A Tutorial on Verifying `LinkedList` using KeY

Hans-Dieter A. Hiep[0000−0001−9677−6644] **, Jinting Bian,
Frank S. de Boer, and Stijn de Gouw

CWI, Science Park 123, 1098 XG Amsterdam, The Netherlands
{`hdh,j.bian,frb,stijn.de.gouw`}`@cwi.nl`

Abstract. This is a tutorial paper on using KeY to demonstrate formal verification of state-of-the-art, real software. In sufficient detail for a beginning user of JML and KeY, the specification and verification of part of a corrected version of the `java.util.LinkedList` class of the Java Collection framework is explained. The paper includes video material that shows recordings of interactive sessions, and project files with solutions. As such, this material is also interesting for the expert user and the developer of KeY as a 'benchmark' for specification and (automatic) verification techniques.

Keywords: Program correctness, linked list, theorem proving, KeY

1 Introduction

Software libraries are the building blocks of many programs that run on the devices of many users every day. The functioning of a system may rely for a large part on its used software libraries. A small error present in a heavily-used software library could lead to serious unwanted outcomes, such as system outages and failures. Using informal root cause analysis [16], one could find from a system failure its root causes, which may include programming errors. But root cause analysis can only be applied *after* a failure has happened. To prevent certain failures from happening in the first place, program correctness is of the utmost importance. Although establishing program correctness seems to be an expensive activity, it may be worthwhile for critical software libraries, such as the standard library that all programs rely on.

This tutorial intends to show how we take an existing Java program that is part of the Java standard library, and study it closely to increase our understanding of it. If we are only interested in showing the presence of an issue with the program, e.g. that it lacks certain functionality, it suffices to show an example run which behaves unexpectedly. But to reach the conclusion that no unexpected behavior ever results from running the program, first requires a precise specification of what behavior one expects, and further requires a convincing argument that all possible executions of the program exhibit that behavior.

** Corresponding author: `hdh@cwi.nl`

We take a formal approach to both specification and reasoning about program executions, allowing us to increase the reliability of our reached conclusions to near certainty. In particular, the specifications we write are expressed in the Java Modeling Language (JML), and our reasoning is tool-supported and partially automated by KeY. To the best of the authors' knowledge, KeY is the only tool that supports enough features of the Java programming language for reasoning about real programs, of which its run-time behavior crucially depends on the presence of features such as: dynamic object creation, exception handling, integer arithmetic with overflow, `for` loops with early returns, nested classes (both static and non-static), inheritance, polymorphism, erased generics, etc.

As a demonstration of applying KeY to state-of-the-art, real software, we focus on Java's `LinkedList` class, for two reasons. First, a (doubly-)linked list is a well-known basic data structure for storing and maintaining unbounded data, and has many applications: for example, in Java's secure sockets implementation.[1] Second, it has turned out that there is a 20-year-old bug lurking in its program, that might lead to security-in-depth issues on large memory systems, caused by the overflow of a field that caches the length of the list [12]. Our specification and verification effort is aimed at establishing the absence of this bug from a repaired program.

This article is based on the results as described in another closely related paper [12]. That paper provides a high-level overview on the specification and verification effort of the linked list class as a whole, for a more general audience. In the present article, more technical details on how we use the KeY theorem prover are given, and we give more detail concerning the production of the proofs. In particular, this tutorial consists of the on-line repository of proof files [11], and on-line video material that shows how to (re)produce the proofs [10]: these are video recordings of the interactive sessions in KeY that demonstrates exactly what steps one could take to complete the correctness proofs of the proof obligations generated by KeY from the method contracts.

We see how to set-up a project and configure the KeY tool (Section 2). We then study the source code of the `LinkedList` to gain an intuitive understanding of its structure: how the instances look like, and how the methods operate (Section 3). We formulate, based on previous intuition, a *class invariant* in JML that expresses a property that is true of every instance (Section 4). An interesting property that follows from the class invariant, that is used as a separation principle, is described next (Section 5). To keep this presentation reasonably short, we further focus on the methods which pose the main challenges to formal verification, `add` and `remove`. We give a *method contract* for the `add` method that describes its expected behavior and we verify that its implementation is correct (Section 6). The difficulty level increases after we specify the `remove` method (Section 7), as its verification requires more work than for `add`. We study a deeper method that `remove` depends on, and finally use *loop invariants* to prove the correctness of `remove`. We conclude with some proof challenges for the reader's recreation of further specifications and (in)formal proofs (Section 8).

[1] E.g. see the JVM internal class `sun.security.ssl.HandshakeStateManager`.

2 Preliminaries

```
linkedlist-tutorial
 ├── key-2.6.3.zip
 ├── LinkedList.key
 ├── src
 │    └── java
 │         └── util
 │              ├── LinkedList.old
 │              └── LinkedList.java
 ├── jre
 │    └── java
 │         └── ...
 └── proof
      └── ...
```

Table 1: Directory structure of project files. The `src` directory contains the Java classes we want to specify and verify. The `jre` directory contains stub files, with specifications of unrelated classes. The `LinkedList.java` file is the source file we end up with after following this tutorial. The `proof` directory contains the completed proofs.

First we set-up the project files needed to use KeY. The project files are available on-line [11]: these can be downloaded and include the KeY software version that we use. After unpacking the project files, we end up with the directory structure of Table 1.

The original source file of `LinkedList.old` was obtained from OpenJDK version `jdk8-b132`. The original has been pre-processed: generic class parameters are removed, and all methods and nested classes irrelevant to this tutorial are removed. Both the removal of generics and the stub files in the `jre` folder were generated automatically, using the Eclipse extensions for KeY. Repeating those steps is not necessary here.

Over the course of the next sections, we modify the original source file and add annotations to formally specify its behavior, and helper methods for presenting intermediary lemmas. The annotations are usual Java comments, and thus ignored when the file is read by a Java compiler. The helper methods introduce slight performance overhead (of calling a method that performs no operations, and immediately returning from it); it is clear that these do not change the original behavior of the program.

2.1 KeY Settings Set-up

To produce proofs in KeY, the first step is to set-up KeY's proof strategy and taclet options. This has to be done only once, as these taclet settings are stored per computer user. Sometimes, KeY overwrites or corrupts these settings if different versions are used. To ensure KeY starts in a fresh state, one can remove

Max. rule applications	1000
Stop at	Default
Proof splitting	Delayed
Loop treatment	Invariant
Block treatment	Contract
Method treatment	Contract
Dependency contract	On
Query treatment	Off
Expand local queries	On
Arithmetic treatment	Basic
Quantifier treatment	No splits
Class axiom rule	Off
Auto induction	Off
User-defined taclets	All off

(a) Proof search strategy

JavaCard	Off
Strings	On
Assertions	Safe
BigInt	On
Initialization	Disable static ...
Int Rules	Java semantics
Integer Simpl. Rules	Full
Join Generate	Off
Model Fields	Treat as axiom
More Seq. Rules	On
Permissions	Off
Program Rules	Java
Reach	On
Runtime Exceptions	Ban
Sequences	On
Well-def. Checks	Off
Well-def. Operator	L

(b) Taclet options

Table 2

the `.key` directory from the user's home directory, and clean out preferences from the `.java/.userPrefs` directory by deleting the `de/uka/ilkd/key` hierarchy containing `prefs.xml` files.[2] Now start up KeY, and the example selection screen appears (if not, selecting File ▷ Load Example opens the same screen). Load any example, to enter proof mode.

