Open Universiteit Nederland

Graduation Assignment

Verifying OpenJDK's LinkedList using KeY

Student (name, number)*:*
O. Maathuis
851835934

Thesis committee:
Prof. dr. M. van Eekelen (chair)
Dr. S. de Gouw (supervisor)

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering

at the

Department of Computer Science

Issue date: March 11, 2020
Presentation date: March 13, 2020

Open Universiteit
www.ou.nl

Acknowledgements

First of all, I would like to thank my supervisor Stijn de Gouw for his excellent guidance during the graduation process. Despite his busy schedule, he was able to provide precise, helpful feedback on my work and to have regular meetings with me to discuss the progress of our study.

I also wish to thank Marko van Eekelen, the chairman of my graduation committee, for his feedback on my research proposal and on this thesis.

Completing my master would not have been possible without my employer, Achmea. They supported and enabled me to do this. Special thanks in this respect to Nico Heijnen.

Finally, but not in the least, I want to thank my wife Sandra. For keeping up with me during this study, for missing out on the things we could have done together while I was busy studying. And for taking care of me when I forgot myself.

Olaf Maathuis
Hengelo Ov., March 11, 2020

Contents

List of Figures

List of Tables

List of Listings

Abstract

Software libraries are the building blocks of millions of programs, and they run on the devices of billions of users every day. Therefore, their correctness is of the utmost importance.

Formal software verification is a rigorous way of functional software validation, in that it ensures that software is correct, i.e., it proofs the absence of bugs. Due to the intrinsic complexity of modern software, the possibility of interventions by a human verifier is indispensable for proving correctness. This holds in particular for the Java Collection library, where programs are expected to behave correctly for inputs of arbitrary size. This software is widely used and stable, making the effort of formal verification worthwhile. The research question is:

How to specify, verify and improve Java's LinkedList class?

We are using Java's OpenJDK implementation of a linked list. To specify its methods we use Java Modelling Language (JML). The state-of-the-art verification system KeY is used to check whether a method complies to its specification. It turns out that 24 out 42 public methods are broken. Test cases are presented to show where bugs originate and where they get 'passed on'. The root cause of the bugs can be eliminated by a few slight modifications in the source code. To that end, we created a clone of `LinkedList` with these modifications. This clone has been verified to a large extent, and the conclusion is that the absence of bugs has been proven in the verified part of the clone

Chapter 1

Introduction

The impact of software on society is huge. For instance, think of telephone network controls, patient hospital monitoring, avionics, GPS navigation, etcetera. If software fails, these kind of tools and applications — and many more — may be affected.

On June 4, 1996, during its first flight, the Ariane 5 rocket exploded 37 seconds after take off from the coast of French Guiana. The explosion was caused by a fault in the rocket's Inertial Reference System. This system was used to determine whether the rocket was pointing up or down, which is formally known as the horizontal bias, or informally as a BH value. This value was represented by a 64-bit floating variable, and problems began to occur when this variable was stuffed into a 16-bit integer variable: as the rocket's velocity increased, the 64-bit variable exceeded 65k, and became too large to fit in a 16-bit variable. The error was not caught by an exception handler,[1] resulting in a software crash culminating into an explosion. The disastrous launch costed approximately $370 M and is widely acknowledged as one of the most expensive software failures in history.

Begin of June 2018, the Parity Ethereum client software turned out having a critical vulnerability, causing those running the software to fall out of sync, meaning others using different software would not recognize their transactions. A Parity developer stated: "We missed a conditional check in our code that caused full node Parity to accept a block containing invalid transactions". According to Parity, the software was patched before anyone was actually able to exploit the bug and as a result a mainnet split was obviated.[2]

A reliable source for stories about unreliability is the Risks forum (www.risks.org), where a few thousands issues can be found starting at August 1985. The issues reported demonstrate the potential for software errors to result in large liability, repair costs, and embarrassment. The press represents the embarrassment, and the exposure that increases liability.

Correctness of software (libraries) thus is of the utmost importance. The importance and potential of formal software verification as a means of rigorously validating state-of-the-art, real software and improving it, is convincingly illustrated by its application to TimSort, the default sorting library in many widely used programming languages, including Java and Python, and platforms like Android (see [1, 2]): a crashing implementation bug was found.

1.1 Software Verification

The ISO/IEC/IEEE 24765:2010 standard, providing a common vocabulary applicable to all systems and software engineering work, defines verification as [3, p. 394]:

[1] Code which would have caught and handled these conversion errors had been disabled for the BH value, due to performance constraints on the Ariane 4 hardware which did not apply to Ariane 5.

[2] It helped that the bug was discovered on a test environment.

1. the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase
2. formal proof of program correctness
3. confirmation, through the provision of objective evidence, that specified requirements have been fulfilled
4. confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled
5. the evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition
6. process of providing objective evidence that the software and its associated products comply with requirements (e.g., for correctness, completeness, consistency, and accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance), satisfy standards, practices, and conventions during life cycle processes, and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities (e.g., building the software correctly)

Verifying software ought to ensure the software to be reliable. In general, software reliability not only includes correctness, but "precision" (ensuring information is delivered at an appropriate level of detail) and "timeliness" (ensuring that information is delivered when it is required) as well [4, p. 293]. We do not make this distinction; instead the terms reliability and correct software are used interchangeably, both referring to system behaviour consistent with that defined in the specification. Users of a software system may perceive the system as unreliable when the system specification does not match their expectations [4, p. 296], however that's another story.

We narrow software verification to *formal proof of program correctness* (see nr. 2 in the above list), defined by the IEEE Std 610.12-1990 standard as [5, p. 59]:

proof of correctness (1) A formal technique used to prove mathematically that a computer program satisfies its specified requirements. (2) A proof that results from applying the technique in (1).

Software components are then considered to be mathematical artefacts [6, p. 43], where a specification language, with a formal syntax and a formal semantics, is needed. Our focus is on a part of the Java Collections framework, viz., the `LinkedList` class. This software is widely used and stable, making the effort of formal verification worthwhile. We will be using OpenJDK 8.[3] JML[4] [7, 8, 9] is a specification language for Java, where specifications are written in special annotation comments (which start with an at-sign (@)). In order to use formal methods, a formal proof system (verification system) is needed that will analyse and/or transform specifications along with the source code into a proof using a mathematical approach. We will verify the `LinkedList` class with KeY[5,6][10, 11]. In addition, if the existence of bugs can be proven, enhancements which solve the bugs should be presented.

[3] The specific source code can be retrieved from https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src. (Path: `share/classes/java`.) Download: https://hg.openjdk.java.net/jdk8/jdk8/jdk/archive/687fd7c7986d.zip/src/.

[4] Java Modeling Language: See http://www.eecs.ucf.edu/~leavens/JML/index.shtml.

[5] See https://www.key-project.org/.

[6] For the version of KeY we are using see chapter 4.

1.2 KeY

The KeY tool is a source-code-based verification tool aiming at verification of Java programs. Specifications can be given in an extension to the JML, viz., KeY-JML. At the core of KeY is an interactive theorem prover for formulas expressed in a program logic called Java Dynamic Logic [11, 12], Java DL for short, being an extension of dynamic logic (DL) [13]. In addition to this (interactive) deductive verification it uses symbolic program execution [14], that enumerate reachable states without loss of precision. Symbolic program execution can be seen as abstract interpretation [15, p. 27], and abstract interpretation on its turn can be seen as a very general method to render infinite computations finite in a sound manner. The combination of these methodologies — deductive verification and abstract interpretation — makes KeY a state-of-the-art tool to use.

Complete coverage of Java is still a challenge in KeY; however it fully supports the verification of programs written in a stripped down version of Java for programming smart cards and small mobile devices, viz., Java Card. KeY reflects mostly Java 1.5. Features not supported by KeY are: floating-point types, concurrency, and generic types.[7]

Java methods can be verified by providing KeY with the source code together with formal specifications. For this purpose the source code is annotated with KeY-JML, i.e., JML extended with features intended for KeY. KeY can prove whether a method satisfies its contract. If not, the contract breaks, otherwise it holds — and the method is said to be formally verified. The semantics of a contract should completely and precisely render the intended behaviour of its implementation.

1.3 Related Work

De Gouw et al. [16] proved the correctness of two sorting algorithms: Counting sort and Radix sort. Radix sort relies on an external sorting algorithm (Counting sort, for instance), and for its correctness it is crucial that the external sorting algorithm is stable, which means that equal elements in the input array appear in the same order in the output array. They formalized stability using an auxiliary array variable *idx* that keeps track of the original index in the input array of each element in the output array. This proves correctness with respect to an external (stable) sorting algorithm that updates *idx* appropriately.

The correctness of TimSort, which is the main sorting algorithm provided by the Java standard library (for arrays of reference type[8]), was investigated by De Gouw et al. [2]. Initially, it turned out that they weren't able proving its correctness. A closer analysis uncovered that this was because TimSort was broken, i.e., a bug was found in the method `mergeCollapse()`. Conditions under which the bug occurred where identified, and from this a slightly modified version of this method resulted. The bug-free version of TimSort doesn't compromise performance.

While KeY aims mostly at verification to prove the absence of bugs, software verification can also be applied as a safety[9] prover. David et al. [17] used the logic SLH for reasoning about the safety and termination of heap-manipulating programs with potentially cyclic singly-linked lists. Their work relates the structure of lists to their length, i.e., reachability constraints with numeric ones, and they empirically

[7] Generic types: This can be accounted for without loss of precision, see chapter 4.

[8] The KIT group used KeY to verify the JDK's dual pivot quicksort implementation that is used as default sorting algorithm for integer or long arrays.

[9] E.g., memory safety, i.e., absence of null or dangling pointer dereferences.

show that SLH is both efficient and expressive enough for reasoning about safety and (especially) termination of list programs. Brain et al. [18] propose a decision procedure for reasoning about aliasing and reachability based on Abstract Conflict Driven Clause Learning (ACDCL) [19]. As they don't capture the lengths of lists, these logics are better suited for safety and less for termination proving.

Most current approaches to software verication are one-sided — a safety prover will try to prove that a program is safe, while a bug-finding tool will try to find bugs. Lewis [20] analyses C programs based on underapproximate loop acceleration (used as a viable technique for proving safety, as well as finding bugs) and second-order SAT[10] (for program analysis) respectively. He shows that programs accelerated in this way can be optimised by inlining trace automata to reduce their reachability diameter.

Knüppel et al. [21] provide a report on the specification and verification of some methods of the classes ArrayList, Arrays, and Math of the OpenJDK Collections framework using KeY. Their report is mainly meant as a "stepping stone towards a case study for future research." To the best of our knowledge, no formal specification and verification of the actual Java implementation of a linked list has been investigated. In general, the data structure of a linked list has been studied mainly in terms of pseudo code of an idealized mathematical abstraction (see [22] for an Eiffel version and [23] for a Dafny version).

1.4 Research Question

The research question is:

How to specify, verify and improve Java's LinkedList class?

This question is divided into the following:

RQ1 Can we find bugs in the LinkedList class?

RQ2 Can we synthesize testcases that expose identified bugs?

RQ3 How to improve the LinkedList implementation?

RQ4 To what extent can we verify the LinkedList class (or a fixed version of it)?

The general workflow underlying the TimSort case, as depicted in figure 1.1, will be followed. The workflow starts with a formalisation of the informal documentation of the Java code in JML. This formalisation goes hand in hand with the formal verification: failed verification attempts can provide information about further refinements of the specifications. A failed verification attempt may also indicate an error in the code, and can as such be used for the generation of test cases to detect the error at run-time.[11] Of course, test cases can also be constructed manually. Another possibility is that a failed verification attempt leads directly to a code revision, and after that verification starts again, whether or not preceded by altering the specification. A test case can also be the end of the workflow; this is the case when after a code revision the next (cyclus of) (specification and) verification is successful. This stresses the *motivation* for this research and its approach: if we succeed in finding bugs, which could be solved by a formally verified fix, in theory testing is not needed in the first place. Still it's nice to compare test cases before and after a fix.

[10] An extension of propositional SAT that allows quantification over functions.

[11] To use KeYTestGen, KeY needs the Z3 SMT solver and OpenJML to be installed.

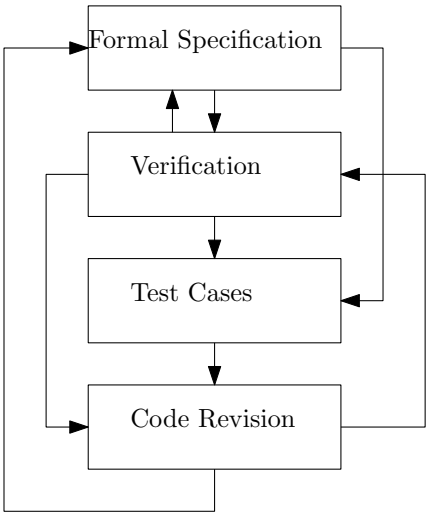

FIGURE 1.1: Workflow

1.5 Article TACAS Conference

During the course of this research the student's supervisor came up with the idea to write an article on this subject for the TACAS conference on 25–30 April 2020.[12] The paper, which has the same title as this thesis, has been accepted, and very recently a camera-ready version has been submitted to the people of this conference [24]. The authors are:

Hans-Dieter A. Hiep[a]
Jinting Bian[a]
Frank S. de Boer[a]
Marko van Eekelen[b]
Stijn de Gouw[b]
Olaf Maathuis[c]

[a]CWI, Science Park 123, 1098 XG Amsterdam, The Netherlands
{hdh,j.bian,frb}@cwi.nl
[b]Open University, P.O. Box 2960, 6401 DL Heerlen, The Netherlands
{marko.vaneekelen,stijn.degouw}@ou.nl
[c]Achmea, P.O. Box 700, 7300 HC Apeldoorn, The Netherlands
olaf.maathuis@achmea.nl

This thesis is also based on that article. As such, others have contributed substantial to chapter 5, in particular w.r.t. section 5.1. Figure 2.2 (created by Hans-Dieter A. Hiep) is taken from the article. Figure 2.1 is based on a similar one in the article. For additional remarks see section 5.3.

1.6 Outline

Chapter 2 describes Java's OpenJDK implementation of a linked list and its relevant properties. Chapter 3 starts with a bug analysis, followed by test cases that originate or exhibit bugs. In addition, we present a bug solving code revision. This refinement is taken as the basis for chapter 4, where the adapted linked list is specified and verified.

[12] 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.

Chapter 5 finalizes with a discussion and a conclusion. The Zenodo repository [25] contains an on-line appendix of the thesis.

Chapter 2

Linked List Implementation

2.1 Linkedlist in OpenJDK

`LinkedList` was introduced in Java version 1.2 as part of Java's collection framework in 1998. Figure 2.1 shows in a UML diagram how `LinkedList` fits in the type hierarchy of this framework. This picture shows the complex inheritance structure. `LinkedList` implements the `List` interface, and also supports all general `Collection` methods as well as the methods from the `Queue` and `Deque` interface. The `List` interface provides positional access to the elements of the list, where each element is indexed by Java's primitive `int` type.

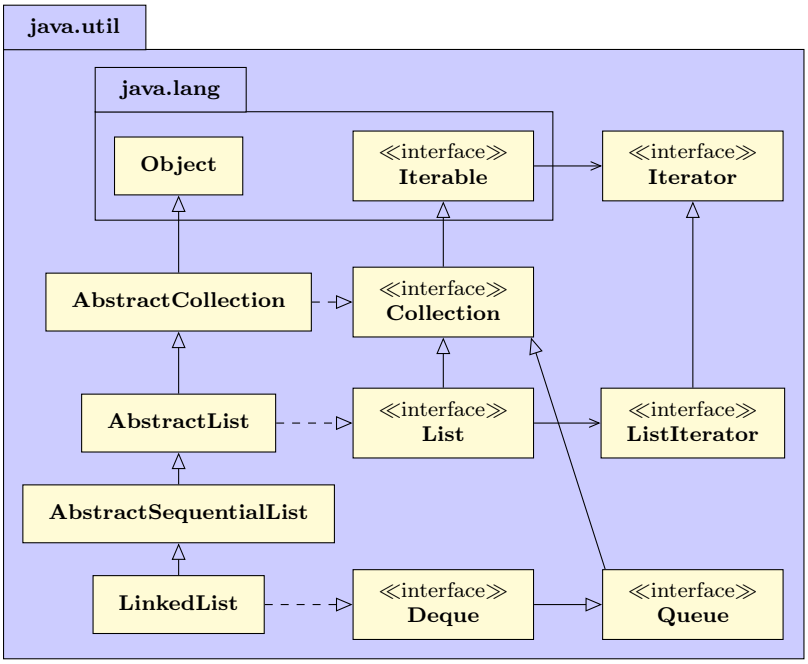

FIGURE 2.1: LinkedList within Java Collections framework

2.2 Properties

A linked list is a sequence of elements, of type `Node<E>` (see Listing 2.1), a private static class of `LinkedList<E>`. The head of the sequence is referred to by `first`, and the tail by `last`.

[13]The symbol ↪ represents a line break, to prevent overflow in this document. It's not a line break in the actual source.

