that the techniques which help prove correctness of CDSs can also guide in developing new CDSs.

Several techniques have been proposed for proving linearizability- a correctness-criterion for concurrent objects. But LPs continue to remain most popular way of illustrating correctness of CDS among practitioners since it is seems intuitive and constructive. One of the main challenges with the LP based approach is to identify the correct LPs of a CDS. Identifying the correct LPs can be deceptively wrong in many cases. In fact in many cases, the LP identified or even worse the CDS could be wrong.

Considering the complexity of developing a CDS and verifying its correctness, we address the most basic problem of this domain in this paper: given the set of LPs of a CDS, how to
show its correctness? We assume that we are given a CDS and its LPs. We have developed a hand-crafted technique of proving correctness of the CDSs by validating it LPs. We believe that our technique can be applied to prove the correctness of several commonly used CDSs developed in literature such as Lock-free Linked based Sets [17], lazy-list [6, 8], Skiplists [18] etc. Our technique will also work for CDSs in which the LPs of a method might lie outside the method such as lazy-list. To show the efficacy of this technique, we show the correctness of lazy-list and hand-over-hand locking list (hoh-locking-list) [2, 8].

As a part of our technique, we have identified a generic lemma (Lemma 8). We show that any CDS for which this lemma is true and satisfies our assumptions on the LPs, is linearizable. Thus, we would like to view this lemma as an abstract class in a language like C++. It is specific to each CDS and has to be proved (like instantiation of the abstract class in C++). In Section 4, we demonstrate this technique by giving a high-level overview of the correctness of this lemma for lazy-list and of hoh-locking-list.

In Section 3, we postulated that the hand-crafted mechanism of proving the generic lemma for a given CDS might bring out errors in the LPs proposed if they are incorrect. Further, we also theorized that this technique might give new insights for designing new CDSs. But the actual details of these can be accomplished are still not clear. Ideally, a programmer should have a set of design patterns using which s/he would be able to develop correct CDS which are also efficient. As observed earlier, this has been acknowledged as a very complicated problem. We believe that we have just scratched the surface of this problem in this paper. We plan to explore further in this direction as a part of future work.

To this end, Transactional Memory Systems [7] or TMs have been proposed as an alternative to address this challenge of designing efficient concurrent structures. But the design of efficient CDS using TMs would again require the programmer to designate portions of code as transactions. Not doing this properly could again lead to loss in efficiency and/or correctness. Hence, we believe that the TMs can help with this objective although they may not be the final solution. As a part of our future work, we also plan to explore how TMs can help us achieve the objective.

An important point to be noted with our approach: we assumed that only LP events change the AbDS (Assumption 4). Although this is true in case of many CDSs considered, this is not always true. As an example consider a shared array which has an lock for each entry and is modified by multiple threads concurrently. Threads wishing to update several entries in a linearizable manner can obtain locks on the relevant entries of the array using two-phase locking (2PL) and then perform the updates. In this case, one can choose any event between the last locking and the first unlocking as the LP. But then, the LP event is not where all the updates to the shared entries of the array takes place. So with this kind of 2PL usage, our technique will not directly work. In that case, we believe that we have to consider the notion of Linearization Blocks instead of Linearization Points. We plan to explore this notion in future. On the other hand, we believe that our technique will work for those CDSs which has at least one wait-free method (like the contains method in the case of lazy-list).

References

[1] Daphna Amit, Noam Rinetzyk, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In CAV, volume 4590 of Lecture Notes in Computer Science, pages 477–490. Springer, 2007.

[2] Rudolf Bayer and Mario Schkolnick. Concurrency of operations on b-trees. Acta Informatica, 9:1–21, 1977.

[3] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable refinement checking for concurrent objects. In POPL, pages 651–662. ACM, 2015.
REFERENCES

[4] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Verifying linearisability with potential linearisation points. In *Proceedings of the 17th International Conference on Formal Methods*, FM’11, pages 323–337, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] J He, C A R Hoare, and J W Sanders. Data refinement refined. In *Proc. Of the European Symposium on Programming on ESOP 86*, pages 187–196, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[6] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4):411–424, 2007.

[7] Maurice Herlihy and J. Eliot B.Moss. Transactional memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[8] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[9] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[10] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[11] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. On correctness of data structures under reads-write concurrency. In *DISC*, volume 8784 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2014.

[12] Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A constructive approach for proving data structures’ linearizability. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2015.

[13] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009.

[14] Peter W. O’Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC ’10, pages 85–94, New York, NY, USA, 2010. ACM.

[15] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010.

[16] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP*, pages 129–136. ACM, 2006.

[17] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC ’95, pages 214–222, New York, NY, USA, 1995. ACM.

[18] V. Luchangco Y. Lev, M. Herlihy and N. Shavit. A provably correct scalable skiplist (brief announcement). In *Proc. of the 10th International Conference On Principles Of Distributed Systems (OPODIS 2006)*, 2006.

[19] Shao Jie Zhang. Scalable automatic linearizability checking. In *ICSE*, pages 1185–1187. ACM, 2011.
[20] He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In CAV (2), volume 9207 of Lecture Notes in Computer Science, pages 3–19. Springer, 2015.