First, we set-up a proof strategy: this ensures the steps as done in the videos can be reproduced. On the left side of the window, change the settings in the Proof Search Strategy tab to match those of Table 2a. We ensure to use particular taclet rules that correctly model Java's integer overflow semantics. Select Options ▷ Taclet Options, and configure the options as in Table 2b. The taclet options become effective after loading the next problem. We do that now: the main proof file `LinkedList.key` can be loaded, and a Proof Management window opens up, showing a class hierarchy and its methods. A method is not shown when no specifications are present. After giving specifications below, more methods can be selected in this window.

3 Structure and Behavior of `java.util.LinkedList`

In this section we walk through part of the source code of Java's linked list: see the `LinkedList.old` file. Over the course of this tutorial, we add annotations at the appropriate places. We finally obtain the `LinkedList.java` file.

[2] On Windows, the preferences are instead stored in the Windows Registry. Use the `regedit` tool and clean out under `HKEY_CURRENT_USER\Software\JavaSoft\Prefs` or `HKEY_CURRENT_USER\Software\Wow6432Node\JavaSoft\Prefs` the same hierarchy. On Mac OS, open a terminal, change directory to `~/Library/Preferences` and delete `de.uka.ilkd.plist`

```
package java.util;

public class LinkedList {

    transient int size = 0;
    transient Node first;
    transient Node last;

    public LinkedList() {}
```

Listing 1: The `LinkedList` class fields and constructor (begin of file).

Our `LinkedList` class has three attributes and a constructor (Listing 1): a `size` field, which stores a cached number of elements in the list, and two fields that store a reference to the first and last `Node`. The public constructor contains no statements: thus it initializes `size` to zero, and `first` and `last` to `null`.

The linked list fields are declared transient and package private. The transient flag is not relevant to our verification effort. The reason the fields are declared package private seems to prevent generating accessor methods by the Java compiler. However, in practice, the fields are treated as if they were private.

```
    private static class Node {
        Object item;
        Node next;
        Node prev;

        Node(Node prev, Object element, Node next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }

}
```

Listing 2: The `Node` nested class fields and constructor (end of file).

The `Node` class is defined as a private static nested class to represent the containers of items stored in the list (Listing 2). A static nested private class behaves like a top-level class, except that it is not visible outside the enclosing class. The nodes are doubly-linked, that is, each node is connected to its preceding node (through field `prev`) and succeeding node (through field `next`). These fields contain `null` in case no preceding or succeeding node exists. The data itself is contained in the `item` field of a node.

```
    // implements java.util.Collection.add
    public boolean add(Object e) {
        linkLast(e);
        return true;
    }
```

Listing 3: The method `add`.

The method `add` for adding elements to the list, takes one argument, the item to add (Listing 3). The informal Java documentation for `Collection` specifies it always returns `true`. The implementation immediately calls `linkLast`.

```
    // implements java.util.Collection.remove
    public boolean remove(Object o) {
        if (o == null) {
            for (Node x = first; x != null; x = x.next) {
                if (x.item == null) {
                    unlink(x);
                    return true;
                }
            }
        } else {
            for (Node x = first; x != null; x = x.next) {
                if (o.equals(x.item)) {
                    unlink(x);
                    return true;
                }
            }
        }
        return false;
    }
```

Listing 4: The method `remove`.

The method `remove` for removing elements, also takes one argument, the item to remove (Listing 4). If that item was present in the list, then its first occurrence is removed and `true` is returned; otherwise, if the item was not present, then the list is not changed and `false` is returned.

Presence of the item depends on whether the argument of the remove method is `null` or not. If the argument is `null`, then it searches for the first occurrence of a `null` item in the list. Otherwise, it uses the `equals` method, that every `Object` in Java has, to determine the equality of the argument with respect to the contents of the list. The first occurrence of an item that is considered equal by the argument is then returned. In both cases, the execution walks over the linked list, from the first node until it has reached the end. In the case that the node was found that contains the first occurrence of the argument, an internal method is called: `unlink`, and afterwards `true` is returned.

Observe that the linked list is not modified if `unlink` is not called. Although not immediately obvious, this requires that the `equals` method of every object cannot modify our current linked list, for the duration of the call to `remove`. When the `remove` method is called with an item that is not contained in the list, either loop eventually exits, and `false` is returned.

```
void linkLast(Object e) {
    final Node l = last;
    final Node newNode = new Node(l, e, null);
    last = newNode;
    if (l == null) first = newNode;
    else l.next = newNode;
    size++;
}
```

Listing 5: The internal method `linkLast`.

The internal method `linkLast` changes the structure of the linked list (Listing 5). After performing this method, a new node has been created, and the `last` field of the linked list now points to it. To maintain structural integrity, also other fields change: if the linked list was empty, the `first` field now points to the new, and only, node. If the linked list was not empty, then the new last node is also reachable via the former last node's `next` field. It is always the case that all items of a linked list are reachable through the `first` field, then following the `next` fields, and also for the opposite direction.

```
Object unlink(Node x) {
    final Object element = x.item;
    final Node next = x.next;
    final Node prev = x.prev;
    if (prev == null) {first = next;}
    else {
        prev.next = next;
        x.prev = null;
    }
    if (next == null) {last = prev;}
    else {
        next.prev = prev;
        x.next = null;
    }
    x.item = null;
    size--;
    return element;
}
```

Listing 6: The internal method `unlink`.

The internal method `unlink` is among the most complex methods that alters the structure of a linked list (Listing 6). The method is used only when the linked list is not empty. Its argument is a node, necessarily one that belongs to the linked list. We first store the fields of the node in local variables: the old item, and next and previous node references.

After the method returns, the argument fields are all cleared, presumably to help the garbage collector. However, other fields also change: if the argument is the first node, the `first` field is updated; if the argument is the last node, the `last` field is updated. The predecessor or successor fields `next` and `prev` of other nodes are changed to maintain the integrity of the linked list: the successor of the unlinked node becomes the successor of its predecessor, and the predecessor of the unlinked node becomes the predecessor of its successor.

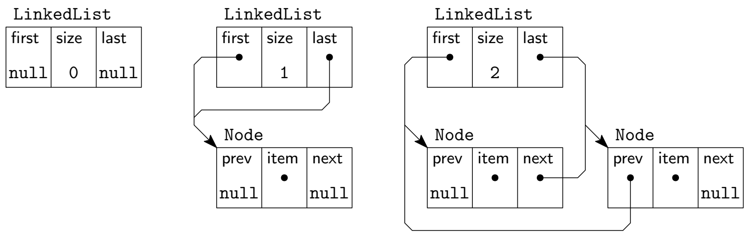

Fig. 1: Three example linked lists: empty, with a chain of one node, and with a chain of two nodes. Items themselves are not shown.

3.1 Expected and Unexpected Method Behavior

We draw pictures of linked list instances to understand better how the structure looks like over time. In Fig. 1, we see three linked list instances. The left-most linked list is an object without any items: its `size` is zero. When we perform `add` with some item (it is not important what item), a new node is created and the first and last pointers are changed to point to the new node. Now the `prev` and `next` fields of the new node are `null`, indicating that there is no other node before and there is no other node after it. Also, the `size` field is increased by one. Adding another item further creates another node, that is linked up to the previous node properly; the `last` field is then pointed to the newly created node. In the third instance, suppose we would perform `remove` with the first item. We would then have to unlink the node, see the code of `unlink` in Listing 6: the new value of `first` becomes the value of `next` which is the last node, and the value of `prev` of the succeeding node becomes the value of `prev` of the node that is unlinked, which is `null`. We thus end up in a similar situation as the second instance (except for the item that may be different). Removing the last item brings us back into the situation depicted by the first instance.