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable,
    ↪  java.io.Serializable
{
    transient int size = 0;
    transient Node<E> first;
    transient Node<E> last;
    ...
}
```

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E>
    ↪  next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

Listing 2.1: Outline of LinkedList<E>. The LinkedList class defines a doubly-linked list data structure.[13]

Let $n \geq 0$ denote the number of elements. The head and the tail have value `null` when there are no elements. We have:

$$n = 0 \leftrightarrow (\texttt{first} = \mathit{null} \wedge \texttt{last} = \mathit{null})$$

(To abstract somewhat from Java, we write $x = y$ instead of $x == y$.) The other way around[14]: $n > 0 \leftrightarrow \neg(\texttt{first} = \mathit{null} \wedge \texttt{last} = \mathit{null})$. De Morgan[15]: $n > 0 \leftrightarrow (\texttt{first} \neq \mathit{null} \vee \texttt{last} \neq \mathit{null})$. A non-empty sequence ($n > 0$) however, must have *both* fields (`first`, `last`) refer to an instance of type `Node`, i.e., they must both not be `null`. Let f be the instance of `Node` to which `first` refers and let l be the instance of `Node` to which `last` refers, i.e., f and l are inherently not `null`. Then we get: $n > 0 \leftrightarrow (f = \texttt{first} \wedge l = \texttt{last})$. f and l have unique features that other nodes (when present) have not (see properties 2.1 and 2.2): $f.\texttt{prev} = \mathit{null} \wedge l.\texttt{next} = \mathit{null}$. Combining the last two formulas yields:

$$n > 0 \leftrightarrow (f = \texttt{first} \wedge l = \texttt{last} \wedge f.\texttt{prev} = \mathit{null} \wedge l.\texttt{next} = \mathit{null})$$

When $n = 1$, f and l coincide: $n = 1 \leftrightarrow f = l$. Combining the last two formulas yields:

$$n = 1 \leftrightarrow (f = \texttt{first} \wedge l = \texttt{last} \wedge f.\texttt{prev} = \mathit{null} \wedge l.\texttt{next} = \mathit{null} \wedge l.\texttt{prev} = \mathit{null} \wedge f.\texttt{next} = \mathit{null})$$

When $n > 1$, f and l do not coincide:

$$n > 1 \leftrightarrow f \neq l$$

Figure 2.2 visualises aforementioned properties. It shows properties of instances l_i ($i \in \{0 \ldots 2\}$) of type `LinkedList`, where the number of elements $n_i = i$. When $n_i > 0$ the list is considered as a chain. The $i > 1$ case demonstrates the way connectivity between nodes[16] is ensured: using fields `next` and `prev` a node refers to (present) neighbours. An item can be the `null` value. When an item is not the `null` value, i.e., when it refers to an object, there might be other nodes having their `item` field refer to the same object.

Let $0 \leq j < n$ and let $\sigma[j]$ be the $(j+1)^{\text{th}}$ node in the chain. Then we can distinguish properties *Unique head and tail* and *Connectivity*:

[14] $(p \leftrightarrow q) \leftrightarrow (\neg p \leftrightarrow \neg q)$
[15] $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$
[16] A node in the sequence refers to an instance of type `Node`, and is thus inherently not `null`.

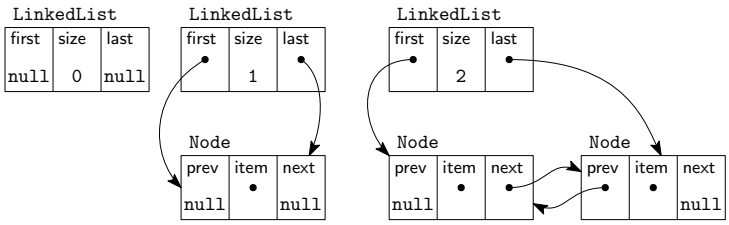

FIGURE 2.2: Structure of linked list. Items themselves are not shown.

Unique head and tail

$$j = 0 \leftrightarrow \sigma[j] = f \leftrightarrow \sigma[j].\texttt{prev} = null \tag{2.1}$$
$$j = n - 1 \leftrightarrow \sigma[j] = l \leftrightarrow \sigma[j].\texttt{next} = null \tag{2.2}$$

Connectivity

$$0 < j < n \leftrightarrow \sigma[j] = \sigma[j-1].\texttt{next} \tag{2.3}$$
$$0 \leq j < n - 1 \leftrightarrow \sigma[j] = \sigma[j+1].\texttt{prev} \tag{2.4}$$

2.2.1 Invariant

When a method is called on an instance *list* of `LinkedList` that changes its internal state, the properties 'Unique head and tail', and 'Connectivity' must be preserved. Let $s \equiv V(\texttt{size})$ denote the value of `size`, the *cached* size of *list*. Listing 2.2 shows method `add(E e)` of `LinkedList` that appends the specified element e to the end of this list. A call of $list.\texttt{add}(e)$ must ensure that adding the element is done in such a way that after the method has terminated, these properties still hold.

```
public boolean add(E e) {
    linkLast(e);
    return true;
}
```

```
1  /**
2   * Links e as last element.
3   */
4  void linkLast(E e) {
5      final Node<E> l = last;
6      final Node<E> newNode = new Node<>(l, e,
       ↪  null);
7      last = newNode;
8      if (l == null)
9          first = newNode;
10     else
11         l.next = newNode;
12     size++;
13     modCount++;
14 }
```

LISTING 2.2: add(E e)

In the method, an instance of type `Node` is created that will be added

- to an empty list,[17] in which case the new node becomes the only element of the (new) chain, and thus coincides with f and l; or
- to an non-empty list, i.e., a chain, in which case the new node becomes the new tail, the old and new tail pointing to each other.

Note that s is updated by incrementing it with 1. The method shows that when the properties hold in its prestate, they still do after the method has terminated.

[17] In the method this is detected by checking whether $l = \texttt{null}$; see line 8 in Listing 2.2.

Updating s is crucial for methods that depend on s, like method `node(int index)` (see Listing 2.3), that returns the (non-null) `Node` at the specified element index. It traverses the list from the beginning or the end, whichever is closer to the specified index i.[18] When the node at the position corresponding with the given index is reached, it is returned to the caller.

```
1  Node<E> node(int index) {
2      // assert isElementIndex(index);
3      if (index < (size >> 1)) {
4          Node<E> x = first;
5          for (int i = 0; i < index; i++)
6              x = x.next;
7          return x;
8      } else {
9          Node<E> x = last;
10         for (int i = size - 1; i > index; i--)
11             x = x.prev;
12         return x;
13     }
14 }
```

Listing 2.3: node(int index)

2.3 Acyclicity

Lemma 2.1. *The linked list, i.e., the sequence of nodes, is acyclic: for each* $0 \leq j < n$ *there is no* $0 \leq i < j$ *such that* $\sigma[j] = \sigma[i]$.

We have to proof this[19]

$$\forall_{0 \leq j < n} \left(\neg \exists_{0 \leq i < j} \left(\sigma[j] = \sigma[i]\right)\right) \tag{2.5}$$

Proof by Contradiction

Proof. Suppose the opposite is true, viz., the list is cyclic:

$$\exists_{0 \leq j < n} \left(\exists_{0 \leq i < j} \left(\sigma[j] = \sigma[i]\right)\right) \tag{2.6}$$

From this it follows:

$$\forall_{j \leq k < n} \left(\sigma[k] = \sigma[k - (j - i)]\right) \tag{2.7}$$

Proof by Induction

Base $k = j$: $\sigma[j] = \sigma[i]$ by assumption (see 2.7)

Step $k + 1$:

$$\begin{aligned} \sigma[k+1] &= \sigma[k].\texttt{next} && \text{(see 2.3)} \\ &= \sigma[k - (j - i)].\texttt{next} && \text{(base)} \\ &= \sigma[k - (j - i) + 1] && \text{(see 2.3)} \end{aligned}$$

We have now proven that *if* 2.6 holds, *then* 2.7 holds. When $k = n - 1$, it follows from 2.2 that $\sigma[k].\texttt{next} = \mathit{null}$. 2.7 then implies that $\sigma[n - 1 - (j - i)].\texttt{next} = \mathit{null}$,

[18] Whether f or l is closer to i is determined by applying the signed right shift operator $\gg 1$ to s. See line 3 in Listing 2.3.

[19] We may also write (duality, i.e., $\neg\exists P(x) \leftrightarrow \forall\neg P(x)$): $\forall_{0 \leq j < n} \left(\forall_{0 \leq i < j} \neg \left(\sigma[j] = \sigma[i]\right)\right)$, and thus: $\forall_{0 \leq j < n} \left(\forall_{0 \leq i < j} \left(\sigma[j] \neq \sigma[i]\right)\right)$.

which according to 2.3 is impossible.[20] In other words: 2.7 does not hold, and thus 2.6 does not hold,[21] from which it follows that 2.5 must hold. □

[20] $j - i > 0$, and thus $n - 1 - (j - i) < n - 1$.
[21] $(p \to q) \leftrightarrow (\neg q \to \neg p)$ (Take 2.6 for p and 2.7 for q.)

Chapter 3

Analysis and Testcases

When writing specifications for `LinkedList`, we soon discovered that this class is error-prone w.r.t. integer overflow. Listing 3.1 shows a preliminary contract for `addFirst(Object e)`. This method inserts the specified element at the beginning of this list. What is fundamental is the difference between the semantics of a mathematical addition and Java's implementation of it. The casting with the `\bigint` type is to enforce a *mathematical* add operation, instead of Java's version (see also Table 4.2). Java's semantics of an add operation of two variables of type `int` is one where *overflow* has to be taken into account. Looking at the `ensures` clause, which acts as a postcondition (see also chapter 4), it is evident that here we have a problem at some point: as soon as in the prestate `size()` (which simply returns `size`) has reached the maximum value associated with type `int`, the ensures clause yields false. (Note that the last conjunct of the postcondition yields true: $\forall i; \alpha(i); \beta(i)$ is equivalent with $\forall i; \alpha(i) \rightarrow \beta(i)$, and $\alpha(i)$ yields false since `size()` is negative.)

The contract actually states that in *all* circumstances normal execution of `addFirst()` must result in the situation as stated in the postcondition.[22] It is evident that this is not true: The contract of addFirst() does not hold, i.e., the method is broken! This insight stopped us from further verification at that moment and forced us to gain more insight in the overflow issue.

```
/*@
  @ public normal_behavior
  @   ensures
  @     size() == (\bigint)\old(size()) + 1 &&
  @     get(0) == e &&
  @     (\forall int i; 1 <= i < size();
  @      get(i) == \old(get(i-1)));
  @*/
public void addFirst(/*@ nullable @*/ Object e) {
    linkFirst(e);
}
```

LISTING 3.1: JML addFirst(), preliminary version. This contract is expressed in terms of pure public methods `size()` and `get(int index)`.

An integer overflow happens when the result of an arithmetic operation is too big to fit in the number of bits of the variable. In Java, integers use 32 bits to represent signed numbers. Positive values have values from 0 to $2^{31} - 1$. Negative values have values from -2^{31} to -1 . If value $2^{31} - 1$ is incremented, the result does not represent 2^{31} but -2^{31}.

[22] In the absence of a precondition: nothing is required, i.e., no restrictions in the prestate.

3.1 Integer Overflow

`LinkedList` contains methods adding one or more elements to an instance of it, e.g., method `add(E e)` (see section 2.2.1). Each such method will do so unconditionally, and it will increase the value of `size` accordingly, eventually leading to integer overflow. What does integer overflow of `size` mean in terms of properties of the linked list? We will formulate some propositions[23] that we will use later. Let $L(s, n)$ be an instance l of type `LinkedList` where $s \equiv V(\texttt{size})$ and $n \geq 0$ is the number of elements. Considering `size`'s datatype, Java's `int` type, it follows:

$$s = -2^{31} + (2^{31} + n) \bmod 2^{32} \tag{3.1}$$

Let O_s denote overflow of s. Then: $O_s \equiv s \neq n \wedge n \geq 2^{31}$ (3.2)

And the opposite: $\neg O_s \equiv s = n \wedge n < 2^{31}$ (3.3)

From 3.2 it follows: $s < 0 \rightarrow O_s$ (3.4)

The other way around[24]: $\neg O_s \rightarrow s \geq 0$ (3.5)

From 3.1 and 3.2 it follows: $\neg(s \geq 0 \rightarrow \neg O_s)$

The other way around: $\neg(O_s \rightarrow s < 0)$ (3.6)

Furthermore: $O_s \leftrightarrow n > s$ (3.7)

Let $k_s \geq 0$ be the number of times that O_s has occurred: $n = k_s \cdot 2^{32} + s$ (3.8)

From 3.7 and 3.8 it follows: $O_s \leftrightarrow k_s > 0$ (3.9)

The other way around: $\neg O_s \leftrightarrow k_s = 0$ (3.10)

Index For $1 \leq j \leq n$ we can associate the $j^{\texttt{th}}$ element with an index `index` of Java's type `int` whose value $V(\texttt{index}) \equiv i$ is related to j as follows:

$$i = -2^{31} + (2^{31} + j - 1) \bmod 2^{32} \tag{3.11}$$

$$k_s = 0 \rightarrow 0 \leq i \leq 2^{31} - 2 \tag{3.12}$$

$$(-2^{31} \leq i \leq -1 \vee i = 2^{31} - 1) \rightarrow k_s > 0 \tag{3.13}$$

3.2 Assessment

The source code of each method is examined: is it error-prone w.r.t. integer overflow or not. Its documentation, viz., the Javadoc, is taken into account if necessary. If it is not error-prone, it is considered as *compliant*, otherwise as the opposite. We call this non-compliant, but 'broken' might be a better word. Table 3.1 lists (numbered from 1–58, see column 'Nr') all public methods of `LinkedList`: own methods (optionally overridden), as well as inherited.[25] Column 'Type': the type (interface or class) in which the methods reside.[26] For inherited methods this refers to the type closest to `LinkedList<E>`, such that the method is not abstract.

If a method originate integer overflow bug(s) or gets it passed on (whether or not direct) by a bug-originating method, it is considered as non-compliant; otherwise as compliant. Ahead of the outcome: the interruption in the table between nrs. 39 and

[23] These propositions are meant to be be true under all circumstances, i.e., they act as tautologies.

[24] $(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$

[25] Methods inherited from `Object` have been omitted.

[26] `Iterable` resides in package `java.lang`, the other types reside in package `java.util`. See also Figure 2.1.

40 is the marking point between compliant (Nrs. 40–58) and non-compliant methods (Nrs. 1–39).

Table 3.1: Methods

Nr	Type	Method
1	Iterable<T>	forEach(Consumer<? super E> action)
2	Collection<E>	parallelStream()
3	Collection<E>	removeIf(Predicate<? super E> filter)
4	Collection<E>	stream()
5	List<E>	replaceAll(UnaryOperator<E> operator)
6	List<E>	sort(Comparator<? super E> c)
7	AbstractCollection<E>	containsAll(Collection<?> c)
8	AbstractCollection<E>	isEmpty()
9	AbstractCollection<E>	removeAll(Collection<?> c)
10	AbstractCollection<E>	retainAll(Collection<?> c)
11	AbstractCollection<E>	toString()
12	AbstractList<E>	equals(Object o)
13	AbstractList<E>	listIterator()
14	AbstractList<E>	subList(int fromlndex, int tolndex)
15	AbstractSequentialList<E>	iterator()
16	LinkedList<E>	add(E e)
17	LinkedList<E>	add(int index, E element)
18	LinkedList<E>	addAll(Collection<? extends E> c)
19	LinkedList<E>	addAll(int index, Collection<? extends E> c)
20	LinkedList<E>	addFirst(E e)
21	LinkedList<E>	addLast(E e)
22	LinkedList<E>	clone()
23	LinkedList<E>	contains(Object o)
24	LinkedList<E>	descendingIterator()
25	LinkedList<E>	get(int index)
26	LinkedList<E>	indexOf(Object o)
27	LinkedList<E>	lastlndexOf(Object o)
28	LinkedList<E>	listIterator(int index)
29	LinkedList<E>	offer(E e)
30	LinkedList<E>	offerFirst(E e)
31	LinkedList<E>	offerLast(E e)
32	LinkedList<E>	push(E e)
33	LinkedList<E>	remove(int index)
34	LinkedList<E>	set(int index, E element)
35	LinkedList<E>	size()
36	LinkedList<E>	spliterator()
37	LinkedList<E>	toArray()
38	LinkedList<E>	toArray(T[] a)
39	LinkedList<E>	LinkedList(Collection<? extends E> c)
40	AbstractList<E>	hashCode()
41	LinkedList<E>	clear()
42	LinkedList<E>	element()
43	LinkedList<E>	getFirst()

Table 3.1 – continued from previous page

Nr	Type	Method
44	LinkedList<E>	getLast()
45	LinkedList<E>	peek()
46	LinkedList<E>	peekFirst()
47	LinkedList<E>	peekLast()
48	LinkedList<E>	poll()
49	LinkedList<E>	pollFirst()
50	LinkedList<E>	pollLast()
51	LinkedList<E>	pop()
52	LinkedList<E>	remove()
53	LinkedList<E>	remove(Object o)
54	LinkedList<E>	removeFirst()
55	LinkedList<E>	removeFirstOccurrence(Object o)
56	LinkedList<E>	removeLast()
57	LinkedList<E>	removeLastOccurrence(Object o)
58	LinkedList<E>	LinkedList()

3.2.1 Non-compliant Methods

In methods that add one or more elements to the list, the field `size` is incremented accordingly, which may cause integer overflow. The Javadoc of `Collection` states that `size` equals $2^{31} - 1$ if more than that number of elements are present. The methods do not prevent having more than $2^{31} - 1$ elements. And because of that, it is relevant that they don't 'lock' `size` at $2^{31} - 1$ whenever there are (or will be) more than that number of elements. Therefor, these methods, numbered 16, 17, 18, 19, 20, 21, 22, 29, 30, 31, 32, and 39 in Table 3.1, are all considered as non-compliant.

3.2.1.1 Originating Bugs

When overflow of `size` has occurred, things start to go wrong, originating from methods that exhibit bug(s) at the lowest level:

1. descendingIterator() (Table 3.1, Nr. 24)
2. `indexOf(Object o)` (Nr. 26)
3. `lastIndexOf(Object o)` (Nr. 27)
4. `node(int index)` (non-public method[27])
5. `spliterator()` (Nr. 36)

These methods are thus non-compliant.

descendingIterator() See Listing 3.2. An instance of `DescendingIterator` (a private class of `LinkedList`) is created, that on it turn creates an instance of `ListItr` (another private class of `LinkedList`). Having $V(\texttt{index}) \equiv i$, `new ListItr(`$i$`)` is invoked, where $i = s$, which means that field `nextIndex` is set to s. For $s < 0$ we then have that `hasNext()` of descendingIterator() evaluates to `false`.

[27] As such it is not listed in Table 3.1.

```
public Iterator<E> descendingIterator() {
    return new DescendingIterator();
}

private class DescendingIterator implements
↪  Iterator<E> {
    private final ListItr itr = new
    ↪  ListItr(size());
    public boolean hasNext() {
        return itr.hasPrevious();
    }
    public E next() {
        return itr.previous();
    }
    public void remove() {
        itr.remove();
    }
}
```

```
private class ListItr implements
↪  ListIterator<E> {
    private Node<E> next;
    private int nextIndex;
    ...
    ListItr(int index) {
        // assert isPositionIndex(index);
        next = (index == size) ? null :
        ↪  node(index);
        nextIndex = index;
    }
    ...
    public boolean hasPrevious() {
        return nextIndex > 0;
    }
    ...
}
```

Listing 3.2: descendingIterator()

indexOf(Object o) Suppose for the call of `indexOf(`o`)`, the first occurrence of o can be found as the $(i+1)^{\text{th}}$ element in the linked list, then `indexOf(`o`)` returns i as given in Equation 3.11 (see Listing 3.3: $i \equiv V($`index`$)$ denotes the value of `index` of type `int`, and since `index` gets incremented until the search ends, i may suffer from overflow). From relations 3.12 and 3.13 it follows that it is possible that the resulting value is -1 in case of $k_s > 0$. However, this value has a special meaning: it is meant to indicate that the specified element is not present.[28] When `indexOf(`o`)` returns -1, it does not mean that the element *is* associated with index -1. However, if it is, we have a false negative.

```
public int indexOf(Object o) {
    int index = 0;
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null)
                return index;
            index++;
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
    }
    return -1;
}
```

Listing 3.3: indexOf(Object o)

lastIndexOf(Object o) `lastIndexOf(Object o)` contains the same bug as `indexOf(Object o)`. The difference is that it is about the *last* occurrence of o that can be found as the $(i+1)^{\text{th}}$ element in the linked list.

node(int index) Results are questionable/erroneous. We will come to that later, in section 3.2.3.

[28] The Javadoc states this explicitly. In the implementation -1 is returned when a full search has failed to find the element.

spliterator() This method returns an instance of `LLSpliterator<E>`. This is a static nested class of `LinkedList<E>`, that implements `Spliterator<E>`. The constructor sets the field `est`, that represents the estimated size, to -1, but when the field is needed it takes the value of `size` of `LinkedList`, s. When $s < 0$, `getEst()` will return a negative value which will cause `trySplit()` to return `null`. Furthermore, `forEachRemaining(Consumer<? super E> action)` and `tryAdvance(Consumer<? super E> action)` won't do anything (the latter returns `false`). There is a test case (see nr. 56 in Table 3.8) that shows the behavior of `trySplit()` when $s < 0$. `getEst()` is not public, however `estimateSize()`, which returns `getEst()`, is. The main purpose of the test case at hand is to prove that `getEst()` < 0 in case of overflow.

3.2.1.2 Coverage

To determine the coverage of the bugs, we have to find out which methods (directly or indirectly) call the ones listed in section 3.2.1.1. The scope of the analysis is restricted to methods listed in Table 3.1, which enables us to focus solely on the subject of this thesis, viz., `LinkedList`, in a feasible way.

descendingIterator() No calls of this method.

indexOf(Object o) Only used by `containsOf(Object o)`. `containsOf(`o`)` triggers a call of `indexOf(`o`)`, and it returns `false` when the resulting value of that call is -1 (and `true` otherwise). `containsOf(Object o)` then passes on the bug exhibited by `indexOf(Object o)`. Thus, `containsOf(Object o)` (Nr. 23) and the methods to which it passes the bug on (listed in Table 3.2) are non-compliant.

Table 3.2: Callers of contains(Object o)

Type	Call	Nr[29]
`AbstractCollection<E>`	`containsAll(`c`)`[30,31]	7
`AbstractCollection<E>`	`removeAll(`c`)`[30,32]	9
`AbstractCollection<E>`	`retainAll(`c`)`[30,32]	10

lastIndexOf(Object o) No calls of this method.

node(int index) Its use is listed in Table 3.3. Each method asserts the condition in column 'Check' to hold when the call takes place. If the condition does not hold, the call is not executed; instead an error is thrown. It is evident that when $s < 0$, an error is thrown, where as $n > 0$. Furthermore retrieving (get), updating (set) or deleting (remove) a single element when $s = 0$ and $k_s > 0$, results in an error, where as $n > 0$. The methods of Table 3.3 thus are non-compliant. When $k_s > 0$, $i \geq 0$ (if i plays a role) and $s \geq 0$ (or $s > 0$, depending on the caller), no error is thrown; `node(`i`)` *is* called. However, results are questionable/erroneous. node(int index) is non-compliant as well. We will come to that later, in section 3.2.3.

[29] See corresponding values of the equally named column of Table 3.1.
[30] c is of type `Collection<?> c`
[31] `contains(`o`)` is called by `this`.
[32] `contains(`o`)` is called by c.
[33] Actually $0 \leq i \leq s$ is checked, but a call of `node(`i`)` corresponds to a branch in which $i < s$.
[34] c is of type `Collection<?> c`

TABLE 3.3: Callers of node(int index)

Call	Check	Nr
`add(`i`,`$element$`)`	$0 \leq i < s$[33]	17
`addAll(`c`)`[34,35]	$0 \leq s$	18
`addAll(`i`,`c`)`[34]	$0 \leq i \leq s$	19
`get(`i`)`	$0 \leq i < s$	25
`listIterator(`i`)`[36]	$0 \leq i \leq s$	28
`remove(`i`)`	$0 \leq i < s$	33
`set(`i`,`$element$`)`	$0 \leq i < s$	34
`new LinkedList(`c`)`[34,37]	$0 \leq s$	39

Of the method calls listed in Table 3.3, `listIterator(`i`)` is the only one that is used within our scope. Table 3.4 lists the corresponding calls, of which all methods except `hashCode()` are non-compliant.[38] Methods `forEach()`, `containsAll()`, and `hashCode()` make use of enhanced for loops, which are nothing but syntactic sugar over `Iterator`. We show the source code along with the equivalent code with `Iterator`. See Listing 3.4.[39]

TABLE 3.4: Callers of node(int index) via listIterator(int index)

Type	Call	Check	Nr
`Iterable<T>`	`forEach(`$action$`)`[40]	$0 \leq s$	1
`Collection<E>`	`removeIf(`$filter$`)`[41]	$0 \leq s$	3
`List<E>`	`replaceAll(`$operator$`)`[42]	$0 \leq s$	5
`List<E>`	`sort(`c`)`[43,44]	$0 \leq s$	6
`AbstractCollection<E>`	`containsAll(`c`)`[45,46]	$0 \leq s$	7
`AbstractCollection<E>`	`removeAll(`c`)`[45,47]	$0 \leq s$	9
`AbstractCollection<E>`	`retainAll(`c`)`[45,48]	$0 \leq s$	10
`AbstractCollection<E>`	`toString()`[49]	$0 \leq s$	11
`AbstractList<E>`	`equals(`o`)`[50]	$0 \leq s$	12
`AbstractList<E>`	`listIterator()`[51]	$0 \leq s$	13
`AbstractSequentialList<E>`	`iterator()`[52]	$0 \leq s$	15
`AbstractList<E>`	`hashCode()`[53]	$0 \leq s$	40

[35] Indirect, by calling `addAll(`i`,`c`)` where $i = s$.
[36] Indirect: after check, `new ListItr(index)` is called where `index` $= i$. When $i < s$, `node(`i`)` is called.
[37] Indirect, by calling `addAll(`c`)`.
[38] See section 3.2.2 why `hashCode()` is compliant.
[39] The code containing `Iterator` is obtained by decompiling the class files of interest, using Eclipse.
[40] `forEach(`$action$`)` calls `iterator()`.
[41] `removeIf(`$filter$`)` calls `iterator()`.
[42] `replaceAll(`$operator$`)` calls `listIterator()`.
[43] c is of type `Comparator<? super E> c`
[44] `sort(`c`)` calls `Collections.sort(`$list$`,`c`)`, which calls `listIterator()`.
[45] c is of type `Collection<?> c`
[46] `containsAll(`c`)` calls `iterator()`.
[47] `removeAll(`c`)` calls `iterator()`.
[48] `retainAll(`c`)` calls `iterator()`.
[49] `toString()` calls `iterator()`.
[50] `equals(`o`)` calls `listIterator()`.

```java
default void forEach(Consumer<? super T>
↪  action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

```java
default void forEach(Consumer<? super T>
↪  var1) {
  Objects.requireNonNull(var1);
  Iterator var2 = this.iterator();
  while (var2.hasNext()) {
    Object var3 = var2.next();
    var1.accept(var3);
  }
}
```

```java
public boolean containsAll(Collection<?> c) {
    for (Object e : c)
        if (!contains(e))
            return false;
    return true;
}
```

```java
public boolean containsAll(Collection<?>
↪  var1) {
  Iterator var2 = var1.iterator();
  Object var3;
  do {
    if (!var2.hasNext()) {
      return true;
    }
    var3 = var2.next();
  } while (this.contains(var3));
  return false;
}
```

```java
public int hashCode() {
    int hashCode = 1;
    for (E e : this)
        hashCode = 31*hashCode + (e==null ? 0
        ↪  : e.hashCode());
    return hashCode;
}
```

```java
public int hashCode() {
  int var1 = 1;
  Object var3;
  for (Iterator var2 = this.iterator();
  ↪  var2.hasNext(); var1 = 31 * var1 +
  ↪  (var3 == null ? 0 : var3.hashCode())) {
    var3 = var2.next();
  }
  return var1;
}
```

LISTING 3.4: Enhanced for loop vs Iterator

TABLE 3.5: Callers of spliterator()

Type	Call	Nr
`Collection<E>`	`parallelStream()`	2
`Collection<E>`	`stream()`	4

spliterator() The two methods mentioned in Table 3.5 are non-compliant, as they are assumed to malfunction in case of overflow of `size`. Since this is not confirmed by a test this statement is to be taken with precaution!

3.2.1.3 Other Methods with Bugs

- Method `isEmpty()` (Nr. 8). Returns `true` if $s = 0$, and `false` otherwise. This is wrong when $n > 0$, i.e., in case of overflow.
- Method `subList(int fromIndex, int toIndex)` (Nr. 14). Throws an error in case of overflow.
- Method `size()` (Nr. 35). Returns s. This is what it should do. On the other hand, if this method would return $2^{31} - 1$ in case of overflow, which is the specified behavour according to the Java documentation, then, e.g., `subList()` would not throw an error.

[51] `listIterator()` calls `listIterator(0)`, i.e., `node(i)` is called where $i = 0$.
[52] `iterator()` calls `listIterator(0)`.
[53] `hashCode()` calls `iterator()`.

- Method `toArray()` (Nr. 37). We found that class `ArrayList<E>` has a `private static` field `MAX_ARRAY_SIZE` of type `int` with value `Integer.MAX_VALUE` − 8, accompanied by this Javadoc:
```
/**
 * The maximum size of array to allocate.
 * Some VMs reserve some header words in an array.
 * Attempts to allocate larger arrays may result in
 * OutOfMemoryError: Requested array size exceeds VM limit
 */
```
The restriction is contrary to `toArray()`'s Javadoc, according to which the resulting array contains all of the elements in the list.

- Method `toArray(T[] a)` (Nr. 38). Similar to `toArray()`.

3.2.2 Compliant Methods

All methods numbered with Nrs. 40–58 behave compliant. A note for `hashCode()` (Nr. 40): Looking at the source code of `hashCode()` (see Listing 3.4), it is clear that it suffers from the same bug as the one `equals()` suffers from, i.e., the bug that an iterator has only partial access to the list in case of overflow (see section 3.2.3). However, since the bug is the same, access of elements also is the same (whether or not overflow occurs). That means that x`.equals(`y`)` implies that x`.hashCode() == `y`.hashCode()` for any two lists, x and y. And that's exactly what `hashCode()`'s Javadoc says. See also a corresponding test case in section 3.3.2.1. Nrs. 42–58 do not depend on `size` or an index, and their behavior is like specified, under all circumstances. The latter also holds for method `clear()` (Nr. 41): it uses `size`, but only to set its value to 0, which is fine when the list gets empty.

3.2.3 Reachability

Reachability is mainly determined by non-public query method `Node<E> node(int index)` (see Listing 2.3), which returns the (non-null) Node at the specified element index. It is used by public CRUD methods that are index-based; among them methods that use an instance of type `Iterator`. From Tables 3.3 and 3.4 it follows that this method will be only be called when $0 \leq i < s$ (methods: Nrs. 17, 25, 33, and 34), $0 \leq i \leq s$ (methods: Nrs. 19, 28) or $0 \leq s$ (methods: Nrs. 1, 3, 5, 6, 7, 9, 10, 11, 12, 12, 15, 18, 39, and 40). In other cases its call is prevented, and an error is thrown instead, which makes none of the elements reachable. Let's forget about $i < 0$ since its unlikeliness to happen (at least we didn't encounter it in our analysis). We are interested in the impact of overflow, i.e., $n > 2^{31} - 1$ and thus $k_s > 0$ (see relations 3.4, 3.6, and 3.9), on reachability. For $s < 0$ we observe that 0 out of n elements are reachable.

$s = 0$ 0 out of $n = k_s \cdot 2^{32}$ elements reachable.

$s > 0$ s out of $n = k_s \cdot 2^{32}$ elements reachable. Let $0 \leq i < s$, let $\phi(i)$ denote the node that is the result of the call of `node(`i`)`, and let $u = s/2$,[54] then we can relate

[54] Java's int division.

$\phi(i)$ and $\sigma[i]$:

$$0 \leq i < u \leftrightarrow \phi(i) = \sigma[i] \tag{3.14}$$

$$u \leq i < s \leftrightarrow \phi(i) = \sigma[i + k_s \cdot 2^{32}] \tag{3.15}$$

When $u > 0$ (which implies that $s \geq 2$), then from relations 2.3, 2.4, 3.14, and 3.15 it follows:

$$\sigma[u - 1 + k_s \cdot 2^{32}].\texttt{next} = \phi(u) \wedge \sigma[u - 1 + k_s \cdot 2^{32}] = \phi(u).\texttt{prev} \tag{3.16}$$

From 3.15 it follows:

$$k_s > 0 \leftrightarrow (u \leq i < s \leftrightarrow \phi(i) \neq \sigma[i]) \tag{3.17}$$

Let us assume $k_s > 0$. Furthermore, let's have $u > 0$, then we have a rupture between $\phi(u - 1)$ and $\phi(u)$:

$$\phi(u - 1).\texttt{next} \neq \phi(u) \wedge \phi(u - 1) \neq \phi(u).\texttt{prev} \tag{3.18}$$

Distribution Let $\sigma[x \,..\, y] = \emptyset$ when $y \leq x$, $\sigma[x \,..\, y] = \sigma[x]$ when $y = x + 1$, and $\sigma[x \,..\, y] = \sigma[x \,..\, y - 1] \cup \sigma[y - 1]$ when $y > x + 1$.[55] Let $v = s - u$, let $U = \sigma[0 \,..\, u]$ (compare 3.14) and $V = \sigma[k_s \cdot 2^{32} + u \,..\, k_s \cdot 2^{32} + s]$ (compare 3.15). $\texttt{length}(U) = u$, $\texttt{length}(V) = s - u$. Table 3.6 shows examples of the distribution of $\phi(0), \phi(1) \,..\, \phi(s - 1)$ over $\sigma[0 \,..\, n]$ for $1 \leq s \leq 6$. Note that when $u = 0$ (and thus $s = 1$), that $\phi(0)$ points to `last`, which only coincides with `first` when $n = 1$, i.e., the no-overflow case ($k_s = 0$).

TABLE 3.6: Distribution examples, for $1 \leq s \leq 6$.

s	u	U	V
1	0	$\sigma[0 \,..\, 0]$[56]	$\sigma[k_s \cdot 2^{32} \,..\, k_s \cdot 2^{32} + 1]$
2	1	$\sigma[0 \,..\, 1]$	$\sigma[k_s \cdot 2^{32} + 1 \,..\, k \cdot 2^{32} + 2]$
3	1	$\sigma[0 \,..\, 1]$	$\sigma[k_s \cdot 2^{32} + 1 \,..\, k_s \cdot 2^{32} + 3]$
4	2	$\sigma[0 \,..\, 2]$	$\sigma[k_s \cdot 2^{32} + 2 \,..\, k_s \cdot 2^{32} + 4]$
5	2	$\sigma[0 \,..\, 2]$	$\sigma[k_s \cdot 2^{32} + 2 \,..\, k_s \cdot 2^{32} + 5]$
6	3	$\sigma[0 \,..\, 3]$	$\sigma[k_s \cdot 2^{32} + 3 \,..\, k_s \cdot 2^{32} + 6]$

Iterator An `Iterator` is associated with a list; it acts like a cursor on that list. When the list is an instance of `LinkedList`, the public method `ListIterator<E> listIterator(int index)`[57] is called, which returns an instance of `ListItr`. Let $V(\texttt{index}) \equiv i$, then the initial position of this 'cursor' is determined by $\phi(i)$. Normally $i = 0$.[58] Starting from $\phi(i)$ one can traverse using `.prev` and `.next` fields, by using methods `prev()` and `next()`, respectively. The reachable elements are all connected and so no rupture here. But the number of reachable is still s, since s is used to check whether there is a next of previous element relative to a specific element in the list.

[55] Example: $\sigma[3 \,..\, 5] = \sigma[3] \cup \sigma[4]$.

[56] $\sigma[x \,..\, x] = \emptyset$

[57] See Nr. 28 of Table 3.1.

[58] Method `iterator()` is inherited from `AbstractSequentialList`. It calls `listIterator()` of `AbstractList` that calls on its turn `listIterator(int index)` of `LinkedList` it self (of course only when `this` concerns an instance of `LinkedList`), where `index` $= 0$.

3.3 Reproduction

Of the 58 methods of `LinkedList`, 39 of them are considered as non-compliant according to the source code based assessment of section 3.2; thus 39 methods cause or exhibit integer overflow bugs. We want to reproduce these bugs (i) to examine whether the results support or dispute the assessment, and (ii) to see in practice how a linked list behaves in case of overflow.

Table 3.7 is a clone of Table 3.1, except that two colums have been added. The findings of the assessment have been noticed in column 'C.' (compliancy): is the method compliant with its contract (✅) or not (❌). For 34 of the 39 non-compliant methods we created one or more test cases. For 5 of the 39 methods it was decided to leave out a test case because of a lack of time. For the 19 methods for which the assessment leaded to the 'compliant' outcome, no test cases were created to 'prove the opposite'. Excluding inherited methods, we find that 18 out of 42 methods are compliant. 'Test' refers to test cases listed in Table 3.8, which were carried out to reveal integer overflow related bugs (except for the methods numbered 1–5).