An important aspect of the implementation of our linked list is the cached `size` field: it represents the number of nodes that form a chain between `first` and `last`. It turns out an overflow may happen under certain conditions [12]. Consider two facts: Java integer primitives are stored in signed 32-bit registers, and it is possible to create a chain that is larger than the maximum positive value that can be stored in such fields, $2^{31} - 1$. Now, the cached size and the actual size no longer correspond. In the methods we have seen above, this seems to be no issue. But another method of the linked list may be used to demonstrate the key problem: `toArray`. The intention of `toArray` is to give back an array containing all the items of the list (see Listing 7). There are two problems: after the overflow has occurred and the `size` is negative, the `toArray` throws an unexpected `NegativeArraySizeException`. Also, after adding more items that brings the size back to a positive integer, e.g. adding $2^{32} + 1$ items in total, the array is of the wrong (positive) size and cannot contain *all* items of the list, and an `IndexOutOfBoundsException` is thrown.

```
public Object[] toArray() {
    Object[] result = new Object[size];
    int i = 0;
    for (Node x = first; x != null; x = x.next)
        result [i++] = x.item;
    return result;
}
```

Listing 7: The method `toArray` has unexpected behavior.

3.2 Verification Goal

We thus revise the source code and add a method that implements an overflow check (see Listing 8). The intention is that the overflow is signalled before it occurs, by throwing an exception. This ensures that the integrity of the linked list is always maintained. We modify the `add` method, and perform a call to this `checkSize()` method before invoking `linkLast`.

```
// new method, not in original LinkedList
private void checkSize() {
    if (size == Integer.MAX_VALUE)
        throw new IllegalStateException (...);
}
```

Listing 8: A new internal method `checkSize`.

Our aim in this tutorial is to keep the discussion general enough, without losing interesting particular details. We apply step-wise refinements to our arguments, where we start with a higher-level intuition and drill-down on technical details as they become relevant. The reader can always see the video material; but, a high-level intuition seems essential for following along.

Our specification and verification goals comprise two points:

1. Specification captures the ‘intended’ behavior of the methods with respect to its structural properties: in particular, we abstract away from all properties pertaining to the contents of the linked list.

2. Verification ensures the overflow bug no longer happens in the revised linked list: the actual number of nodes and the cached size are always the same.

The first point depends on the aim of a verification attempt. Are we using the specification to verify correctness of clients of the linked list? Then properties of the contents of a linked list are essential. But, for our purpose of showing the absence of an overflow bug, we abstract away some properties of the contents of linked lists. However, this abstraction is not yet fully complete, and the complete abstraction from contents is left as a proof challenge to the reader (see Section 8).

The second point deserves an introduction: how can we be sure that in every linked list the number of nodes and the value stored in the `size` field are the same? Can we keep the number of nodes bounded by what a Java integer can represent? Keeping this number in a ghost field is not sufficient, since the number of nodes depends on the success of a `remove` call: removing an item not present in the linked list should not affect its size, while removing an item that is present decreases its size by one. We refine: we could keep track of the items that are stored by the linked list. The structure to collect these items cannot be a set of items, since we could have duplicate items in the list. A multiset of items, the contents of a linked list, seems right: the size field of the linked list must be the same as the size of its contents, and the remove method is only successful if its argument was contained before the call.

However, working with multisets is quite unnatural, as the `remove` method removes the first item in the list. That can be refined by specifying the contents as a sequence of items instead. Although this could work in principle, a major difficulty when verifying the `remove` method is to give an argument as to why the method terminates. This requires knowledge of the linking structure of the nodes. We could relate the sequence of items to a traversal over nodes, saying that the first item of a linked list is found in the node by traversing `first`, and the item at index $0 < i < n$ is found in the node by traversing `first` and then `next` for $i - 1$ times. Formalizing this seems quite difficult, and as we shall see, not even possible in first-order logic.

Hence, we end up with our last refinement: we keep a sequence of nodes in a ghost field. From this sequence, one obtains the sequence of items. We relate the sequence of nodes to the linked list instance and require certain structural properties to hold of the nodes in the sequence. The length of this sequence is the actual number of nodes, that we show to be equal to the cached size.

4 Formulating a Class Invariant

We now formalize a class invariant, thereby characterizing all linked list instances. We focus on unbounded linked lists first, as these are the structures we intend to model: so most properties are expressed using KeY's unbounded integer type \bigint. Only at the latest we restrict the size of each linked list to a maximum, as a limitation imposed by the implementation. The setting in which to do our characterization is multi-sorted first-order logic. This logic is presented in a simplified form, leaving out irrelevant details (such as the heap): the full logic used by KeY is described in Chapter 2 of [13].

Consider the following sorts, or type symbols: *LinkedList* for a linked list, *Node* for a node, *Object* for objects and *Null* for `null` values. We have a type hierarchy, where *Null* is related to *LinkedList* and *Node*, and *LinkedList* and *Node* are both related to *Object*. This means that any object of sort *Null* is also an object of sort *LinkedList* and *Node*. Moreover, every object that is a linked list is also of type *Object*, and similar for nodes. We have the following signature: $\mathit{first} : \mathit{LinkedList} \to \mathit{Node}$ and $\mathit{last} : \mathit{LinkedList} \to \mathit{Node}$ for the `first` and `last` fields of linked lists, and $\mathit{prev} : \mathit{Node} \to \mathit{Node}$, $\mathit{item} : \mathit{Node} \to \mathit{Object}$, and $\mathit{next} : \mathit{Node} \to \mathit{Node}$ for the `prev`, `item` and `next` fields of nodes. Further, we assume there is exactly one object of *Null* sort, which is the `null` constant, for which above functions are left undefined: `null` is a valid object of the `LinkedList` and `Node` Java types, but one may not access its fields.

We search for an axiomatization that characterizes linked list instances. One can find these axioms by trial and error. We start listing some (obvious) axioms:

1. $\forall x^{\mathit{LinkedList}}; (x \neq \mathtt{null} \to (\mathit{first}(x) \neq \mathtt{null} \leftrightarrow \mathit{last}(x) \neq \mathtt{null}))$
   Every linked list instance either has both first and last set to `null`, or both point to some (possibly different) node.
2. $\forall x^{\mathit{LinkedList}}; (x \neq \mathtt{null} \to (\mathit{first}(x) \neq \mathtt{null} \to \mathit{prev}(\mathit{first}(x)) = \mathtt{null}))$
   The predecessor of the first node of a linked list is set to `null`.
3. $\forall x^{\mathit{LinkedList}}; (x \neq \mathtt{null} \to (\mathit{last}(x) \neq \mathtt{null} \to \mathit{next}(\mathit{last}(x)) = \mathtt{null}))$
   The successor of the last node of a linked list is set to `null`.
4. $\forall x^{\mathit{Node}}; (x \neq \mathtt{null} \to (\mathit{prev}(x) \neq \mathtt{null} \to \mathit{next}(\mathit{prev}(x)) = x))$
   Every node that has a predecessor, must be the successor of that predecessor.
5. $\forall x^{\mathit{Node}}; (x \neq \mathtt{null} \to (\mathit{next}(x) \neq \mathtt{null} \to \mathit{prev}(\mathit{next}(x)) = x))$
   Every node that has a successor, must be the predecessor of that successor.