Table 3.7: Assessment result

Nr	Type	Method	C.	Test
1	Iterable<T>	forEach(Consumer<? super E> action)	❌	
2	Collection<E>	parallelStream()	❌	
3	Collection<E>	removeIf(Predicate<? super E> filter)	❌	
4	Collection<E>	stream()	❌	
5	List<E>	replaceAll(UnaryOperator<E> operator)	❌	
6	List<E>	sort(Comparator<? super E> c)	❌	1–2
7	AbstractCollection<E>	containsAll(Collection<?> c)	❌	3–5
8	AbstractCollection<E>	isEmpty()	❌	6
9	AbstractCollection<E>	removeAll(Collection<?> c)	❌	7–9
10	AbstractCollection<E>	retainAll(Collection<?> c)	❌	10–12
11	AbstractCollection<E>	toString()	❌	13
12	AbstractList<E>	equals(Object o)	❌	14–17
13	AbstractList<E>	listIterator()	❌	18–19
14	AbstractList<E>	subList(int fromIndex, int toIndex)	❌	20
15	AbstractSequentialList<E>	iterator()	❌	21–22
16	LinkedList<E>	add(E e)	❌	23
17	LinkedList<E>	add(int index, E element)	❌	24–25
18	LinkedList<E>	addAll(Collection<? extends E> c)	❌	26–29
19	LinkedList<E>	addAll(int index, Collection<? extends E> c)	❌	30–34
20	LinkedList<E>	addFirst(E e)	❌	35
21	LinkedList<E>	addLast(E e)	❌	36
22	LinkedList<E>	clone()	❌	37
23	LinkedList<E>	contains(Object o)	❌	38

Table 3.7 – continued from previous page

Nr	Type	Method	C.	Test
24	LinkedList<E>	descendingIterator()	✗	39–40
25	LinkedList<E>	get(int index)	✗	41–42
26	LinkedList<E>	indexOf(Object o)	✗	43
27	LinkedList<E>	lastIndexOf(Object o)	✗	44
28	LinkedList<E>	listIterator(int index)	✗	45–46
29	LinkedList<E>	offer(E e)	✗	47
30	LinkedList<E>	offerFirst(E e)	✗	48
31	LinkedList<E>	offerLast(E e)	✗	49
32	LinkedList<E>	push(E e)	✗	50
33	LinkedList<E>	remove(int index)	✗	51–52
34	LinkedList<E>	set(int index, E element)	✗	53–54
35	LinkedList<E>	size()	✗	55
36	LinkedList<E>	spliterator()	✗	56
37	LinkedList<E>	toArray()	✗	57–58
38	LinkedList<E>	toArray(T[] a)	✗	59–60
39	LinkedList<E>	LinkedList(Collection<? extends E> c)	✗	61–62
40	AbstractList<E>	hashCode()	✓	
41	LinkedList<E>	clear()	✓	
42	LinkedList<E>	element()	✓	
43	LinkedList<E>	getFirst()	✓	
44	LinkedList<E>	getLast()	✓	
45	LinkedList<E>	peek()	✓	
46	LinkedList<E>	peekFirst()	✓	
47	LinkedList<E>	peekLast()	✓	
48	LinkedList<E>	poll()	✓	
49	LinkedList<E>	pollFirst()	✓	
50	LinkedList<E>	pollLast()	✓	
51	LinkedList<E>	pop()	✓	
52	LinkedList<E>	remove()	✓	
53	LinkedList<E>	remove(Object o)	✓	
54	LinkedList<E>	removeFirst()	✓	
55	LinkedList<E>	removeFirstOccurrence(Object o)	✓	
56	LinkedList<E>	removeLast()	✓	
57	LinkedList<E>	removeLastOccurrence(Object o)	✓	
58	LinkedList<E>	LinkedList()	✓	

3.3.1 Setup

Normally a client won't keep track of the number of elements added to an instance l of `LinkedList`, since it will rely on s as the correct value representing the number of elements. However, when having test cases aiming at deliberately creating an overflow situation, the client must keep track of n (and k_s). Let $l_0 = L(s_0, n_0)$ denote the state of instance l where $k_s = 0$ (i.e., $n_0 = s_0$). Let $l_1 = L(s_1, n_1)$ be the state of instance l after adding t elements to l_0. To have $k_s > 0$ for l_1, we must have $t > 2^{31} - 1 - s_0$.

In almost all of our test cases $s_0 = 2^{31} - 1$. $t > 0$ is such that $s_1 = -2^{31}$ ($t = 1$, $n_1 = 2^{31}$), $s_1 = 0$ ($t = 2^{31}+1$, $n_1 = 2^{32}$), or $s_1 = v$ ($t = 2^{31}+1+v$, $n_1 = 2^{32}+v$) with $v > 0$ and $v \approx 0$. Let these three situations be marked as S_1, S_2, and S_3, respectively. To reach state l_0 we start with an instance $L(0,0)$, and then have method `add(E e)` called n_0 times. The situation where $n_0 < 2^{31} - 1$ we call S_0.

The Java class `Work` is used to run the test cases. For each test case it has a corresponding method. Each test method uses instance field `linkedList` of type `LinkedList<String>` to call a method under test. If for the method under test an additional collection is needed, a local variable named `c` is used for that purpose.[59] Test methods follow (more or less) the same pattern: It starts with adding n elements to a list of type `linkedList<String>`. For a few methods $n = n_0 \leq 2^{31} - 1$, but for most ones $n = n_1 > 2^{31} - 1$.[60] After that the method under test is called. The expected behavior of the call[61] is printed before the call takes place. Depending on the test case, we have distuingished four cases regarding the content of the individual elements: (i) all elements except the last one are `null` (ii) all elements except first and the last one are `null` (iii) all elements except first and the second one are `null` (iv) all elements are `null`. Properties of `linkedList` (and c) supportive to the test case are printed to the output. Listing 3.5 shows an excerpt of the Java class `Work`. One of the methods that adds elements is shown, viz., the one that creates null elements except the last one.

Each test case in Table 3.8 is numbered, from 1–62 (column 'Nr'), and refers to a method than can be called from a `LinkedList` instance. 'Method' and 'Type' refers to the method name and the type in which the method resides, respectively. 'Description' contains a short description of the use case, where as 'Testscript' shows the name of the script that triggers the test. 'S_i' ($i \in \{0 \ldots 3\}$) refers to the aforementioned situations w.r.t. the different branches for n_0, s_0, n_1, and s_1. The number of different use cases for one method differs from 1–5.

For S_0 and S_1 the minimum requirement is 65 gigabytes of memory, where as for S_2 and S_3 this amounts to 167 gigabytes for the JRE. The latter requires a virtual machine (VM) with (at least) 172 gigabytes of memory.[62] The running Java version is Oracle's JDK8 (build 1.8.0 201-b09) that has the same LinkedList implementation as in OpenJDK 8.

Two additional test cases Later on we decided to have two additional test cases, that didn't fit in the ones listed in Table 3.8. The first one is a test case for `hashCode()`, to show that this method is compliant, i.e., not broken, as stated in section 3.2.2. The other one refers to method `add(E e)` of `LinkedLinked`'s inner class `ListItr`. This is not one of the 57 methods `LinkedLinked`, which is the reason that it was not part of the original set of test cases. However, this test case is of importance since it simply is possible to call this method by first calling (inherited) method `listIterator()` on an instance of `LinkedLinked`, and then call `add(E e)` on the resulting instance of the `listIterator()`, viz., `ListItr`.

[59] E.g., x`.equals(`y`)` and x`.containsAll(`y`)`.

[60] In that case, we could still think of $n_0 \leq 2^{31} - 1$; however that state is not recognizable as such in the log of the run.

[61] Based on the assessment.

[62] For S_0 and S_1 a VM with 68 gigabytes of memory would be sufficient. However, switching the setup of the VM complicates the process of running the test cases, considering their number and processing time. Furthermore, the approach is more clean when all the test cases are performed in a single run, since that ensures that for every test case the source code of the classes underlying the testing process is the same.

3.3.2 Result

All test cases were carried out on a machine in a private cloud (SURFsara[63]), which provides instances that satisfy the system requirements. The appendix [25] shows the result of a single run of all test cases, that takes about 6–7 hours. The result is in accordance with the assessment of section 3.2. In other words: the assessment is supported by the results of the test cases.

3.3.2.1 Some Examples

The first example refers to one of the test cases of Table 3.8. The other examples are the additional test cases.

Test case 4 for equals() [Nr. 17] Listing 3.6 shows an example for the corresponding method along with an equally named class that instantiates `Work` and calls the method. The test script is also shown. The purpose of the test case is to show that, when having collections x and y of type `linkedList<E>`, then x`.equals(`y`)` may yield `true` when x and y have the same non-negative value of `size` ($s > 0$) and different n. Let's take $x = c$, and $y =$ `linkedList`, then for c we have $s = 1$ and $n = 2^{32} + 1$, where as for `linkedList` we have $s = 1$ and $n = 1$. The iterator attached to c can only reach one element (the last one, not the first),[64] which is filled with the same content as the only element of `linkedList`. The output of the test has been logged, and is shown in Listing A.17 of the appendix [25].[65]

Test case for hashCode()

Type: `AbstractList<E>`

Method: `hashCode()`

Testscript: hashCode_testcase_extra_1.sh

Minimum memory requirement for the JRE: 167 gigabytes.

Listing 3.7 shows an example for the corresponding method along with an equally named class that instantiates `Work` and calls the method. The test script is also shown. The purpose of the test case is to show that, when having collections x and y of type `linkedList<E>`, then if x`.equals(`y`)` yields `true`, then their hash codes are the same, as stated in section 3.2.2. Let's take $x = c$, and $y =$ `linkedList`, where x and y have the same non-negative value of `size` ($s > 0$) and different n. For c we have $s = 2$ and $n = 2^{32} + 2$, where as for `linkedList` we have $s = 2$ and $n = 2$. The iterator attached to c can only reach the first and the second element.[64] The first and second element of both c and `linkedList` are filled with the same non-null `String` items. The output of the test has been logged, and is shown in Listing A.64 of the appendix [25].

[63] A center that provides supercomputers to universities and research institutes in the Netherlands.
[64] This follows from section 3.2.3.
[65] Logs of all tests can be found in the appendix.

Test case for add(E e) of ListItr

Type: `AbstractList<E>`

Method: `listIterator()`

Testscript: listIterator_testcase_extra_1.sh

Minimum memory requirement for the JRE: 65 gigabytes.

Listing 3.8 shows an example for the corresponding method along with an equally named class that instantiates `Work` and calls the method. The `LinkedList` instance calls method `iterator()` which returns an instance of `LinkedList`'s internal class `ListItr`, which implements the interface `ListIterator`. As such, it has an `add` method. Calling this method means that a new element is added to the `LinkedList` object. In other words: this method can unconditionally cause overflow of size. The output of the test has been logged, and is shown in Listing A.63 of the appendix [25].

In section 3.3.1 we stated that method `add(E e)` is called n_0 times (to reach state l_0), before calling the method under test. By this method we meant method `add(E e)` of `LinkedList`. In *this* case, this was not possible, since that triggers an exception of type `ConcurrentModificationException`, as soon as the method under test is called after the other `add(E e)` method had been called before. So in this test case only the method under test must be called to add elements.

```
public class Work {

  // INT_MAX_VALUE should equal Integer.MAX_VALUE (2^31-1) but may differ for
  // testing purposes.
  static int INT_MAX_VALUE = Integer.MAX_VALUE;
  private static String thisInstance = "linkedList";
  private static String anotherInstance = "c";
  private LinkedList<String> linkedList;

  private void pre(String linkedListMethod, String testcaseMethod) {
    Helper.showTitle(testcaseMethod, linkedListMethod);
    this.linkedList = new LinkedList<>();
    Helper.showSize(thisInstance, this.linkedList, this.linkedList.size());
  }
  private static Collection<String> createCollection() {
    Collection<String> c = new LinkedList<>();
    Helper.showSize(anotherInstance, c, c.size());
    return c;
  }
  private void showConditionally(Collection<String> c, long longSize, long upper) {
    if (this.doShow(longSize, upper)) {
      Helper.showContext(c);
    }
  }
  boolean doShow(long longSize, long upper) {
    return longSize == upper || longSize % (INT_MAX_VALUE + (long) 1) == 0;
  }
  private static void beforeFilling(Collection<String> c, long upper, String instance) {
    System.out.println("First, set data ...");
    Helper.showSizeLongVsInt(upper, instance);
    Helper.showTime();
    Helper.showInitial(c);
  }
  private void afterFilling(Collection<String> c, long upper) {
    Helper.showSize((c == this.linkedList ? thisInstance : anotherInstance), c, upper);
    Helper.showTime();
    System.out.println("Setting data done");
  }
  private void fillListWithLastElementNonEmpty(Collection<String> c, long upper) {
    beforeFilling(c, upper, c == this.linkedList ? thisInstance : anotherInstance);
    long longSize = 0;
    while (longSize < upper) {
      if (longSize == upper - 1) {
        c.add(String.valueOf(++longSize));
        this.showConditionally(c, longSize, upper);
      } else {
        c.add(null);
        this.showConditionally(c, ++longSize, upper);
      }
    }
    Helper.showEmptyString("Non-last elements of " + (c == this.linkedList ? thisInstance :
    ↪  anotherInstance));
    Helper.showLongAsStringValue("The last element of " + (c == this.linkedList ? thisInstance
    ↪  : anotherInstance),
        upper);

    this.afterFilling(c, upper);
  }
  ...

}
```

Listing 3.5: Structure of Work

```
void equals_testcase_4() {
  this.pre("AbstractList::public boolean equals(Object o)", "equals_testcase_4");
  Collection<String> c = createCollection();
  String call = Helper.getCallString(thisInstance, "equals", anotherInstance);
  Helper.showMessage("Call to be executed", call);
  long upper = INT_MAX_VALUE + (long) 1 + INT_MAX_VALUE + (long) 1 + (long) 1;

  this.fillListWithLastElementNonEmpty(c, upper);

  Helper.showMessage(Helper.getCallString(thisInstance, "add", upper),
      this.linkedList.add(String.valueOf(upper)));
  Helper.showSize(thisInstance, this.linkedList, this.linkedList.size());

  // iterator over this.linkedList: s == 1
  // reachable: node[0] = first, since s == n (k == 0)
  // iterator over c: s == 1
  // reachable: node[0] = last
  // first of this.linkedList equals String.valueOf(upper), as does last of c
  Helper.showExpectedBehavior(call, true + ", since iterators over " + thisInstance + " and " +
  ↪  anotherInstance
      + " both reach only one element that is in both cases the same, i.e., " +
      ↪  String.valueOf(upper) + ".");
  Helper.showTime();
  System.out.println(call + " in progress ...");
  Helper.showMessage(call, this.linkedList.equals(c));
  System.out.println(call + " done");
  Helper.showTime();
}
public class equals_testcase_4 {
  public static void main(String[] args) {
    new Work().equals_testcase_4();
  }
}
```

```
java -Xms167G -Xmx167G -Xloggc:equals_testcase_4_167G_OOP_gc.log
↪  -XX:+PrintClassHistogramBeforeFullGC -XX:+PrintGC -XX:+PrintGCDateStamps
↪  -XX:+PrintGCDetails -XX:+PrintGCTimeStamps equals_testcase_4
```

Listing 3.6: Test case 4 for equals()

```
void hashCode_testcase_extra_1() {
  this.pre("AbstractList::public int hashCode()", "hashCode_testcase_extra_1");
  Collection<String> c = createCollection();
  String call = Helper.getCallString(anotherInstance, "hashCode");
  Helper.showMessage("Call to be executed", call);
  long upper = INT_MAX_VALUE + (long) 1 + INT_MAX_VALUE + (long) 1 + (long) 2;

  this.fillListWithFirstAndSecondElementNonEmpty(c, upper);
  // iterator over c: s == 2
  // reachable: node[0] + node[0].next = first + first.next
  // other nodes are not reachable, i.e., thus are neglected
  Helper.showTime();
  System.out.println(call + " in progress ...");
  Helper.showMessage(call, c.hashCode());
  System.out.println(call + " done");
  Helper.showTime();

  call = Helper.getCallString(thisInstance, "hashCode");
  Helper.showMessage("Call to be executed", call);
  upper = 2;

  this.fillListWithFirstAndSecondElementNonEmpty(this.linkedList, upper);
  // iterator over this.linkedList: s == 2
  // reachable: node[0] + node[0].next = first + first.next
  Helper.showTime();
  System.out.println(call + " in progress ...");
  Helper.showMessage(call, this.linkedList.hashCode());
  System.out.println(call + " done");
  Helper.showTime();

  System.out.println("String.valueOf(c.hashCode()).equals(String.valueOf(this.linkedList.hashCode()))
  ↪  = "
      + String.valueOf(c.hashCode()).equals(String.valueOf(this.linkedList.hashCode())));

  System.out.println(String.valueOf("c.equals(this.linkedList) = " +
  ↪  c.equals(this.linkedList)));

}
public class hashCode_testcase_extra_1 {

  public static void main(String[] args) {
    new Work().hashCode_testcase_extra_1();
  }

}
```

```
java -Xms167G -Xmx167G -Xloggc:hashCode_testcase_extra_1_167G_OOP_gc.log
↪  -XX:+PrintClassHistogramBeforeFullGC -XX:+PrintGC -XX:+PrintGCDateStamps
↪  -XX:+PrintGCDetails -XX:+PrintGCTimeStamps hashCode_testcase_extra_1
```

Listing 3.7: Test case for hashCode()

```
void listIterator_testcase_extra_1() {
  this.pre("AbstractList::public ListIterator<E> listIterator()",
  ↪  "listIterator_testcase_extra_1");
  String call = Helper.getCallString(Helper.getCallString(thisInstance, "listIterator"), "add",
  ↪  null);
  Helper.showMessage("Call to be executed", call);
  ListIterator<String> it = this.linkedList.listIterator();
  long upper = INT_MAX_VALUE;

  this.fillListWithEmptyElements(it, this.linkedList, upper);

  Helper.showExpectedBehavior(call,
      true + ", since " + Helper.getMethodString("add", "e") + " permits overflow to occur.");
  Helper.showTime();
  System.out.println(call + " in progress ...");
  try {
    it.add(null);
  } catch (Throwable t) {
    t.printStackTrace();
  } finally {
    System.out.println(call + " done");
    Helper.showSize(thisInstance, this.linkedList, upper + (long) 1);
    Helper.showTime();
  }
}
public class listIterator_testcase_extra_1 {

  public static void main(String[] args) {
    new Work().listIterator_testcase_extra_1();
  }

}
```

```
java -Xms65G -Xmx65G -Xloggc:listIterator_testcase_extra_1_65G_OOP_gc.log
↪  -XX:ObjectAlignmentInBytes=32 -XX:+PrintClassHistogramBeforeFullGC -XX:+PrintGC
↪  -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
↪  -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseG1GC
↪  listIterator_testcase_extra_1
```

Listing 3.8: Test case for listIterator()

TABLE 3.8: Reproduction. Minimum memory requirement for the JRE: S_0, $S_1 = 65$ gigabytes; S_2, $S_3 = 167$ gigabytes.