These axioms are not yet sufficient: consider a linked list, in which its first and last nodes are different and both have neither a predecessor nor a successor. This linked list should not occur: intuitively, we know that the nodes between first and last are all connected and should form a doubly-linked ‘chain’. Moreover, for every linked list, this chain is necessarily finite: one can traverse from first to last by following the next reference a finite number of times. This leads to a logical difficulty.

Proposition 1. *It is not possible to define the reachability of nodes of a linked list in first-order logic.*

Proof. Let x be a linked list and y a node: there is no formula $\phi(x, y)$ that is true if and only if $\mathit{next}^i(\mathit{first}(x)) = y$ for some integer $i \geq 0$.[3] Suppose towards contradiction that there is such a formula $\phi(x, y)$. Now consider the infinite

[3] next^i is not a function symbol in first-order logic but an abbreviation of a finite term built by iteration of i times next, where $\mathit{next}^0(x) = x$ and $\mathit{next}^i(x) = \mathit{next}(\mathit{next}^{i-1}(x))$ for all $i > 0$.

set Δ of first-order formulas $\{\phi(x,y)\} \cup \{\neg(\mathit{next}^i(\mathit{first}(x)) = y) \mid 0 \leq i\}$. Let Γ be an arbitrary finite subset of Δ. Consider that there must exists some j such that $\Gamma \subseteq \{\phi(x,y)\} \cup \{\neg(\mathit{next}^i(\mathit{first}(x)) = y) \mid 0 \leq i < j\}$, so we can construct a linked list with j nodes, and we interpret x as that linked list and y as the last node. Clearly $\phi(x,y)$ is true as the last node is reachable, and all $\neg(\mathit{next}^i(\mathit{first}(x)) = y)$ is true for all $0 \leq i < j$ because j is not reachable within i steps from the first node. Since Γ is arbitrary, we have established that all finite subsets of Δ have a model. By compactness, Δ must have a model too. However, that is contradictory: no such model for Δ can exists, as neither $\phi(x,y)$ and $\mathit{next}^i(\mathit{first}(x)) \neq y$ for all integers $0 \leq i$ can all be true. □

We extend our signature to include other sorts: sequences and integers. These sorts are interpreted in the standard model. A schematic rule to capture integer induction is included in KeY (see [13, Section 2.4.2]). Sequences (see [13, Chapter 5.2]) have a non-negative integer length n, and consist of an element at each position $0 \leq i < n$. We write $\sigma[i]$ to mean the ith element of sequence σ, and $\ell(\sigma)$ to mean its length n.

Intuitively, each linked list consists of a sequence of nodes between its first and last node. Let $\mathrm{instanceof}_{Node}$: Object be a built-in predicate that states that the object is not `null` and of sort *Node*. A *chain* is a sequence σ such that:

(a) $\forall i^{\mathrm{int}}; (0 \leq i < \ell(\sigma) \to \mathrm{instanceof}_{Node}(\sigma[i]))$
All its elements are nodes and not `null`
(b) $\forall i^{\mathrm{int}}; (0 < i < \ell(\sigma) \to \mathit{prev}(\sigma[i]) = \sigma[i-1])$
The predecessor of node at position i is the node at position $i - 1$
(c) $\forall i^{\mathrm{int}} : (0 \leq i < \ell(\sigma) - 1 \to \mathit{next}(\sigma[i]) = \sigma[i+1])$
The successor of node at position i is the node at position $i + 1$

Let $\phi(\sigma)$ denote the above property that σ is a chain. If $\ell(\sigma) = 0$ then $\phi(\sigma)$ is vacuously true: the empty sequence is thus a chain. We now describe properties $\psi_1(\sigma, x)$ and $\psi_2(\sigma, x)$ that relate a chain σ to a linked list x. These denote the following intuitive properties: there is no first and last node and the chain is empty, or the chain is not empty and the first and last node are the first and last elements of the chain.

$$\psi_1(\sigma,x) \equiv (\ell(\sigma) = 0 \wedge \mathit{first}(x) = \mathit{last}(x) = \texttt{null})$$
$$\psi_2(\sigma,x) \equiv (\ell(\sigma) > 0 \wedge \mathit{first}(x) = \sigma[0] \wedge \mathit{last}(x) = \sigma[\ell(x) - 1])$$

6. $\forall x^{LinkedList}; (x \neq \texttt{null} \to \exists \sigma^{\mathrm{sig}}; (\phi(\sigma) \wedge (\psi_1(\sigma,x) \vee \psi_2(\sigma,x))))$
Every linked list necessitates the existence of a chain of either property

Further, we require that the size field of the linked list and the length of the chain are the same: this property is essential to our verification goal. The size field is modeled by the function *size* : *LinkedList* → *int*, and we require its value (1) to equal the length of the chain, and (2) to be bounded by the maximum value stored in a 32-bit integer. In formulating above properties in JML, we skolemize the existential quantifier using a ghost field: see Listing 9.

This has the additional benefit that we can easily refer to the chain ghost field in specifications.

```
/*@ nullable @*/ transient Node first;
/*@ nullable @*/ transient Node last;
//@ private ghost \seq nodeList;
/*@ invariant
  @   nodeList.length == size &&
  @   nodeList.length <= Integer.MAX_VALUE &&
  @   (\forall \bigint i; 0 <= i < nodeList.length;
  @        nodeList[i] instanceof Node) &&
  @   ((nodeList == \seq_empty && first == null && last == null)
  @     || (nodeList != \seq_empty && first != null &&
  @            first.prev == null && last != null &&
  @            last.next == null && first == (Node)nodeList[0] &&
  @            last == (Node)nodeList[nodeList.length-1])) &&
  @   (\forall \bigint i; 0 < i < nodeList.length;
  @        ((Node)nodeList[i]).prev == (Node)nodeList[i-1]) &&
  @   (\forall \bigint i; 0 <= i < nodeList.length-1;
  @        ((Node)nodeList[i]).next == (Node)nodeList[i+1]);
  @*/
```

Listing 9: The class invariant of LinkedList expressed in JML.

The class invariant is implicitly required to hold for the this object when invoking methods on a linked list instance. In particular, for the constructor of the linked list, the class invariant needs to be established after it returns. In Listing 10, we state that the constructor always constructs a linked list instance for which its chain is empty. The proof of the correctness follows easily: at construct time, the fields (including the ghost field) of the linked list instance are initialized with their default values. This means the size is zero, and the first and last references are null, and the ghost field is the empty sequence.

```
/*@
  @ public normal_behavior
  @   ensures nodeList == \seq_empty;
  @*/
public LinkedList() {}
```

Listing 10: The method contract of the constructor of LinkedList in JML.

For verifying the constructor above, see the video [1, 0:23–0:53], where the relevant video material is between timestamps 0:23 and 0:53.

5 The Acyclicity Property

An interesting consequence of the class invariant is the property that traversal of only next fields is acyclic. In other words, following only next references of any node that is present in a chain never reaches itself. The acyclicity property implies there is a number of times to follow the `next` reference until the last node is reached. For the last node, this number is zero (the last node is already reached). A symmetric property holds for `prev` too.

We logically specify the acyclicity property as follows. Let σ be the chain of a non-empty linked list x for which the class invariant holds. The following holds:

$$\forall i^{\text{int}}; (0 \leq i < \ell(\sigma) - 1 \to \forall j^{\text{int}}; (i < j < \ell(\sigma) \to \sigma[i] \neq \sigma[j]))$$

Let n abbreviate $\ell(\sigma)$. By contradiction: assume there are two indices, $0 \leq i < j < n$, such that the nodes $\sigma[i]$ and $\sigma[j]$ are equal. Then it must hold that for all k such that $j \leq k < n$, the node $\sigma[k]$ is equal to the node $\sigma[k-(j-i)]$: by induction on k. Base case: if $k = j$, then node $\sigma[j]$ and node $\sigma[j-(j-i)]$ are equal by assumption, since $\sigma[j-(j-i)] = \sigma[i]$. Induction step: suppose node at $\sigma[k]$ is equal to node at $\sigma[k-(j-i)]$. We must show if $k+1 < n$ then node $\sigma[k+1]$ equals node $\sigma[k+1-(j-i)]$. This follows from the fact that $\sigma[k+1] = \mathit{next}(\sigma[k])$ and $\sigma[k+1-(j-i)] = \mathit{next}(\sigma[k-(j-i)])$ for $k < n-1$, since σ is a chain and the chain property (c) of last section. Now we have established, for all $j \leq k < n$, node $\sigma[k]$ equals node $\sigma[k-(j-i)]$. In particular, this holds when k is $n-1$, the index of the last node: so we have $\sigma[n-1] = \sigma[n-1-(j-i)]$. Since the difference $(j-i)$ is positive, we know $\sigma[n-1-(j-i)]$ is not the last node. By the linked list property 3 we have $\mathit{next}(\mathit{last}(x)) =$ `null` and by $\psi_2(\sigma, x)$ we have $\mathit{last}(x) = \sigma[n-1]$: so we have $\mathit{next}(\sigma[n-1]) =$ `null`. By the chain properties (c) and (a) we have $\mathit{next}(\sigma[n-1-(j-i)]) = \sigma[n-(j-i)]$ and $\mathrm{instanceof}_{\mathit{Node}}(\sigma[n-(j-i)])$, respectively. From the latter we know $\sigma[n-(j-i)] \neq$ `null`. So we have $\mathit{next}(\sigma[n-1-(j-1)]) \neq$ `null`. But this is a contradiction: if nodes $\sigma[n-1]$ and $\sigma[n-1-(j-i)]$ are equal then their `next` fields must also have equal values, but $\mathit{next}(\sigma[n-1]) =$ `null` and $\mathit{next}(\sigma[n-1-(j-i)]) \neq$ `null`!

```
/*@ private normal_behavior
  @ requires true;
  @ ensures (\forall \bigint i;
  @     0 <= i < (\bigint)nodeList.length - (\bigint)1;
  @   (\forall \bigint j; i < j < nodeList.length;
  @     nodeList[i] != nodeList[j]));
  @*/
private /*@ strictly_pure @*/ void lemma_acyclic() {}
```

Listing 11: The method of a lemma added to `LinkedList` expressed in JML.

For verifying the lemma as formalized in Listing 11, see the video [2].

6 The `add` Method

Due to the revision of the source code, the add method now calls `checkSize` first (see Listing 8) to ensure that the size field does not overflow when we add another item. This means that the add method has two expected behaviors: the normal behavior when the length of the linked list is not yet at its maximum, and the exceptional behavior when the length of the linked list is at its maximum.

In the normal case, we expect the `add` method to add the given argument as an item to the linked list. Thus the sequence of nodes must become larger. We further specify the position where the item is added: at the end of the list. If add return normally, it returns true. In the exceptional case, we expect that an exception is thrown. We formalize the contract for `add` in Listing 12.

```
/*@
  @ public normal_behavior
  @   requires nodeList.length + (\bigint)1 <= Integer.MAX_VALUE;
  @   ensures
  @      nodeList == \seq_concat(\old(nodeList),
  @         \seq_singleton(nodeList[nodeList.length-1])) &&
  @      ((Node)nodeList[nodeList.length-1]).item == e &&
  @      \result;
  @ public exceptional_behavior
  @   requires nodeList.length == Integer.MAX_VALUE;
  @   signals_only IllegalStateException;
  @   signals (IllegalStateException e) true;
  @*/
public boolean add(/*@ nullable @*/ Object e) {
    checkSize(); // new
    linkLast(e);
    return true;
}
```

Listing 12: The `add` method with its method contract expressed in JML.

Since the `add` method calls the deeper methods `checkSize` and `linkLast`, we may employ their method contracts when verifying this method. So, before we verify `add`, we specify and verify these methods first.

We expect `checkSize` to throw an exception if the length of the linked list is too large to add another element, and it returns normally otherwise: see Listing 13 for its specification. Verification of `checkSize` in both normal and exceptional cases is done automatically by KeY, as can be seen in [1, 0:54–1:24].

```
/*@
  @ private exceptional_behavior
  @   requires nodeList.length == Integer.MAX_VALUE;
  @   signals_only IllegalStateException;
  @   signals (IllegalStateException e) true;
  @ private normal_behavior
  @   requires nodeList.length != Integer.MAX_VALUE;
  @   ensures true;
  @*/
```

Listing 13: The method contract in JML of the `checkSize` method.

For the `linkLast` method, we assume that the length of the linked list is smaller than its maximum length, so we can safely add another node without causing an overflow of the size field. When adding a new node, the resulting chain now is an extension of the previous chain, and additionally the class invariant holds afterwards—this is an implicit post-condition. Since we modify the chain, we need a `set` annotation that changes the ghost field.

```
/*@
  @ normal_behavior
  @   requires
  @     nodeList.length + (\bigint)1 <= Integer.MAX_VALUE;
  @   ensures
  @     nodeList == \seq_concat(\old(nodeList),
  @       \seq_singleton(nodeList[nodeList.length−1])) &&
  @     ((Node)nodeList[nodeList.length−1]).item == e;
  @*/
void linkLast(/*@ nullable @*/ Object e) {
    final Node l = last;
    final Node newNode = new Node(l, e, null);
    last = newNode;
    if (l == null) first = newNode;
    else l.next = newNode;
    size++;
    //@ set nodeList = \seq_concat(nodeList,\seq_singleton(last));
}
```

Listing 14: The `linkLast` method with its method contract expressed in JML.

The verification of this method is no longer fully automatic, see [1, 1:25–6:52].

Observe that there are two different situations we have to deal with: either the linked list was empty, or it was not. If the linked list was empty, then `last` is `null`, and we not only set the `last` field but also the `first`. Otherwise, if the linked list was not empty, we update the former last node to set its `next` field. The challenge is to prove that the class invariant holds after these heap updates, knowing that the class invariant holds in the before heap. The main insight is that the creation of a new node does not alias with any of the existing nodes, and that the modification of the `next` field only affects the old last node. Intuitively, we have a proof situation with two heaps as depicted in Fig. 2.