Nr	Type	Method	Description	Testscript	S_i
1	List<E>	sort(Comparator<? super E> c)	`this.size < 0`: NegativeArraySizeException	sort_testcase_1.sh	S_1
2			OutOfMemoryError	sort_testcase_2.sh	S_0
3	AbstractCollection<E>	containsAll(Collection<?> c)	`c.size < 0`: IndexOutOfBoundsException	containsAll_testcase_1.sh	S_1
4			`c.size >= 0` and $n > 0$: may give questionable result	containsAll_testcase_2.sh	S_3
5			`this` has an element associated with index -1: may give questionable/erroneous result	containsAll_testcase_3.sh	S_2
6		isEmpty()	`this.size == 0` and $n > 0$: questionable/erroneous result	isEmpty_testcase_1.sh	S_2
7		removeAll(Collection<?> c)	`this.size < 0`: IndexOutOfBoundsException	removeAll_testcase_1.sh	S_1
8			`this.size >= 0` and $n > 0$: may give questionable result	removeAll_testcase_2.sh	S_3
9			`c` has an element associated with index -1: may give questionable/erroneous result	removeAll_testcase_3.sh	S_2
10		retainAll(Collection<?> c)	`this.size < 0`: IndexOutOfBoundsException	retainAll_testcase_1.sh	S_1
11			`this.size >= 0` and $n > 0$: may give questionable result	retainAll_testcase_2.sh	S_3
12			`c` has an element associated with index -1: may give questionable/erroneous result	retainAll_testcase_3.sh	S_2

Table 3.8 – continued from previous page

Nr	Type	Method	Description	Testscript	S_i
13		toString()	`this.size >= 0` and $n > 0$: may give questionable result	toString_testcase_1.sh	S_3
14	AbstractList<E>	equals(Object o)	`this.size < 0`: IndexOutOfBoundsException	equals_testcase_1.sh	S_1
15			`this.size >= 0` and $n > 0$: questionable result	equals_testcase_2.sh	S_3
16			`o.size` is < 0: IndexOutOfBoundsException	equals_testcase_3.sh	S_1
17			`o.size` is $>= 0$ and $n > 0$: questionable result	equals_testcase_4.sh	S_3
18		listIterator()	`this.size < 0`: IndexOutOfBoundsException	listIterator_testcase_1.sh	S_1
19			`this.size >= 0` and $n > 0$: gives questionable/erroneous result	listIterator_testcase_2.sh	S_3
20		subList(int fromIndex, int toIndex)	`this.size < 0`: IndexOutOfBoundsException	subList_testcase_1.sh	S_1
21	AbstractSequentialList<E>	iterator()	`this.size < 0`: IndexOutOfBoundsException	iterator_testcase_1.sh	S_1
22			`this.size >= 0` and $n > 0$: gives questionable/erroneous result	iterator_testcase_2.sh	S_3
23	LinkedList<E>	add(E e)	let overflow of `this` occur	add_testcase_1.sh	S_1
24		add(int index, E element)	`this.size < 0`: IndexOutOfBoundsException	add_i_testcase_1.sh	S_1

Table 3.8 – continued from previous page

Nr	Type	Method	Description	Testscript	S_i
25			`this.size >= 0` and $n > 0$: may give questionable result or IndexOutOfBoundsException	add_i_testcase_2.sh	S_3
26		addAll(Collection<? extends E> c)	`this.size < 0`: IndexOutOfBoundsException	addAll_testcase_1.sh	S_1
27			let overflow of `this` occur	addAll_testcase_2.sh	S_1[66]
28			`c.size < 0`: NegativeArraySizeException	addAll_testcase_3.sh	S_1
29			`c.size >= 0` and $n > 0$: ArrayIndexOutOfBoundsException	addAll_testcase_4.sh	S_3
30		addAll(int index, Collection<? extends E> c)	`this.size < 0`: IndexOutOfBoundsException	addAll_i_testcase_1.sh	S_1
31			let overflow of `this` occur	addAll_i_testcase_2.sh	S_1[66]
32			`c.size < 0`: NegativeArraySizeException	addAll_i_testcase_3.sh	S_1
33			`c.size >= 0` and $n > 0$: ArrayIndexOutOfBoundsException	addAll_i_testcase_4.sh	S_3
34			`this.size >= 0` and $n > 0$: questionable result	addAll_i_testcase_5.sh	S_3
35		addFirst(E e)	let overflow of `this` occur	addFirst_testcase_1.sh	S_1
36		addLast(E e)	let overflow of `this` occur	addLast_testcase_1.sh	S_1
37		clone()	permits clone to be oversized	clone_testcase_1.sh	S_1

[66] $s_1 = -2^{31} + 1$

Table 3.8 – continued from previous page

Nr	Type	Method	Description	Testscript	S_i
38		contains(Object o)	`o` is associated with index -1: may give questionable/erroneous result	contains_testcase_1.sh	S_2
39		descendingIterator()	`this.size < 0`: NO IndexOutOfBoundsException!	descendingIterator_testcase_1.sh	S_1
40			`this.size >= 0` and $n > 0$: gives questionable/erroneous result	descendingIterator_testcase_2.sh	S_3
41		get(int index)	`this.size < 0`: IndexOutOfBoundsException	get_testcase_1.sh	S_1
42			`this.size >= 0` and $n > 0$: may give questionable/erroneous result or IndexOutOfBoundsException	get_testcase_2.sh	S_3
43		indexOf(Object o)	`o` is associated with index -1: may give questionable/erroneous result	indexOf_testcase_1.sh	S_2
44		lastIndexOf(Object o)	`o` is associated with index -1: may give questionable/erroneous result	lastIndexOf_testcase_1.sh	S_2
45		listIterator(int index)	`this.size < 0`: IndexOutOfBoundsException	listIterator_i_testcase_1.sh	S_1
46			`this.size >= 0` and $n > 0$: gives questionable/erroneous result	listIterator_i_testcase_2.sh	S_3
47		offer(E e)	let overflow of `this` occur	offer_testcase_1.sh	S_1
48		offerFirst(E e)	let overflow of `this` occur	offerFirst_testcase_1.sh	S_1

Table 3.8 – continued from previous page

Nr	Type	Method	Description	Testscript	S_i
49		offerLast(E e)	let overflow of `this` occur	offerLast_testcase_1.sh	S_1
50		push(E e)	let overflow of `this` occur	push_testcase_1.sh	S_1
51		remove(int index)	`this.size < 0`: IndexOutOfBoundsException	remove_testcase_1.sh	S_1
52			`this.size >= 0` and $n > 0$: may give questionable/erroneous result or IndexOutOfBoundsException	remove_testcase_2.sh	S_3
53		set(int index, E element)	`this.size < 0`: IndexOutOfBoundsException	set_testcase_1.sh	S_1
54			`this.size >= 0` and $n > 0$: may give questionable/erroneous result or IndexOutOfBoundsException	set_testcase_2.sh	S_3
55		size()	$n > 0$: gives questionable/erroneous result	size_testcase_1.sh	S_1
56		spliterator()	`this.size < 0`: `trySplit()` returns `null`	spliterator_testcase_1.sh	S_1
57		toArray()	`this.size < 0`: NegativeArraySizeException	toArray_testcase_1.sh	S_1
58			OutOfMemoryError	toArray_testcase_2.sh	S_0
59		toArray(T[] a)	`this.size < 0`: NegativeArraySizeException	toArray_a_testcase_1.sh	S_1
60			OutOfMemoryError	toArray_a_testcase_2.sh	S_0
61		LinkedList(Collection<? extends E> c)	`c.size < 0`: NegativeArraySizeException	LinkedList_testcase_1.sh	S_1

Table 3.8 – continued from previous page

Nr	Type	Method	Description	Testscript	S_i
62			`c.size >= 0` and $n > 0$: ArrayIndexOutOfBoundsException	LinkedList_testcase_2.sh	S_3

3.4 Fixing Linked List

There are multiple directions to (partially) fix the overflow bug: *long*, *BigInteger*, *capping size*, and *bounded*.

Long The root cause of the bug is the `int` (signed 32-bit integer) overflow of the `size` field. We could clone the `size` field to a new field of type `long`. Moreover, we could clone/overload all methods that deal with `size` or indices of type `int`. Additional: mirror the collection interfaces to expose methods that use `long` indices. For example, one could define `LongCollection` which has a method `longSize()`; a `LongList` with methods `add(long,Object)`, `longIndexOf(Object)`, `set(long,Object)` and others; and a `LongListIterator` with methods such as `nextLongIndex()`. This new interface hierarchy extends from the 32-bit Collection interfaces, but additionally provides access to the underlying `long` indices. The `long` may also overflow, but this is not a problem for 64-bit machines that will run out of memory before the field can overflow.

BigInteger To deal with even larger collections, one could imagine that another mirror hierarchy of collection interfaces can be defined such as `BigCollection` that takes and returns `BigIntegers` in a similar manner. Although the `BigInteger` type could be used to hold the number of elements in the list, and theoretically `BigInteger` type has no limitations that may cause an overflow, there are two drawbacks for verification: it is difficult to relate `BigInteger` instances to JML's `\bigint` type that represents the mathematical integers, and `BigInteger` may still be limited by available memory.

Capping size In this solution, `size` equals `Integer.MAX_VALUE` when there are more than `Integer.MAX_VALUE` number of elements. This is conform what the Javadoc states on size. This requires a small modification in methods that add one or more elements to the list: A check is required in order to prevent `size` having an overflow. The problem with $i = -1$ still can occur in `indexOf()` and `lastIndexOf()`, unless search is restricted to the first `Integer.MAX_VALUE` elements. Index-based methods will no longer throw an error because of a negative size. Reachability w.r.t. `node(int index)` is restricted to the first `Integer.MAX_VALUE` elements. However, that's still better than the current behavior. Another modification that is needed is: if `size` equals `Integer.MAX_VALUE`, and a method is called that removes an element, before removing the element, it has to checked whether `node(size - 1)` is the last node, i.e., if `node(size - 1) == last`. If so, we know that `size` has to be decreased with 1, otherwise `size` must stay the same. This solution mitigates the negative impact of having more than `Integer.MAX_VALUE` elements, at the cost of small modifications.

Bounded In the *bounded* solution, we ensure that the overflow of `size` never occurs. Whenever elements would be added that cause the size field to overflow, the operation throws an exception and leaves the list unchanged. As the exception is triggered right before the overflow would otherwise occur, the value of `size` is guaranteed to be bounded by `Integer.MAX_VALUE`, i.e., it never becomes negative. This solution requires a slight adaptation of the implementation: for methods that increase the `size` field, only one additional check has to be performed before a `LinkedList` instance is modified. This checks whether the result of the method causes an overflow of the `size` field. Under this condition, an `IllegalStateException` is thrown. We distinguish between methods that add one element and methods that add a collection of elements.

Adding one element Only in states where `size` is less than `Integer.MAX_VALUE`, it is acceptable to add a single element to the list. The method which performs the check is shown in Listing 3.9. A method that adds an element must first

```
private void checkSize() {
  if (isMaxSize())
    throw new IllegalStateException("Not enough space left in List to add new elements");
}
private boolean isMaxSize() {
  return size == Integer.MAX_VALUE;
}
```

Listing 3.9: Check for adding one element.

call method `checkSize()`. If $s = 2^{31} - 1$ holds, `checkSize()` throws an exception. Otherwise, it terminates normally, and the method that has called `checkSize()` must add the element to the list.

Adding a collection Only in states where the result of increasing `size` with the size of a specified collection does not lead to overflow of `size`, it is acceptable to add the collection to the list. To this end we have method `checkSize(int s)` where the argument for parameter `s` will be the size of the specified collection. Listing 3.10 shows how this works.

```
private void checkSize(int s) {
  if (!enoughSpace(s))
    throw new IllegalStateException("Not enough space left in this list to add new elements.");
}
private boolean enoughSpace(int s) {
  return 0 <= s && s <= Integer.MAX_VALUE - size();
}
public boolean addAll(int index, Collection<? extends E> c) {
  checkSize(c.size());
  ...
  return true;
}
```

Listing 3.10: Check for adding a collection of elements.

Adaptation of Linked List Solutions *long* and *BigInteger* both are not compatible with the current implementation of the framework. *Capping size* is compatible, and it mitigates negative impact in the case when having more than `Integer.MAX_VALUE` elements. It also stays reasonably close to the original implementation of `LinkedList`. Still: This is not a full-fledged solution. The *bounded* solution also stays reasonably close to the original implementation of `LinkedList` and does not leave any behavior unspecified. This approach is followed in chapter 4: we present the class `BoundedLinkedList`, which, when compared to `LinkedList`, contain a few slight modifications that must prevent overflow (see Listings 3.9 and 3.10).

Chapter 4

Formal Specification and Verification of adapted Linked List

We created the class `BoundedLinkedList`, a modified clone of `LinkedList`. The modification must avoid overflow O_s, i.e., these properties must hold during the lifetime of an instance of type `BoundedLinkedList`:

$$\neg O_s \equiv s = n \wedge n < 2^{31} \quad \text{(3.3 revisited)}$$
$$\neg O_s \rightarrow s \geq 0 \quad \text{(3.5 revisited)}$$
$$\neg O_s \leftrightarrow k_s = 0 \quad \text{(3.10 revisited)}$$
$$k_s = 0 \rightarrow 0 \leq i \leq 2^{31} - 2 \quad \text{(3.12 revisited)}$$

To enforce the absence of overflow, the new class contains additional methods (in comparison to the original `LinkedList`), which were shown in Listings 3.9 and 3.10. A method that adds an element must first call method `checkSize()`. If $s = 2^{31} - 1$ holds, `checkSize()` throws an exception. Otherwise, it terminates normally, and the method that has called `checkSize()` must add the element to the list. We want this construct to be *formally verified*. More over: we want to verify as many as possible methods of `BoundedLinkedList`. To this end, we use KeY, a system that can prove correctness of Java programs. The version of KeY that is used is 2.6.3.[67] KeY's core is a theorem prover that is able to model the Java integer overflow semantics. This is essential, as we want to prove the absence of overflow. Another modification that has made in comparison to the original `LinkedList`, is that generics have been stripped, as KeY cannot work with it.[68] An available transformation tool is used, viz., the Remove Generics feature that can be used in Eclipse.[69]

For methods that we are going to verify with KeY, specifations must be written in JML, that can be considered as *contracts* for KeY. JML allows for the specification of normal behavior as for exceptional behavior as well. For both types of behavior a `requires` clause is used, written as `requires` P, where P is a proposition that must hold in the prestate of the method, i.e., P is the precondition. When a method terminates normally, we can write `ensures` Q, where Q is a proposition that must hold after termination, i.e., Q is the postcondition. Let's have a method `m()` which we want to specify. Let $\{\texttt{m}_n\texttt{()}\}$ denote normal execution of `m()`. Let P_n be a proposition

[67] See https://formal.iti.kit.edu/key/download/releases/2.6.3/. Internal build number: key-2.6.3_7d3deab0763c88edee4f7a08e604661e0dbdd450

[68] This does not affect the validity of the verification, since 'Generic type information is present only at compile time, after which it is erased by the compiler.' (see https://docs.oracle.com/javase/8/docs/technotes/guides/language/generics.html).

[69] Using version 2.6.3 of KeY, the available KeY-features that can be installed may be retrieved from http://formal.iti.kit.edu/key/download/releases/2.6.3/eclipse/site.xml.

associated with normal execution of `m()`. We can write:

$$P_n \rightarrow \{\texttt{m}_n\texttt{()}\}Q \tag{4.1}$$

Thus: *if* P_n holds in the prestate of $\{\texttt{m}_n\texttt{()}\}$, *then* $\{\texttt{m}_n\texttt{()}\}$ has to ensure that Q holds after termination. Exceptional behavior: P_e, $\{\texttt{m}_e\texttt{()}\}$. We will only encounter cases where we have two signals clauses, of the following form: `signals_only` E, and `signals (`E `e) true`, where E is an exception. Let's put these together and call them S. We can write:

$$P_e \rightarrow \{\texttt{m}_e\texttt{()}\}S \tag{4.2}$$

For a method that uses `checkSize()`, we have $P_n \equiv s \neq 2^{31}-1$, $P_e \equiv s = 2^{31}-1$, and an S_e where $E =$ `IllegalStateException`.[70] The code is still allowed to throw an error like `OutOfMemoryError` or a `ClassNotFoundError`. In JML side-effects in specifications are forbidden. Specifications with `normal_behavior` implicitly have the clause `signals (java.lang.exception) false` so the method must not throw an exception. Specifications with `exceptional_behavior` implicitly have the clause `ensures false` so the method must not terminate normally. Let's take the method `add (E e)` (see Listing 2.2, but think of it like it is extended with a call of `checkSize()`) as an example to establish what we need to formulate Q. Let's have $0 \leq i < s$ and let seq denote an index based sequence of nodes, such that seq`[`i`]` refers to the $(i+1)^{\texttt{th}}$ element in the linked list. Let `length(`seq`)` be the length of seq, i.e., its number of nodes. Let $seq_{\texttt{old}}$ denote the state of seq in the prestate of the method. Let $n_{\texttt{e}}$ denote the new node that has been created to contain `e`. We want to be able to express something like this (in pseudo code):

$$\begin{aligned} seq = \texttt{concat(}seq_{\texttt{old}}\texttt{,}seq\texttt{[length(}seq\texttt{)}-1\texttt{])} \wedge seq\texttt{[length(}seq\texttt{)}-1\texttt{].item} \\ = n_{\texttt{e}}\texttt{.item} \end{aligned} \tag{4.3}$$

To that end, we will use the abstract data type `\seq` for finite (but unbounded) sequences that is predefined in KeY-JML. It has an attribute `length` and it has operations that helps us with specifying methods, e.g., for concatenation and access. We will use this type for the ghost instance field `nodeList`. Thus the declaration

```
//@ private ghost \seq nodeList;
```

has been added to `BoundedLinkedList`. A ghost field (or local variable) can be used to cache the internal state of an instance. Compare to `size`. `size` is a property that is cached since it follows *indirectly* from the state of the list. When the list is altered in such a way that it affects the number of elements, `size` is modified accordingly. A ghost field is updated by using the `set` command. Thus: we use a `set` command for updating `nodeList` where that is necessary.

When a method calls another method, there are a few options w.r.t. to verification. When the source code of the callee is not available, a so called *stub* can be used. A stub is an abstract implementation of a method with a default contract, i.e., `requires true` and `ensures true`. KeY's stub generator[71] creates them automatically. In the configuration of a KeY project, KeY can be instructed where to search for (additional) sources, including stubs. We use stubs for other classes that

[70] This follows from Listing 3.9.

[71] The Stubby feature that can be used in Eclipse.

`BoundedLinkedList` depends on, such as for the inherited interfaces and abstract super classes. We think that is a good choice, since (1) the methods in these types will not cause overflow themselves (2) we want to restrict verification to the linked list it self, to keep it feasible. We assume for exceptions that their constructors are *pure*, i.e., they may not have any (visible) side effects. An important stub contract is the equality method `equals()` of the super class `Object`, which we have adapted (see Listing 4.1): we assume that every object has a side-effect free, terminating and deterministic implementation of its equality method.

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

Listing 4.1: Stub for Object.equals()

When method x calls method y, and both methods are part of `BoundedLinkedList`, we have as options:

(1) Verify y and use its contract when verifying x. If x has been verified this way, and afterwards y is verified again because of a modification, x need not be verified over again if the contract of y has not been modified. A contract of a method can be reused in every method where it is used.

(2) Do not verify y and when verifying x, the call of y will be replaced by its source code,[72] where parameter binding will be taken into account.

We prefer the first choice, for obvious reasons. An exception is the private method `outOfBoundsMsg(int index)`, that is used by `checkElementIndex(int index)` and `checkPositionIndex(int index)`. It returns a string that is used in a constructor of `IndexOutOfBoundsException`. There was made no attempt to verify it: KeY has introduced a kind of 'ghost' field for Java string objects that assigns each string a finite sequence of characters based on the `\seq` type. This makes it needlessly complicated. And there is no need to verify this method in order to verify both callers.

Non-public method `linkBefore(Object e, Node succ)` creates a new node that stores `e` in its `item` field, and inserts that new node before a node which is referred to by `succ`. If we want to write a contract for this method, we must know the position of `succ`. Similar: Non-public method `unlink(Node x)` removes the node, which is referred to by `x`, from the list. To 'pass on' the position from caller to callee, we will use the ghost instance field `nodeIndex`. Thus the declaration

```
//@ private ghost \bigint nodeIndex;
```

has been added to `BoundedLinkedList`. The caller must use the `set` command to let `nodeIndex` have the value that corresponds to the position of the node that plays a role to the callee. So we have that ghost field `nodeIndex` plays two different roles: the caller *sets* it and the callee *uses* it, without altering it. But KeY cannot know if it's altered or not. It has to be told it is not. This can be done by taking `nodeIndex == \old(nodeIndex)` as a conjunct of the postcondition of the contract of the callee. By altering the post condition, we force KeY to have `nodeIndex` stay

[72] Inlining the body of the called method.

out of the *frame* of `linkBefore(Object e, Node succ)`, i.e., to stay out of the set of locations to which the method may write at most.

It may happen that a specification turns out to be incorrect in the process of verifying the method. In that case, the specification must be altered after which the verification process has to be started over again. The amount of rework for the verification depends on the situation. In the ultimate case rework is total. Example: in a previous version of the contract of `linkBefore(Object e, Node succ)`, the proposition `nodeIndex == \old(nodeIndex)` was not part of the postcondition. At that time, the proof process got stuck; an examination of the proof obligation revealed that it could only be closed under the condition that `nodeIndex` stays unaffected.

4.1 Class Contract

The properties mentioned in section 2.2, viz., 'Unique head and tail' (see properties 2.1 and 2.2), and 'Connectivity' (see properties 2.3 and 2.4), can be considered as properties that must hold for a linked list, prior and next to a call of a method (see section 2.2.1). By writing these properties in JML in a *class invariant* (see Listing 4.2, and explained for each line in Table 4.1), they implicitly act as pre- and postconditions for all methods. Property 'Acyclicity' (see section 2.3) follows from the

```
1  /*@ invariant
2    @   nodeList.length == size &&
3    @   nodeList.length <= Integer.MAX_VALUE &&
4    @   (\forall \bigint i; 0 <= i < nodeList.length;
5    @       nodeList[i] instanceof Node) &&
6    @   ((nodeList == \seq_empty && first == null && last == null)
7    @    || (nodeList != \seq_empty && first != null &&
8    @        first.prev == null && last != null &&
9    @        last.next == null && first == (Node)nodeList[0] &&
10   @        last == (Node)nodeList[nodeList.length-1])) &&
11   @   (\forall \bigint i; 0 < i < nodeList.length;
12   @       ((Node)nodeList[i]).prev == (Node)nodeList[i-1]) &&
13   @   (\forall \bigint i; 0 <= i < nodeList.length-1;
14   @       ((Node)nodeList[i]).next == (Node)nodeList[i+1]);
15   @*/
```

Listing 4.2: Class invariant for BoundedLinkedList

other properties. This property will be needed for the verification of some methods; in those cases we let KeY infer this property by following the same reasoning as in section 2.3. Thus, we leave it deliberately out of the invariant.

4.2 Method Contracts (Selection)

We show the annotated source code of 4 public methods. Where such a method calls another method of `BoundedLinkedList`, we show the annotated source code of that method too. For most of the methods shown, a table shortly mentions some details about the specifications.

4.2.1 add(Object e)

Listing 4.3 shows method `add(Object e)` of `BoundedLinkedList` annotated with JML. If `size == Integer.MAX_VALUE`, an exception is thrown by `checkSize()` (see Listing 4.4. Otherwise, it calls (non-public) method `linkLast(Object e)` (see Listing 4.5). Note the `set` command at the end of `linkLast(Object e)`. Tables 4.2 and 4.3 show comments on the contracts.

TABLE 4.1: Invariant explained

Line	Description
2–3	`nodeList` represents the nodes in the list; as such its length equals s and must be bounded to $2^{31} - 1$.
4–5	The elements of `\seq` are not typed. For this reason, sequence access always needs to be preceded by a type cast. The type cast of the element implies that the element is not `null`. The JML type `\bigint` represents the mathematical integers $\mathbb{Z}$.
6	`nodeList == \seq_empty` $\leftrightarrow$ `nodeList.length == 0`. `first` and `last` are null if and only if (iff) the list is empty.
7–10	Chain (non-empty list) $\rightarrow$ unique head and tail. Note the cast to type `Node` of elements.
11-14	Mutual connectivity. Type casting also here.

```
1  // implements java.util.Collection.add
2  /*@
3    @ also
4    @ public normal_behavior
5    @   requires
6    @     nodeList.length + (\bigint)1 <= Integer.MAX_VALUE;
7    @   ensures
8    @     nodeList == \seq_concat(\old(nodeList),
9    @       \seq_singleton(nodeList[nodeList.length-1])) &&
10   @     ((Node)nodeList[nodeList.length-1]).item == e &&
11   @     \result;
12   @ public exceptional_behavior
13   @   requires
14   @     nodeList.length == Integer.MAX_VALUE;
15   @   signals_only IllegalStateException;
16   @   signals (IllegalStateException e) true;
17   @*/
18 public boolean add(/*@ nullable @*/ Object e) {
19     checkSize(); // new
20     linkLast(e);
21     return true;
22 }
```

LISTING 4.3: add(Object e) with check on size (annotated with JML)

4.2.2 remove(int index)

Listing 4.6 shows method `remove(int index)` of `BoundedLinkedList`. It checks if $0 \leq$ `index` $<$ `nodeList.length`. If not, an exception is thrown by `checkElementIndex(index)` (see Listing 4.7). Otherwise, it calls `unlink(Node x)` (see Listing 4.9), where x is determined by the result of the call of `node(index)` (see Listing 4.8). Note the `set` command at the beginning of `unlink(Node x)`. Tables 4.4, 4.5, and 4.6 show comments on the contracts. `unlink(Node x)` is one of the two methods that uses ghost field `nodeIndex` in its contract — and in its `set` command. The value of `nodeIndex` is set in the remove method.

4.2.3 set(int index, Object element)

Listing 4.10 shows method `set(int index, Object element)` of `BoundedLinkedList` annotated with JML. It replaces the element at the specified position in this list with the specified element, and returns to the caller the element as it was in the prestate. Note the absence of the `set` command. Table 4.7 shows comments on the contract.

```
// new method, not in LinkedList
/*@
  @ 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;
  @*/
private /*@ strictly_pure @*/ void checkSize() {
  if (isMaxSize())
    throw new IllegalStateException("Not enough space left in List to add new elements");
}
// new method, not in LinkedList
/*@
  @ private normal_behavior
  @   ensures \result <==> nodeList.length == Integer.MAX_VALUE;
  @*/
private /*@ strictly_pure @*/ boolean isMaxSize() {
    return size == Integer.MAX_VALUE;
}
```

Listing 4.4: checkSize() (annotated with JML)

```
1  /*@
2    @ normal_behavior
3    @   requires
4    @     nodeList.length + (\bigint)1 <= Integer.MAX_VALUE;
5    @   ensures
6    @     nodeList == \seq_concat(\old(nodeList),
7    @       \seq_singleton(nodeList[nodeList.length-1])) &&
8    @     ((Node)nodeList[nodeList.length-1]).item == e;
9    @*/
10 void linkLast(/*@ nullable @*/ Object e) {
11     final Node l = last;
12     final Node newNode = new Node(l, e, null);
13     last = newNode;
14     if (l == null) first = newNode;
15     else l.next = newNode;
16     size++;
17     modCount++;
18     //@ set nodeList = \seq_concat(nodeList,\seq_singleton(last));
19 }
```

Listing 4.5: linkLast(Object e) (annotated with JML)

4.2.4 clear()

Listing 4.11 shows method `clear()`. It removes all of the elements from the list. In addition, for each node, its fields `prev`, `item`, and `next` are cleared. Table 4.8 shows comments on the contract.

4.3 Verification

We start by giving a general strategy we apply to verify proof obligations. We also describe in more detail how to produce a proof, for a few contracts.

4.3.1 Sequent Calculus

The theorem prover used by KeY is using sequent calculus. A sequent takes the form $\varphi_1, \ldots, \varphi_m \Longrightarrow \psi_1, \ldots, \psi_n$, where $\Longrightarrow$ is the sequent operator, $\varphi_1, \ldots, \varphi_m$ is the set of formulas called the antecedents, and $\psi_1, \ldots, \psi_n$ is the set of formulas called the

TABLE 4.2: Description of JML of add(Object e)

Line	Description
2–17	JML specification, recognizable as special comments in the Java code, starting with `/*@` or `//@`. It exists of two specification cases. In KeY each specification case is considered a contract. This method implements the equally named method of the `Collection` interface. The `also` keyword (line number 3) expresses the fact that the specification (if any) of the latter is inherited.
5–11	Specification case for normal behavior. The casting with the `\bigint` type is to enforce a *mathematical* add operation, instead of Java's version. Compare line numbers 8–10 to expression 4.3. Notice the use of the JML keyword `\old`. An expression `\old(`x`)` refers to the value of x in the prestate of a method. The `ensures` clause is the only place in JML where `\old` can be used. `\seq_singleton(nodeList[`i`])` is needed when a single element of `nodeList` is used as an argument of the `\seq_concat` operation. Note the cast to type `Node` of the element on line 10. The JML keyword `\result` is used for the return value of a method. The shorthand `\result` for `\result == true` suffices.
13–16	Specification case for exceptional behavior.
18	The expression `/*@ nullable @*/` overrides the nonnull by default behavior of JML.
19	Newly added method to avoid overflow.

TABLE 4.3: Description of JML of linkLast(Object e)

Line	Description
3–8	Specification case for normal behavior is identical with that of add(Object e), except the latter has non-void return type.
18	Updating `nodeList`. The `set` command has been placed on the last line, but every place after line 13 is also fine.

succedents. φ_i and ψ_j are assumed to be ground formulas. m and n are non-negative integers, i.e., the left-hand-side or the right-hand-side (or neither or both) may be empty. The sequent $\varphi_1, \ldots, \varphi_m \Longrightarrow \psi_1, \ldots, \psi_n$ is valid iff the formula $\bigwedge_{1=i}^{n} \varphi_i \rightarrow \bigvee_{1=j}^{m} \psi_j$ is valid [11]. Open goals in KeY take the form of a sequent, and a goal can be closed iff the sequent is valid.

4.3.2 Proof Obligations

Listing 4.12 shows the sequent in the initial proof obligation for the normal behavior contract of `add(Object)`. The part between `\<...\>` refers to modalities, i.e., program fragments. By symbolic execution KeY transforms the statements in the Java program into formulas.[73] `self.<inv>` refers to the invariant (see section 4.1). An open goal is to be considered as a proof obligation to be discharged. Listing 4.12 is the first open goal that shows up when the contract of the method has been loaded in KeY. One could see the initial proof obligation as the top of an upside-down tree, hence the name proof tree. Let's call a proof obligation an open leaf of the tree, belonging to a

[73] To avoid aliasing, Java assignments cannot be symbolically executed by syntactic substitution, but instead Java DL uses (state) updates.

```
1  // implements java.util.List.remove
2  /*@
3    @ also
4    @ public normal_behavior
5    @   requires
6    @     0 <= index < nodeList.length;
7    @   ensures
8    @     nodeList == \old(\seq_concat(nodeList[0..index],
9    @       nodeList[index+1..nodeList.length])) &&
10   @     \result == \old(((Node)nodeList[index]).item);
11   @ public exceptional_behavior
12   @   requires
13   @     index < 0 || index >= nodeList.length;
14   @   signals_only IndexOutOfBoundsException;
15   @   signals (IndexOutOfBoundsException e) true;
16   @*/
17 public /*@ nullable @*/ Object remove(int index) {
18     checkElementIndex(index);
19     //@ set nodeIndex = index;
20     return unlink(node(index));
21 }
```

Listing 4.6: remove(int index) (annotated with JML)

Table 4.4: Description of JML of remove(int index)

Line	Description
6	If $0 \leq$ `index` $<$ `nodeList.length` holds, this will prevent the call of `checkElementIndex(index)` from throwing an exception. It serves as an assertion for the call of `node(index)`.
8–10	The `ensures` clause states that the list is the same at it was at the prestate, except that the node that is associated with position `index` is no longer present. `nodeList`$[i \,..\, j]$: the nodes in `nodeList` starting from index i (including) to index j (excluding). (Note that `nodeList`$[i \,..\, i]$ `== \seq_empty`.) The return value is the item of the node that is no longer present.
13–14	If the opposite of $0 \leq$ `index` $<$ `nodeList.length` holds, this will trigger the call of `checkElementIndex(index)` of throwing an exception of type `IndexOutOfBoundsException`.
17	The return value of the method might be `null`.
19	Use the `set` command to 'pass on' the correct value of `nodeIndex` to `unlink()`.

branch.[74] The strategy is to discharge a leaf, i.e., close it (starting with the top leaf) by applying rules to it such that new leafs are easier to close. Some rules split the branch to which a leaf belongs into multiple branches with each a new leaf, i.e., the open leaf is split into multiple, each belonging to an own branch. Applying a rule to a leaf, closes it and generates one or more (branching) new open leaf(s). A branch is closed whenever there are no (more) open leafs in the branch. The proof is discharged (i.e., the proof tree is closed) when the top leaf has no open branches. A rule can be applied interactive or automatically. The latter usually means that the user let KeY apply a macro which let KeY generate a series of rules to be applied without any interaction. When trying to prove a contract in KeY, one could say: assuming the current branches to be closeable, the conclusion is that the whole tree is closeable.

[74] In KeY a leaf (whether or not open) is called a node. We will refer to it as a leaf to stress the similarity with a tree.

```
/*@
  @ private exceptional_behavior
  @   requires
  @     index < 0 || index >= nodeList.length;
  @   signals_only IndexOutOfBoundsException;
  @   signals (IndexOutOfBoundsException e) true;
  @ private normal_behavior
  @   requires
  @     0 <= index < nodeList.length;
  @   ensures
  @     true;
  @*/
private /*@ strictly_pure @*/ void
checkElementIndex(int index) {
    if (!isElementIndex(index))
        throw new IndexOutOfBoundsException(
            outOfBoundsMsg(index));
}
/*@
  @ private normal_behavior
  @   ensures
  @     \result <==> 0 <= index < nodeList.length;
  @*/
private /*@ strictly_pure @*/ boolean
isElementIndex(int index) {
    return index >= 0 && index < size;
}
```

Listing 4.7: checkElementIndex(int index) (annotated with JML)

Java DL vocabulary To show a glimpse of the Java DL machinery, we present two definitions (first definition taken from [26]; second one from [27]) where parts of them having a direct relation to some parts of proofs we will show in section 4.3.4. The signature of Java DL is a collection of the symbols that can be used to construct formulas.