The properties (b) on page 12, that fixes `prev` fields to point to the previous node in the sequence, and (c), that fixes `next` fields to point to the next node in the sequence, of the chain are the remaning goals in [1, 3:58]. Proving (b) is straightforward if one makes a distinction between old nodes and the new node. Proving (c) in the 'heap after' involves two cases: either the index is between 0 and less than $\ell(\sigma) - 2$, or it used to be the last node and now has index $\ell(\sigma) - 2$. In the former case, the heap update has no effect, as we can show that these nodes are separate from the old last node because they differ in the old value of the next field. In the latter case, the heap update can be used to prove the property directly.

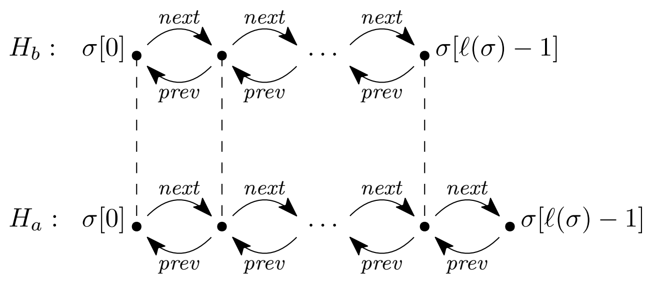

Fig. 2: The heap before (H_b) consists of an arbitrary chain of nodes. In the heap after (H_a) the dashed lines show which objects are identical to the heap before. The old last node at $\sigma[\ell(\sigma)-1]$ has a different value for its *next* field in the heap after: this must be the result of a heap update. The new last node is not created in the heap before: indeed, it is the result of creating a new node.

Finally, we can verify the `add` method: see [1, 6:58] for the normal behavior case, and [1, 8:09] for the exceptional behavior case.

7 The `remove` Method

The `remove` method takes as argument an object; it searches the linked list for the first node which contains the argument as an item. If found, it unlinks the node from the linked list. Using this intuition, we specify the `remove` method contract. Like with the `add` method, there is a deeper method that is called, `unlink`, which we have to specify and verify first.

An immediate difficulty in specifying the `remove` method contract is that its intended behavior depends on the behavior of the `Object.equals` method. Namely, the informal Java documentation states that the first element occurrence in the list that is 'equal to' the argument must be removed. Equality can be user-defined by overriding the `equals` method! We solve this difficulty by assuming a method contract for the equality method, see Listing 15.

```
/*@ public normal_behavior
 @   requires true;
 @   ensures \result == self.equals(param0);
 @*/
public /*@ helper strictly_pure @*/
boolean equals(/*@ nullable @*/ Object param0);
```

Listing 15: The `equals` stub method with its method contract expressed in JML.

We declare the equality method to be strictly pure, which implies that it must be a side-effect free and terminating method (see [13, Section 7.3.5]). Each strictly pure method is also directly accessible as an observer symbol (a function symbol) that can be used in specifications (see [13, Section 8.1.2]). However, no obvious relation between the possibly overridden equality method and its observer symbol is present. The intention of the contract given in Listing 15 is to relate the outcome of the method call of `equals` to the observer symbol *equals*, and this furthermore requires that the implementation is deterministic.

The ramifications of adding this assumed contract are not clear. We note that there are Java classes for which equality is not terminating under certain circumstances. Even `LinkedList` itself does not have a terminating equality, where two linked lists that contain each other may lead to a `StackOverflowError` when testing their equality. This example is described in the Javadoc [14] of the linked list: "Some collection operations which perform recursive traversal of the collection may fail with an exception for self-referential instances where the collection directly or indirectly contains itself." Another approach is to specify the outcome of equality as referential equality only, e.g. see [15, Section 4.4].

Now we can specify the behavior of `remove`. It can be seen as consisting of two cases: either its result is `true` and it has removed the first item equal to its argument from the list, or its result is `false` and the argument was not found and thus not removed. In the first case, the number of elements decreases by one. In the second case, the number of elements in the linked list remains unchanged. See Listing 16.

```
/*@
  @ public normal_behavior
  @   requires true;
  @   ensures \result == false ==>
  @     (\forall \bigint i; 0 <= i < \old(nodeList.length);
  @     (o==null ==> \old(((Node)nodeList[i]).item) != null) &&
  @     (o!=null ==> !\old(o.equals(((Node)nodeList[i]).item)))) &&
  @     nodeList == \old(nodeList);
  @   ensures \result == true ==>
  @     (\exists \bigint j; 0 <= j < \old(nodeList.length);
  @       (\forall \bigint i; 0 <= i < j;
  @       (o==null ==> \old(((Node)nodeList[i]).item) != null) &&
  @       (o!=null ==> !\old(o.equals(((Node)nodeList[i]).item)))) &&
  @     nodeList == \seq_concat(\old(nodeList)[0..j],
  @       \old(nodeList)[j+1..\old(nodeList.length)]) &&
  @     (o==null ==> \old(((Node)nodeList[j]).item) == null) &&
  @     (o!=null ==> \old(o.equals(((Node)nodeList[j]).item))));
  @*/
```

Listing 16: The `remove` method contract expressed in JML.

It is important to note that we make use of JML's `\old` operator to refer to the equality observer symbol in the old heap. Using equality in the new heap is a different observation; and it should not be possible to verify the remove method in this case. To see why, consider two linked list instances x and y: we add x to itself, and to y we add x and then y. Now we perform the `remove` operation on y with y as argument. Clearly x and y are not equal, because they have a different length. But the second item is y itself, and y equals y, so it is removed: see Fig. 3. In the resulting heap, both x and y contain x as only item: thus, x and y are equal. If we would observe equality in the new heap, then the implementation is incorrect: the item to remove should not be the second but the first!

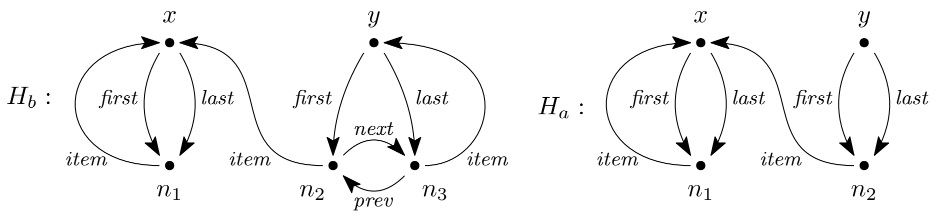

Fig. 3: The situation (H_b) before `remove` is invoked on y with argument y, and after (H_a). The result of this operation is that y's second node n_3 is unlinked, hence the first node n_2 becomes the last node and its next pointer is cleared: now x and y are equal because they have the same length, and they have the same item, namely x.

Before we can verify the `remove` method, we must specify and verify its deeper method: `unlink`. Within the method of unlink we have to update the chain ghost field as well, to remove a node from the sequence, so we add a set annotation to the method body. Additionally, we make use of the lemma `lemma_acyclic` by calling it as a first statement of the method. See Listing 17. This method call is also not present in the original definition for `unlink`, but we already argued that it does not affect behavior.

```
/*@ normal_behavior
  @   requires nodeList != \seq_empty &&
  @      0 <= nodeIndex < nodeList.length &&
  @      (Node)nodeList[nodeIndex] == x;
  @   ensures \result == \old(x.item) &&
  @      nodeList == \seq_concat(\old(nodeList)[0..nodeIndex],
  @         \old(nodeList)[nodeIndex+1..\old(nodeList).length]) &&
  @      nodeIndex == \old(nodeIndex);
  @*/
/*@ nullable @*/ Object unlink(Node x) {
    lemma_acyclic(); // new
    //@ set nodeList = \seq_concat(\dl_seqSub(nodeList,0,nodeIndex),
\dl_seqSub(nodeList,nodeIndex+1,\dl_seqLen(nodeList)));
    // rest of method body
    ...
```

Listing 17: The first part of method `unlink` and its method contract. Note that the @set annotation must not contain a new line, but kept on a single line: otherwise KeY 2.6.3 cannot load the source file. Here, /*@ @*/ does not work.

An interesting aspect of the specification of `unlink` is its use of a *ghost parameter*. Although KeY does not directly support ghost parameters, we are able to work around that by adding the parameter as a ghost field to our class:

//@ private ghost \bigint nodeIndex;

Its value is left undefined for the most part of the lifetime of the linked list, until we are about to invoke `unlink`. In particular, the ghost parameter contains the index of the node argument, thereby requiring that the node object passed in is part of the chain. In the following discussion, let I be the node index ghost parameter and σ the chain of the linked list: then $\sigma[I]$ is assumed to be the node argument of the method `unlink`.

The verification of the unlink method is not fully automatic, see the five videos [3,4,5,6,7].

Verifying unlink consists of four main cases: these correspond to the possible branches of the two if-statements (see Listing 6). The challenge again is to reestablish the class invariant in the heap after the method completes. The main insight is that, by the acyclicity property, all the nodes are separate: this allows us to distinguish the heap updates to apply only to the node that is actually affected, while leaving the other nodes equal to the situation in the heap before. The three important cases are depicted in Figs. 4, 5, 6 (compare with Fig. 2).

1. Suppose the test of both if-statements evaluate to true: for node x, it holds that $\text{next}(x) = $ `null` and $\text{prev}(x) = $ `null`. Then we know the list consists of exactly one node, as the node we are unlinking is the first and the last node. So I cannot be larger than 0. In the case the node index is zero, the class invariant is proven automatically [3, 7:25].

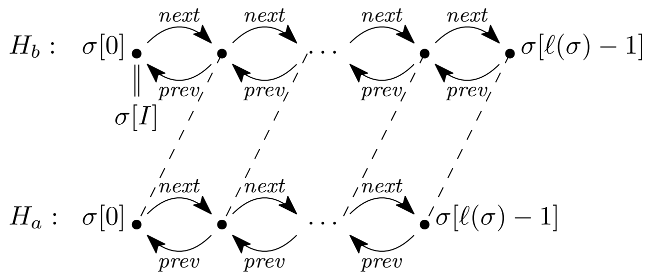

Fig. 4: The heap before (H_b) consists of a chain σ with $\ell(\sigma) \geq 2$. The dashed lines show which objects are identical in the heaps. Interpreting $\sigma[I]$ in the new heap gives $\sigma[I+1]$ in the old heap, as I does not change. The `first` field of the linked list has changed (not shown), and the second node in the old heap now has `null` as `prev`. Moreover, the `next` and `prev` field of the unlinked node have been set to `null` (not shown).

2. Suppose the test of the first if-statement evaluates to true, but the test of the second if-statement evaluates to false: for node x, it holds that $\text{next}(x) \neq$ `null` and $\text{prev}(x) =$ `null`. We thus know that the list consists of at least two nodes, and it is the first node we are unlinking. Thus, I cannot be larger than 0. If the node index is zero, the class invariant is not proven automatically [4, 2:40], but we have two open goals corresponding to the chain properties (b) and (c), cf. proof of `linkLast`. Our situation is different now, see Fig. 4. Here our insight applies: because of acyclicity, we know all nodes are different. Thus, an update of $\sigma[I]$'s fields do not affect the other nodes. When proving (c) this is sufficient as no next field of nodes in the new chain are changed compared to the old heap [4, 10:14-15:28]. When proving (b), we furthermore make a case distinction between the new first node and the other nodes: the former follows from the heap update, the latter from the old invariant [4, 3:27-10:13].
3. Suppose the test of the first if-statement evaluates to false, and the test of the second to true: for node x, it holds that $\text{next}(x) =$ `null` and $\text{prev}(x) \neq$ `null`. This means that the list consists of at least two nodes, and it is the last node we are unlinking. Proof is similar to the previous case: see Fig. 5 and [5].

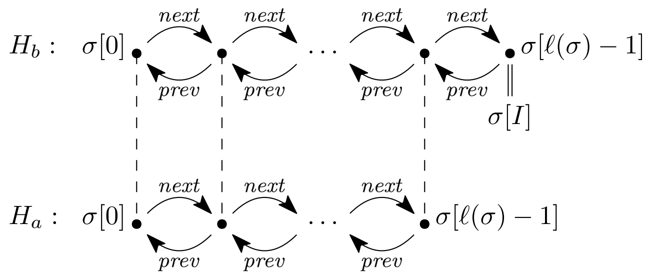

Fig. 5: The heap before (H_b) consists of a chain σ with $\ell(\sigma) \geq 2$. The dashed lines show which objects are identical in the heaps. Interpreting $\sigma[I]$ in the new heap is invalid, as $I = \ell(\sigma)$ in the new heap. The `last` field of linked list has changed (not shown), and the before last node in the old heap now has `null` as `next` field.

4. Suppose both tests of if-statements evaluate to false: for node x, it holds that $\text{next}(x) \neq$ `null` and $\text{prev}(x) \neq$ `null`. This implies that the list consists of at least three nodes: where x is some 'interior' node. This part of the proof is the largest, as it involves many case distinctions. Up to the point where the class invariant is established in the heap after goes as before, except for (b) and (c). Keep in mind the situation as depicted in Fig. 6, and see [6,7].
We distinguish the two cases:
   (b) Establishing the `prev` field property of the chain involves the following observation: there are three cases. First case, for all nodes at an index $0 \leq i < I$ in the old heap, we know they are identical to the nodes at

the same index in the new heap. We know the heap is updated to assign the `prev` field of $\sigma[I+1]$ in the old heap, and by acyclicity we know this node is separate from the nodes all before $\sigma[I]$ in the old heap. Second case, $\sigma[I]$ interpreted in the new heap is identical to $\sigma[I+1]$ in the old heap, and precisely for this node the `prev` field was updated to become $\sigma[I-1]$ in the old heap (which is $\sigma[I]$ in the new heap). Third and last case, for all nodes at an index $I+1 < i < \ell(\sigma)$ in the old heap, we know they are identical to the nodes at $\sigma[i-1]$ in the new heap. Again, by acyclicity we know that the node $\sigma[I+1]$ in the old heap is separate from the nodes with a higher index, so we know their `prev` field cannot be affected by the update.

(c) Establishing the `next` fields property of the chain is very similar, but with the index offset by one. Observe that the `next` field of $\sigma[I-1]$ in the old heap is updated to $\sigma[I+1]$ in the old heap. Thus the three cases are: first, for the nodes with index $0 \leq i < I-1$ in the old heap, second, for the node $\sigma[I-1]$ in the old heap, and third, for the nodes with index $I < i < \ell(\sigma)$ in the old heap.