Definition 4.1 (Signature)**.** *A signature* Σ *is a tuple* $\langle\langle \mathcal{T}, \preceq\rangle, \mathcal{P}, \mathcal{F}, \mathcal{PV}, \mathcal{V}\rangle$ *consisting of a set of types* $\mathcal{T}$ *together with a type hierarchy* $\preceq$ (see Figure 4.1)*, predicates* $\mathcal{P}$*, functions* $\mathcal{F}$*, program variables* $\mathcal{PV}$ *and logical variables* $\mathcal{V}$*.*

Definition 4.2 (Syntax)**.** *Terms* t*, formulas* φ*, updates* $\mathcal{U}$ *and programs* $\mathtt{p}$ *are defined by the following grammar, where* $f \in \mathcal{F}$ *ranges over function symbols,* $p \in \mathcal{P}$ *over predicate symbols,* $\mathtt{x} \in \mathcal{PV}$ *over program variables, and* $y \in \mathcal{V}$ *over logical variables:*

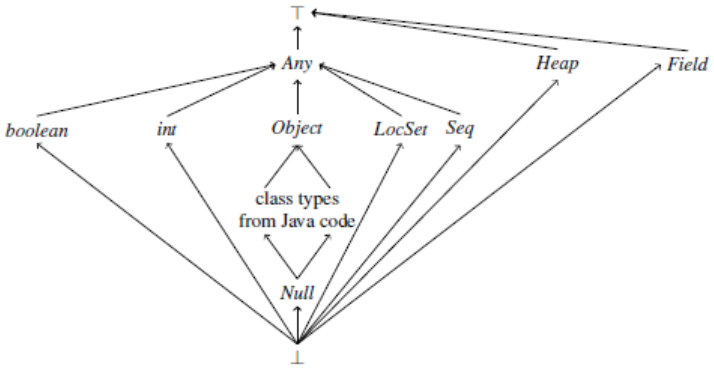

Figure 4.1: The type hierarchy $\preceq$ of types $\mathcal{T}$ of Java DL (taken from [11]); $\top$ (the universal type) being the top element and the other types ordered directly below $\top$. $\bot$ is the empty type.

```
/*@
  @ normal_behavior
  @   requires
  @     0 <= index < nodeList.length;
  @   ensures
  @     \result == nodeList[index];
  @*/
/*@ strictly_pure @*/ Node node(int index) {
    if (index < (size >> 1)) {
        Node x = first;
        /*@
          @ maintaining
          @   x != null && 0 <= i && i <= index &&
          @   x == (Node)nodeList[i];
          @ decreasing
          @   index - i;
          @ assignable
          @   \strictly_nothing;
          @*/
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node x = last;
        /*@
          @ maintaining
          @   x != null && index <= i && i <= size - 1 &&
          @   x == (Node)nodeList[i];
          @ decreasing
          @   i - index;
          @ assignable
          @   \strictly_nothing;
          @*/
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

Listing 4.8: node(int index) (annotated with JML)

4.3.3 Overview of verification steps

When verifying a method, KeY first has to perform symbolic execution. Symbolic execution transforms modal operators on program fragments into Java DL. We have KeY do this by applying a macro. When doing this, simplification rules are also applied automatically. We keep in mind that the class invariant contains a disjunction, and in case we do not want this disjunction to be split during the execution of the macro, KeY has to be instructed to delay unfolding the invariant.[75] When symbolic execution is finished, goals may contain heap expressions that must be simplified. When this is done for all heap expressions, the open goals look as simple as they can be before simplifying them further. This in general may be a good moment to compare the open goals to the method and its annotations, and see whether things in KeY look familiar at this point. In the remaining part of the proof the user must find an appropriate mix between interactive and automatic steps.

There are many ways to construct a closed proof tree. At (almost) every step the user has a choice between applying steps manually or automatically. When applying a manual step, different choices can be made which rule in what order to apply where: it takes some experience to choose the best rule. An important rule is the cut rule that splits a proof tree into two parts. A cleverly chosen instantiation of a cut rule significantly reduces the size of a proof and the effort required to produce it. Even

[75] The disjunction refers to the distinction between an empty and a non-empty list.

```
/*@
  @ 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) {
    //@ set nodeList = \seq_concat(\dl_seqSub(nodeList,0,nodeIndex),
    ↪  \dl_seqSub(nodeList,nodeIndex+1,\dl_seqLen(nodeList)));
    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--;
    modCount++;
    return element;
}
```

Listing 4.9: unlink(Node x) (annotated with JML)

more so, without a cut rule it might be impossible to produce a closed proof tree in the first place. For example, the latter is the case for methods where the acylicity property is needed for the proof.

In the example proofs shown we might refer to relevant snippets in a sequent. We will only refer to snippets in sequents where modalities have been accounted for completely, i.e., where symbolic execution and simplification have been fully executed. Program variables $p \in \mathcal{PV}$ are shown in `typewriter` font where as logical variables $y \in \mathcal{V}$ will be shown in *italic* font. The variable `self` refers to the receiver object and as such is considered as a program variable. The ghost field *nodeList* is considered as a logical variable. KeY normally will take as a name for a logical variable something that more or less resembles the name of the program variable where it comes from, e.g., `index` may become *index*_1_0. Applying `allRight` will create a so called Skolem constant, e.g., when the name of the loop variable is j, the Skolem constant might get a name like j_2.

4.3.4 Proofs (Selection)

For methods `lastIndexOf(Object)`, `linkFirst(Object)`, and `addFirst(Object)` we show in some detail how a closed proof tree can be obtained. This gives a general feel how proving in KeY works. These methods are not complicated to verify. In this manner, we have produced proofs for each method contract that we have specified. Section 4.4 shows overall proof statistics. Note: these three proofs have been re-established with the purpose of describing them. With that in mind we thought it would be useful to minimize the number of interactive steps for these proofs. See Table 4.9 for altered statistics. Minimizing interaction wasn't something we were

TABLE 4.5: Description of JML of node(int index)

Line	Description
2–6	If $0 \leq$ `index` $<$ `nodeList.length` holds, the method will return the node associated with the given index.
12–18	This construct is a loop-invariant, i.e., a loop specification. KeY needs it in order to reason about unbounded loops. The invariant must hold at the start and the end of the loop, and for every iteration. The arguments of the invariant are given in the `maintaining` clause. `x != null` is true since we know that `first != null` because `nodeList.lenght` > 0 (precondition), (and thus: `size` > 0). `0 <= i && i <= index` follows from the for loop at line 20. `x == (Node)nodeList[i]` is needed since it determines the value that the method returns, viz., when `i == index`. Note that if `index == 0`, the loop invariant still holds before and after the loop, except that there will be no iteration. The `decreasing` clause is the termination witness. The `assignable` clause determines the frame of the loop. There are no locations this loop writes to, thus we use `\strictly_nothing`.
26–32	Similar to lines 12–18, except for the opposite direction.

generally aiming for. For `linkFirst(Object)` it will be shown that the acyclicity property (see section 2.3) is needed for the verification.

4.3.4.1 lastIndexOf(Object)

Listing 4.13 shows method `lastIndexOf(Object)`: it searches through the chain of nodes until it finds a node with an item equal to the argument. This method is interesting due to a potential overflow of the resulting index. The method has two contracts: one involves a `null` argument, and another involves a non-`null` argument. Both proofs are similar. Moreover, the proof for `indexOf(...)` is similar but involves the `next` reference instead of the `prev` reference. This contract is interesting, since proving its correctness shows the absence of the overflow bug.

Proof. Choose *Defaults* as predefined strategy, then set *Max. Rule Applications* to 1,000, and set *Class axiom rule* to *Delayed*. Apply macro **Finish Symbolic Execution**[76] to the sequent. We now have six open goals. Set *Arithmic treatment* to *DefOps*. Apply macro **Close provable goals below**[77] to the root of the proof tree. One open goal remains. Apply macro **Update Simplification Only**[78] to the sequent. One open goal remains where modalities have disappeared. Apply macro **Propositional expansion (w/ splits)**[79] to the conjunct on the right side of the sequent. We have seven open goals. Apply macro **Close provable goals below** to the root of the proof tree. One open goal left. The succedent of interest is: $subJint(index\_1\_0, 1) = 0 \leftrightarrow x\_0.\texttt{prev} = null$. *subJint* is a predefined arithmic function symbol in Java DL that represents Java's modulo semantics when subtracting two variables of type `int`. Given the boundaries of $index\_1\_0$ we can rewrite this to $index\_1\_0 - 1$, so that we are left with one open goal where the succedent that must proven to be true is: $index\_1\_0 - 1 = 0 \leftrightarrow x\_0.\texttt{prev} = null$. Split the equivalence using rule `equiv_right`, i.e., the sequent is split in two.

[76] Strategic macros → Auto Pilot → Finish Symbolic Execution.
[77] Strategic macros → Close provable goals below.
[78] Strategic macros → Simplification → Update Simplification Only.
[79] Strategic macros → Propositional → Propositional expansion (w/ splits).

Table 4.6: Description of JML of unlink(Node x)

Line	Description
4	This conjunct is redundant as it follows from line 5. It is left in the precondition so that KeY does not have to infer it.
5	`nodeIndex` plays the same role as `index` does for `remove(int index)`.
6	The argument for parameter `x` of `unlink(Node x)` must refer to the same node as `(Node)nodeList[nodeIndex]` refers to.
8–10	Similar to lines 8–10 of `remove(int index)`.
11	`nodeIndex` stays unaltered.
13	The return value of the method might be `null`.
14	Updating `nodeList`. The `set` command has been placed on the first line, but every other place is fine also. Note the multiple use of the escape sequence `\dl_`. This is followed immediately by a function name, e.g., `seqSub`. We need this when we have a property that cannot be expressed in JML, but can be represented on the level of the logic of KeY it self, viz., Java DL. Thus: KeY introduces escapes from JML into Java DL.

- $index\_1\_0 - 1 = 0, \ldots$ `==>` $\ldots, x\_0.$`prev` $= null$: Set Class axiom rule to *Free*. Applying the macro **Close provable goals below** to this sequent will close it. What will happen is this: KeY unfolds the invariant by which formulas `self.first.prev` $= null$ and `self.first` $=$ `self.`$nodeList[0]$ appear. Furthermore, KeY will rewrite `self.`$nodeList[subJint(index\_1\_0, 1)] = x\_0$ to `self.`$nodeList[index\_1\_0 - 1] = x\_0$. This is sufficient to have KeY infer the antecedent $x\_0.$`prev` $= null$. And since this formula is already a succedent, the goal can be closed.

- $x\_0.$`prev` $= null, \ldots$ `==>` $\ldots, index\_1\_0 - 1 = 0$: Apply a cut rule to the sequent and instantiate it with the formula $index\_1\_0 > 1$, i.e., the sequent is split in two.

  - $index\_1\_0 > 1, x\_0.$`prev` $= null, \ldots$ `==>` $\ldots, index\_1\_0 - 1 = 0$: the strategy is to create a contradiction w.r.t. $x\_0.$`prev` $= null$. Unpack the invariant. This creates branches *Use Axiom* and *Show Axiom Satisfiability*, of which the latter can be closed automatically. Use Axiom: Apply macro **Propositional expansion (w/ splits)** to the conjuncts that have appeared by unfolding the invariant. Two branches, where the one referring to an empty list can be closed automatically. Non-empty list: let us use $\mathcal{N}$ as an abbreviation for `java.util.BoundedLinkedList.Node`. Apply rule `allLeft` to

    $$\forall int\ i\_0;\ (0 < i\_0 \wedge i\_0 < \texttt{self.}nodeList.length \rightarrow (\mathcal{N})(\texttt{self.}nodeList[i\_0]).\texttt{prev} = (\mathcal{N})(\texttt{self.}nodeList[i\_0 - 1]))$$

    where i_0 is instantiated with $index\_1\_0 - 1$. This yields the new antecedent