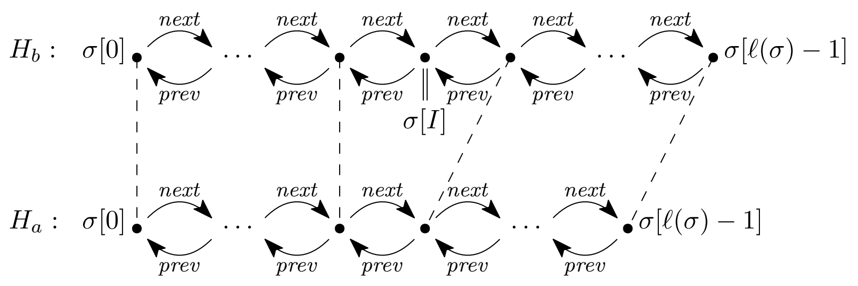

Fig. 6: The situation (H_b) consists of a chain σ with $\ell(\sigma) \geq 3$. The index I remains unchanged in the heaps, thus $\sigma[I]$ in the heap after is equal to $\sigma[I+1]$ in the heap before. The following fields are updated in the new heap: the `next` field of the node at $\sigma[I-1]$ in the old heap becomes the node at $\sigma[I+1]$ in the old heap, and the `prev` field of the node at $\sigma[I+1]$ in the old heap becomes the node at $\sigma[I-1]$ in the old heap. In the new heap these two nodes are present in succession in the chain, thus satisfying the chain properties (b) and (c).

Now that we have established that `unlink` removes a node from the chain while maintaining the class invariant, we can return to the verification of the `remove` method. The remove method iterates over the linked list until it has obtained a node to remove. However, the termination of this iteration is not obvious. Moreover, before invoking the `unlink` method, we need to specify the value of its ghost parameter: the index corresponding to the node. So, before we can verify the `remove` method, we add three kinds of annotations to its source: a ghost variable for maintaining the current index, a loop invariant that establishes termination and maintains the class invariant, and a set annotation before invoking the `unlink` method (see Listing 18).

```
public boolean remove(/*@ nullable @*/ Object o) {
    //@ ghost \bigint index = -1;
    if (o == null) {
        /*@ maintaining 0 <= (index + 1) &&
          @   (index + 1) <= nodeList.length;
          @ maintaining (\forall \bigint i; 0 <= i < (index + 1);
          @      ((Node)nodeList[i]).item != null);
          @ maintaining (index + 1) < nodeList.length ==>
          @   x == nodeList[index + 1];
          @ maintaining
          @   (index + 1) == nodeList.length <==> x == null;
          @ decreasing nodeList.length - (index + 1);
          @ assignable \strictly_nothing; */
        for (Node x = first; x != null; x = x.next) {
            //@ set index = index + 1;
            if (x.item == null) {
                //@ set nodeIndex = index;
                unlink(x);
                return true;
        }   }
    } else {
        /*@ maintaining 0 <= (index + 1) &&
          @   (index + 1) <= nodeList.length;
          @ maintaining (\forall \bigint i; 0 <= i < (index + 1);
          @      !o.equals(((Node)nodeList[i]).item));
          @ maintaining (index + 1) < nodeList.length ==>
          @   x == nodeList[index + 1];
          @ maintaining
          @   (index + 1) == nodeList.length <==> x == null;
          @ decreasing nodeList.length - (index + 1);
          @ assignable \strictly_nothing; */
        for (Node x = first; x != null; x = x.next) {
            //@ set index = index + 1;
            if (o.equals(x.item)) {
                //@ set nodeIndex = index;
                unlink(x);
                return true;
    }   }   }
    return false;
}
```

Listing 18: The JML annotations of method `remove`. We use a slightly unnatural initial value for the index ghost variable, since the KeY 2.6.3 parser does not recognize the @set annotation if it appears after the if-statement.

The verification of above method is not fully automatic, see [8,9]. The proof consists of two parts, corresponding to the branches of the if-statement. In the proof, one shows (among other properties) that the loop invariant holds initially, after each iteration, and at the end of the loop. It is important to note that ($index + 1$) is equal to the length of the chain precisely when the end of the loop has been reached. This holds since we use the `next` field to traverse the chain, and only the last node has a `null` successor. Moreover, the distance to the last node decreases each iteration, and this distance is bounded from below by zero: thus the loop must terminate. Moreover, the loop is *strictly pure*, as it never modifies the heap in any of its completed iterations. The exceptional case is the last iteration in which the `remove` method returns early. Due to the early return, the loop invariant no longer needs to be shown (and so also not its heap purity). For reasons of limited space, further examination of its proof is left as a challenge to the reader.

8 Conclusion

Over the course of this paper, we have studied two essential methods of Java's `LinkedList` class: `add` and `remove`. The original implementation contains an overflow bug, and we have looked at a revised version that imposes a maximum length of the list. Furthermore, we have set out to verify that the overflow bug indeed no longer occurs. Towards this end, we have formally specified a class invariant and method contracts, with two goals: establishing the absence of the overflow bug, and capturing the 'essential' behavior of the methods with respect to the structural properties of the linked list. All methods have been formally verified [11] using the KeY theorem prover, and video material shows how [10].

A number of proof challenges were left for the reader:

Challenge 1. Describe (informally) the high-level steps of the correctness proof of `remove`.

Challenge 2. The proofs shown in the videos may not be the shortest and could contain detours: find proofs that have a fewer number of (interactive) steps.

Challenge 3. Make use of `assignable` and `accessible` clauses in JML with dynamic footprints (see [13, Section 9.3]): does this make the proofs shorter?

Challenge 4. Write a specification and verify the correctness for these linked list methods: `linkFirst`, `linkBefore`, `node`, `indexOf`, `clear`.

Challenge 5. Refine the specifications to relate the items in the 'heap before' to the items in the 'heap after', e.g. all items in the heap before `add` is called must remain at the same position in the heap after, and verify the correctness with respect to the refined contracts.

Challenge 6. Abstract the specifications from properties related to items, i.e. only show that the class invariant is maintained by the methods: can the proof still be done?

Self-references

1. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 1 (2020). https://doi.org/10.6084/m9.figshare.11662824
2. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 2 (2020). https://doi.org/10.6084/m9.figshare.11673987
3. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3a (2020). https://doi.org/10.6084/m9.figshare.11688816
4. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3b (2020). https://doi.org/10.6084/m9.figshare.11688858
5. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3c (2020). https://doi.org/10.6084/m9.figshare.11688870
6. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3d (2020). https://doi.org/10.6084/m9.figshare.11688984
7. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 3e (2020). https://doi.org/10.6084/m9.figshare.11688891
8. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 4a (2020). https://doi.org/10.6084/m9.figshare.11699178
9. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Part 4b (2020). https://doi.org/10.6084/m9.figshare.11699253
10. Bian, J., Hiep, H.A.: A Tutorial on Verifying LinkedList using KeY: Video Material (2020). https://doi.org/10.6084/m9.figshare.c.4826589.v2
11. Hiep, H.A., Bian, J., de Boer, F.S., de Gouw, S.: A Tutorial on Verifying LinkedList using KeY: Proof Files (2020). https://doi.org/10.5281/zenodo.3613711
12. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 217–234. Springer (2020)

References

13. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
14. Bloch, J., Gafter, N.: Collection (Java Platform SE 8), `https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html`, retrieved on 23/01/2020
15. Huisman, M.: Verification of Java's AbstractCollection class: A case study. In: Mathematics of Program Construction, 6th International Conference. pp. 175–194 (2002). https://doi.org/10.1007/3-540-45442-X_11
16. Rooney, J.J., Heuvel, L.N.V.: Root cause analysis for beginners. Quality progress **37**(7), 45–56 (2004)