```
1  // implements java.util.List.set
2  /*@
3    @ also
4    @ public normal_behavior
5    @   requires
6    @     0 <= index < nodeList.length;
7    @   ensures
8    @     ((Node)nodeList[index]).item == element &&
9    @     nodeList == \old(nodeList) &&
10   @     \result == \old(((Node)nodeList[index]).item);
11   @ public exceptional_behavior
12   @   requires
13   @     index < 0 || index >= nodeList.length;
14   @   signals_only IndexOutOfBoundsException;
15   @   signals (IndexOutOfBoundsException e) true;
16   @*/
17 public /*@ nullable @*/ Object
18 set(int index, /*@ nullable @*/ Object element) {
19     checkElementIndex(index);
20     Node x = node(index);
21     Object oldVal = x.item;
22     x.item = element;
23     return oldVal;
24 }
```

LISTING 4.10: set(int index, Object element) (annotated with JML)

$$0 < index\_1\_0 - 1 \land index\_1\_0 - 1 < \mathsf{self}.nodeList.length \rightarrow \\ (\mathcal{N})(\mathsf{self}.nodeList[index\_1\_0 - 1]).\mathsf{prev} = \\ (\mathcal{N})(\mathsf{self}.nodeList[index\_1\_0 - 1 - 1]).$$

Apply the rule impLeft to this new formula. This gives two branches, where the one with

$$0 < index\_1\_0 - 1 \land index\_1\_0 - 1 < \mathsf{self}.nodeList.length$$

as a succedent can be closed automatically. In the open goal we have the new antecedent

$$(\mathcal{N})(\mathsf{self}.nodeList[index\_1\_0 - 1]).\mathsf{prev} = \\ (\mathcal{N})(\mathsf{self}.nodeList[index\_1\_0 - 1 - 1]).$$

Apply rule allLeft to

$$\forall\ int\ i\_1; \\ (0 \leq i\_1 \land i\_1 < \mathsf{self}.nodeList.length \\ \rightarrow \neg(\mathsf{self}.nodeList[i\_1]) = null \land \\ \mathcal{N}::instance(\mathsf{self}.nodeList[i\_1]) = \text{true})$$

where i is instantiated with $index\_1\_0 - 2$. This yields the antecedent

$$0 \leq index\_1\_0 - 2 \land index\_1\_0 - 2 < \mathsf{self}.nodeList.length \\ \rightarrow \neg(\mathsf{self}.nodeList[index\_1\_0 - 2]) = null \land \\ \mathcal{N}::instance(\mathsf{self}.nodeList[index\_1\_0 - 2]) = \text{true}.$$

Apply the rule impLeft to this formula. This gives two branches, where the

TABLE 4.7: Description of JML of set(int index, Object element)

Line	Description
6	If $0 \leq$ `index` $<$ `nodeList.length` holds, this will prevent the call of `checkElementIndex(index)` from throwing an exception. It serves as an assertion for the call of `node(index)`.
8	The argument of the parameter `element` of this method, serves as the new item of the node at the position determined by the specified index.
9	`nodeList` stays unaffected! This is since at the level of the nodes nothing changes.
10	The return value is the item of the node as it was at the prestate of the method.
13–14	If the opposite of $0 \leq$ `index` $<$ `nodeList.length` holds, this will trigger the call of `checkElementIndex(index)` of throwing an exception of type `IndexOutOfBoundsException`.
17	The return value of the method might be `null`.
18	The argument for parameter `element` might be `null`.
¬	No `set` command (see comment w.r.t. line 9).

one with

$$0 \leq index\_1\_0 - 2 \wedge index\_1\_0 - 2 < \texttt{self}.nodeList.length$$

as a succedent can be closed automatically. In the open goal we have the new antecedent

$\neg(\texttt{self}.nodeList[index\_1\_0 - 2]) = null \wedge$
$\mathcal{N}::instance(\texttt{self}.nodeList[index\_1\_0 - 2]) = \text{true}.$

Apply the rule `andLeft` to this formula. As relevant antecedents, we have

$x\_0.\texttt{prev} = null,$
$(\mathcal{N})\texttt{self}.nodeList[index\_1\_0 - 1] = x\_0,$
$(\mathcal{N})(\texttt{self}.nodeList[index\_1\_0 - 1]).\texttt{prev} =$
$(\mathcal{N})(\texttt{self}.nodeList[index\_1\_0 - 1 - 1]),$ and
$\neg\texttt{self}.nodeList[index\_1\_0 - 1 - 1] = null.$

In other words, we have constructed the contradiction. After a few rewrite rules KeY can close the goal automatically.

– $x\_0.\texttt{prev} = null, \ldots$ `==>` $index\_1\_0 > 1, \ldots, index\_1\_0 - 1 = 0$: the sequent can be closed automatically, since we have as an antecedent $index\_1\_0 \geq 1$ (not shown here).

□

4.3.4.2 linkFirst(Object)

This method is very similar to linkLast(Object) (see Listing 4.5), except that here the specified element is not linked as last element, but instead as first element. This proof also shows the absence of the overflow bug.

Table 4.8: Description of JML of clear()

Line	Description
6–10	The postcondition states that the list is empty. Furthermore, it states that for each node that was in the list, the fields `prev`, `item`, and `next` have been cleared.
13–14	Declaration and initialization of ghost variable `index` of JML-type `\bigint`.
16–44	Loop specification, consisting of 8 invariants and one termination witness.
17	Specifying the range of `index`. Note that the upperbound equals the number the number of nodes in the list, i.e., `nodeList.length`.
19	When the last node is reached, the outcome of the statement `x = next;` on line 52 will be that `x` becomes `null`. At that moment `index == nodeList.length`, as a result of the `set` statement on line 51. Thus: `x != null` implies that `index < nodeList.length`.
21	Vice versa, when the last node hasn't been reached (`index < nodeList.length`), we know that `x` refers to `nodeList[index]`.
23–24	For the nodes whose fields haven't been cleared (yet), we know that they are connected.
26–27	For a non-empty list, we know that the next field of the last node is `null`.
29	`nodeList` is unaffected. Fields of nodes are cleared, but the nodes themselves are not. They are still sitting in `nodeList`.
31–34	The fields of `nodeList[0 .. index]` (`[`x `..` `y]`: x inclusive and y exclusive) are cleared.
36–42	The fields of `nodeList[index .. nodeList.length]` are unaffected.
51	Have `index` incremented with 1 by using the `set` command.
56	Empty `nodeList` by using the `set` command.

$$
\begin{array}{lcl}
t & ::= & f(t,\ldots,t) \mid x \mid y \mid if(\varphi) then(t) else(t) \mid \{\mathcal{U}\}t \\
\varphi & ::= & \text{true} \mid \text{false} \mid p(t,\ldots,t) \mid \varphi \;\&\; \varphi \mid (\varphi \mid \varphi) \mid \varphi \texttt{ -> } \varphi \mid \texttt{!}\,\varphi \mid \\
 & & \forall y.\varphi \mid \exists y.\varphi \mid t \doteq t \mid \{\mathcal{U}\}\varphi \mid [\texttt{p}]\varphi \\
\mathcal{U} & ::= & (\texttt{x} := t \parallel \ldots \parallel \texttt{x} := t) \\
\texttt{p} & ::= & \texttt{x = } t \mid \texttt{p;p} \mid \texttt{if (}\varphi\texttt{) \{p\} else \{p\}} \mid \texttt{while (}\varphi\texttt{) \{p\}}
\end{array}
$$

Proof. Choose Defaults as predefined strategy, then set Max. Rule Applications to 1,000, and set Class axiom rule to Delayed. Apply macro **Finish Symbolic Execution** to the sequent. We now have two open goals. Set Max. Rule Applications to 2,000, Class axiom rule to Free, and Arithmic treatment to DefOps. Apply macro **Close provable goals below** to the root of the proof tree. The goal with `self.first` = *null* as antecedent has been closed. Unpack the invariant. This creates branches Use Axiom and Show Axiom Satisfiability, of which the latter can be closed automatically. Use Axiom: Apply macro **Propositional expansion (w/ splits)** to the conjuncts that have appeared by unfolding the invariant. Two branches, where the one referring to an empty list can be closed automatically. Non-empty list: Apply macro **Update Simplification Only** to the sequent. One open goal remains where modalities have disappeared. Apply macro **Propositional expansion (w/ splits)** to the conjunct on the right side of the sequent. We have now four open goals. Apply macro **Close provable goals below** to the root of the proof tree. One open goal remains. Unpack the invariant (on the right side of the sequent). This creates

Method	i/r	i/r r-e
`lastIndexOf(Object)`	23/4,571	18/4,880
`linkFirst(Object)`	665/33,650	24/13,974
`addFirst(Object)`	31/1,863	0/612

TABLE 4.9: Re-established proof statistics (three proofs). i/r = Iterative steps versus total number of rules; r-e = re-established.

branches Use Axiom and Show Axiom Satisfiability, of which the latter can be closed automatically. Use Axiom: Apply macro **Propositional expansion (w/ splits)** to the conjuncts that have appeared by unfolding the invariant. We have now 23 open goals. Apply macro **Close provable goals below** to the root of the proof tree. One open goal remains. Apply macro **Heap Simplification**[80] to the sequent. Set Max. Rule Applications to 60. Left-click on the sequent arrow and select the context menu entry **Apply rules automatically here**. As a succedent ρ of interest, we have $\rho =$

$$\begin{aligned}
&\backslash if\ (\texttt{self.first} = (\mathcal{N})(seqConcat(seqSingleton(n_2), \texttt{self}.nodeList)[i_0_3])) \\
&\backslash then\ (n_2) \\
&\backslash else\ (\backslash if\ (n_2 = (\mathcal{N})(seqConcat(seqSingleton(n_2), \texttt{self}.nodeList)[i_0_3])) \\
&\qquad \backslash then\ (null) \\
&\qquad \backslash else\ ((\mathcal{N})(seqConcat(seqSingleton(n_2), \texttt{self}.nodeList)[i_0_3]).\texttt{prev})) \\
&= (\mathcal{N})(seqConcat(seqSingleton(n_2), \texttt{self}.nodeList)[i_0_3 - 1])
\end{aligned}$$

where $\mathcal{N}$ serves as an abbreviation for `java.util.BoundedLinkedList.Node` to save space. Logical variable i_0_3 is bounded by the antecedents α and β:

$$\begin{aligned}
&0 < i_0_3, && (\alpha) \\
&i_0_3 < seqConcat(seqSingleton(n_2), \texttt{self}.nodeList).length && (\beta)
\end{aligned}$$

The logical variable n_2 reflects the new node, where as

$$seqConcat(seqSingleton(n\_2), \texttt{self}.nodeList)$$

represents the list in the poststate of the method. Let's abbreviate the latter with L_p. We could then rewrite $\rho =$

$$\begin{aligned}
&\backslash if\ (\texttt{self.first} = (\mathcal{N})(L_p[i_0_3])) \\
&\backslash then\ (n_2) \\
&\backslash else\ (\backslash if\ (n_2 = (\mathcal{N})(L_p[i_0_3])) \\
&\qquad \backslash then\ (null) \\
&\qquad \backslash else\ ((\mathcal{N})(L_p[i_0_3]).\texttt{prev})) \\
&= (\mathcal{N})(L_p[i_0_3 - 1]).
\end{aligned}$$

We need to prove that ρ follows from the left side of the sequent. In order to do so, we need to 'inject' the *acyclicity* property of the list (see Lemma 2.1) as an antecedent. We will use a Java DL formula that reflects this property. Suppose we denote this formula with τ, then we need to have this:

$$\tau,\ \alpha,\ \beta,\ \ldots\ \texttt{==>}\ \ldots,\ \rho$$

[80] Strategic macros → Simplification → Heap Simplification.

However, in order to do so, we first must make sure that τ may be used as an antecedent in the first place. In other words, we need to prove that τ follows from the left side of the sequent, viz., $\ldots$ `==>` $\ldots, \tau$. (α and β play no role here.) We use a cut rule instantiated with τ, which generates two branches in KeY (the order is as KeY shows them):

τ: $\tau, \alpha, \beta, \ldots$ `==>` $\ldots, \rho$

$\neg\tau$: $\ldots$ `==>` $\ldots, \tau$

Our goal is to first discharge the proof obligation for branch $\neg\tau$. In this branch we want to proof that τ *follows* from the left side of `==>`. If and only if we succeed in doing that, it makes sense to use τ as an antecedent. In the rest of the proof we show discharging proof obligations in chronological order. We instantiate a cut rule on the sequent of the open goal with $\tau =$

$$\begin{array}{l}\forall\ int\ j;\\ \quad \forall\ int\ i;\\ \quad\quad (0 \leq i \land i < j \land j < \texttt{self}.nodeList.length\\ \quad\quad\ \rightarrow \neg(\mathcal{N})(\texttt{self}.nodeList[i]) = (\mathcal{N})(\texttt{self}.nodeList[j])).\end{array}$$

Compare with property 2.5. We are using Unicode symbols (e.g., $\forall$ instead of $\backslash forall$) to stay more close to a mathematical way of presenting formulas. This is a *view*, which can be enabled in KeY as a View Setting. Note that this is *not* what can be used for instantiation since KeY will not parse it. Literally we have to use this:[81]

$$\begin{array}{l}\backslash forall\ int\ j;\\ \quad \backslash forall\ int\ i;\\ \quad\quad (0 <= i\ \&\ i < j\ \&\ j < \texttt{self}.nodeList.length\\ \quad\quad\ \texttt{->}\ \texttt{!}(\mathcal{N})(\texttt{self}.nodeList[i]) = (\mathcal{N})(\texttt{self}.nodeList[j]))\end{array}$$

Applying the cut rule thus creates two branches, and we will discharge them in the opposite order as shown by Key.

$\neg\tau$: $\ldots$ `==>` $\ldots, \tau$: Apply allRight to τ and then once again to the result of it. After that the result is an implication with two new logical variables i_3 and j_0.[82] Apply impRight and then apply macro **Propositional expansion (w/ splits)** to the new variables (which have been moved to the left side of the sequent). To close this branch we need a proof *by contradiction*. We use a cut rule where we use as instantiation $\upsilon =$

$$\begin{array}{l}\forall\ int\ k;\\ \quad (j\_0 \leq k \land k < \texttt{self}.nodeList.length\\ \quad\ \rightarrow (\mathcal{N})(\texttt{self}.nodeList[k]) = (\mathcal{N})(\texttt{self}.nodeList[k - (j\_0 - i\_3)])).\end{array}$$

Compare with property 2.7. We will proof that υ cannot be true, and by that τ must be true.

$\neg\upsilon$: $\ldots$ `==>` $\ldots, \upsilon$: Apply auto_induction to υ. Set Max. Rule Applications to 2,000. Apply macro **Close provable goals below** to the node in the

[81] Still keep in mind that $\mathcal{N}$ is used as an abbreviation for `java.util.BoundedLinkedList.Node`.

[82] The exact names of the variables may vary, e.g., it is affected by the number of times a proof has been pruned.

proof tree rule where auto_induction was applied. Of the three branches that were created by auto_induction, Step Case (positive) is still open. Apply impRight to the succedent that refers to a single step of v and then once again to the result of it. Consecutively apply macro **Propositional expansion (w/ splits)** to the first and second antecedent. We now have three open goals. Apply macro **Close provable goals below** to branch Step Case (positive). Now (again) one goal open. Apply a cut rule to this goal and instantiate it with the formula $k\_0 + 1 = j\_0$. The branch where this formula is an antecedent can be closed automatically. The goal where it is a succedent stays open. Rewrite antecedent $j\_0 \leq k\_0 + 1$ to $j\_0 < k\_0 + 1 \vee j\_0 = k\_0 + 1$, and apply orLeft to the disjunct. This splits the goal in two, where the one with $j\_0 = k\_0 + 1$ can be closed automatically. The one with $j\_0 < k\_0 + 1$ stays open. Now we have

$$(\mathcal{N})(\texttt{self}.nodeList[k\_0]) = (\mathcal{N})(\texttt{self}.nodeList[k\_0 - (j\_0 - i\_3)])$$

as an antecedent, and

$$(\mathcal{N})(self.nodeList[k\_0 + 1]) =$$
$$(\mathcal{N})(self.nodeList[k\_0 + 1 - (j\_0 - i\_3)])$$

as a succedent. Apply allLeft to

$$\forall\ int\ i;$$
$$(0 \leq i \wedge i < javaSubInt(\texttt{self}.nodeList.length, 1)$$
$$\rightarrow (\mathcal{N})(\texttt{self}.nodeList[i + 1]) = (\mathcal{N})(\texttt{self}.nodeList[i]).\texttt{next})$$

where i is instantiated with $k\_0 - (j\_0 - i\_3)$.[83] This gives rise to the new antecedent

$$0 \leq k\_0 - (j\_0 - i\_3) \wedge$$
$$k\_0 - (j\_0 - i\_3) < javaSubInt(\texttt{self}.nodeList.length, 1)$$
$$\rightarrow (\mathcal{N})(\texttt{self}.nodeList[k\_0 - (j\_0 - i\_3) + 1]) =$$
$$(\mathcal{N})(\texttt{self}.nodeList[k\_0 - (j\_0 - i\_3)]).\texttt{next}.$$

Apply the rule impLeft to this new formula. This splits the sequent in two where the one with

$$0 \leq k\_0 - (j\_0 - i\_3) \wedge$$
$$k\_0 - (j\_0 - i\_3) < javaSubInt(\texttt{self}.nodeList.length, 1)$$

as a succedent can be closed automatically. The open goal contains the new antecedent

$$(\mathcal{N})(\texttt{self}.nodeList[k\_0 - (j\_0 - i\_3) + 1]) =$$
$$(\mathcal{N})(\texttt{self}.nodeList[k\_0 - (j\_0 - i\_3)]).\texttt{next}.$$

This goal can be closed automatically by first cut it with

[83] $javaSubInt$ is KeY's translation of the Java integer expression where two integer variables are subtracted.

$k\_0 - (j\_0 - i\_3) + 1 = k\_0 + 1 - (j\_0 - i\_3)$

as instantiation. No more open goals here.

v: v, ... ==> ...: Apply allLeft to v with $\texttt{self}.nodeList.length - 1$ as instantiaton for k. This creates the new antecedent

$$\begin{aligned} & j\_0 \leq \texttt{self}.nodeList.length - 1 \wedge \\ & \quad \texttt{self}.nodeList.length - 1 < \texttt{self}.nodeList.length \\ & \rightarrow (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1]) = \\ & \quad (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3)]). \end{aligned}$$

Apply the rule impLeft to this new formula. This splits the sequent in two where the one with

$$\begin{aligned} & j\_0 \leq \texttt{self}.nodeList.length - 1 \wedge \\ & \quad \texttt{self}.nodeList.length - 1 < \texttt{self}.nodeList.length \end{aligned}$$

as a succedent can be closed automatically. The open goal contains the new antecedent

$$\begin{aligned} & (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1]) = \\ & (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3)]). \end{aligned}$$

Apply allLeft to

$$\begin{aligned} & \forall\ int\ i; \\ & \quad (0 \leq i \wedge i < javaSubInt(\texttt{self}.nodeList.length, 1) \\ & \quad \rightarrow (\mathcal{N})(\texttt{self}.nodeList[i + 1]) = (\mathcal{N})(\texttt{self}.nodeList[i]).\texttt{next}) \end{aligned}$$

where i is instantiated with $\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3)$. This gives rise to the new antecedent

$$\begin{aligned} & 0 \leq \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) \wedge \\ & \quad \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) < \\ & \quad javaSubInt(\texttt{self}.nodeList.length, 1) \\ & \rightarrow (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1]) = \\ & \quad (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3)]).\texttt{next}. \end{aligned}$$

Apply the rule impLeft to this new formula. This splits the sequent in two where the one with

$$\begin{aligned} & 0 \leq \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) \wedge \\ & \quad \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) < \\ & \quad javaSubInt(\texttt{self}.nodeList.length, 1) \end{aligned}$$

as a succedent can be closed automatically. The open goal contains the new antecedent

$$\begin{aligned} & (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1]) = \\ & (\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3)]).\texttt{next}. \end{aligned}$$

Apply rule allLeft to

$$\forall\ int\ i\_1;\\ (0 \leq i\_1 \land i\_1 < \texttt{self}.nodeList.length\\ \rightarrow \neg(\texttt{self}.nodeList[i\_1]) = null \land\\ \mathcal{N}::instance(\texttt{self}.nodeList[i\_1]) = \text{true})$$

where i is instantiated with $\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1$. This gives rise to the new antecedent

$$0 \leq \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1 \land\\ \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1 <\\ \texttt{self}.nodeList.length\\ \rightarrow\\ \neg\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1] = null \land\\ \mathcal{N}::instance(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1\\ - (j\_0 - i\_3) + 1]) = \text{true}.$$

Apply the rule impLeft to this new formula. This splits the sequent in two where the one with

$$0 \leq \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1 \land\\ \texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1 <\\ \texttt{self}.nodeList.length$$

as a succedent can be closed automatically. The open goal contains the new antecedent

$$\neg\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1] = null \land\\ \mathcal{N}::instance(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1\\ - (j\_0 - i\_3) + 1]) = \text{true}.$$

Apply andLeft to this new formula. Apply ineffectiveCast to

$$(\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1]).$$

This creates the new antecedent

$$(\mathcal{N})(\texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1]) =\\ \texttt{self}.nodeList[\texttt{self}.nodeList.length - 1 - (j\_0 - i\_3) + 1].$$

KeY is now able to close this goal automatically: There are two nodes with different indexes whose next field $= null$. One of these nodes is the last node, where this is legitimate. The other has an index $j\_0 - i\_3$ smaller than that of the last node. That node cannot have $\texttt{next} = null$, since next must point to a non-null node. KeY signals this contradiction and discharges the proof obligation.[84] So no more open goals in this branch.

τ: τ, ... ==> ..., ρ: KeY can close this branch completely automatic.

[84] In this case it needs 425 rules (counting from the node where ineffectiveCast was applied) to close the goal automatically. Manually this could be done in a few steps, but for this proof description we wanted to keep the number of interactive steps as low as possible.

□

4.3.4.3 addFirst(Object)

In terms of KeY this method has two contracts: one for normal behavior and one for exceptional behavior. We will take normal behavior as an example, where we will make use of the contract of `linkFirst`. Thus: this proof also shows the absence of the overflow bug.

Proof. Choose Defaults as predefined strategy, then set Max. Rule Applications to 300, and set Class axiom rule to Delayed. Apply macro **Finish Symbolic Execution** to the sequent. We now have three goals, one for branch Post (linkFirst), one for branch Pre (linkFirst), and one for branch Pre (checkSize). The first two both fall in branch Post (checkSize). Set Arithmic treatment to DefOps. Set Max. Rule Applications to 100. Apply macro **Close provable goals below** to the root of the proof tree. One open goal remains, for branch Post (linkFirst). Apply macro **Heap Simplification** to the sequent. Left-click on the sequent arrow and select the context menu entry **Apply rules automatically here**. Left-click on last succedent (the one with modalities) and select the context menu entry **Apply rules automatically here**. Two open goals. Apply macro **Close provable goals below** to the root of the proof tree. One open goal remains. Apply macro **Heap Simplification** to the second succedent. Apply macro **Close provable goals below** to the root of the proof tree. That's it. 612 rules were applied and 0 interactive proof steps. Technically the latter is correct. However we had to have KeY guided a little by applying some sort of interference. The main point here is that we used the contract of `linkFirst()`[85] which makes proving `addFirst()` very easy. No need to 'inject' the acyclicity property here whatsoever, since that is already incorporated in the validity of the contract of `linkFirst`. □

4.4 Proof Results

Table 4.10 shows the methods that were in scope of verification. 33 out of the 42 public methods have been verified. All (four) package private methods have been verified. Six out of eleven private methods have been verified. Three methods have been excluded from verification: `addAll(Collection c)` (Nr. 20), `addAll(int index, Collection c)` (Nr. 21), and `toArray(T[] a)` (Nr. 59). Nrs. 20–21 since `Collection` is an interface, which poses a problem for verification (see chapter 5), and Nr. 59 since it uses reflection,[86] which by its nature cannot be verified. Eight methods have been skipped, viz., Nrs. 8, 9, 10, 13, 27, 29, 36, 57. Nr. 8: see chapter 4. Nrs. 9, 13: lack of time. Nrs. 10, 27: We did not create/have a contract for `Cloneable`, i.e., these methods cannot be verified. Nrs. 29, 36, 57: We did not create/have a contract for `Iterable` or for `Iterator`, i.e., these methods cannot be verified. Tables 4.11 and 4.12 show statistics for proofs concerning normal and exceptional behavior, respectively.

Table 4.10: BoundedLinkedList

Nr	Visibility	Method	Status
1	private	checkElementIndex(int index)	verified
2	private	checkPositionIndex(int index)	verified

[85] And also that of `checkSize()` by the way.
[86] See https://www.oracle.com/technical-resources/articles/java/javareflection.html.

Table 4.10 – continued from previous page

Nr	Visibility	Method	Status
3	private	checkSize()	verified
4	private	isElementIndex(int index)	verified
5	private	isMaxSize()	verified
6	private	isPositionIndex(int index)	verified
7	private	linkFirst(Object e)	verified
8	private	outOfBoundsMsg(int index)	skipped
9	private	readObject(java.io.ObjectInputStream s)	skipped
10	private	superClone()	skipped
11	private	unlinkFirst(Node f)	verified
12	private	unlinkLast(Node l)	verified
13	private	writeObject(java.io.ObjectOutputStream s)	skipped
14	package	linkBefore(Object e, Node succ)	verified
15	package	linkLast(Object e)	verified
16	package	node(int index)	verified
17	package	unlink(Node x)	verified
18	public	add(Object e)	verified
19	public	add(int index, Object element)	verified
20	public	addAll(Collection c)	excluded
21	public	addAll(int index, Collection c)	excluded
22	public	addFirst(Object e)	verified
23	public	addLast(Object e)	verified
24	public	BoundedLinkedList()	verified
25	public	BoundedLinkedList(Collection c)	verified
26	public	clear()	verified
27	public	clone()	skipped
28	public	contains(Object o)	verified
29	public	descendingIterator()	skipped
30	public	element()	verified
31	public	get(int index)	verified
32	public	getFirst()	verified
33	public	getLast()	verified
34	public	indexOf(Object o)	verified
35	public	lastIndexOf(Object o)	verified
36	public	listIterator(int index)	skipped
37	public	offer(Object e)	verified
38	public	offerFirst(Object e)	verified
39	public	offerLast(Object e)	verified
40	public	peek()	verified
41	public	peekFirst()	verified
42	public	peekLast()	verified
43	public	poll()	verified
44	public	pollFirst()	verified
45	public	pollLast()	verified
46	public	pop()	verified
47	public	push(Object e)	verified
48	public	remove()	verified
49	public	remove(int index)	verified

Table 4.10 – continued from previous page

Nr	Visibility	Method	Status
50	public	remove(Object o)	verified
51	public	removeFirst()	verified
52	public	removeFirstOccurrence(Object o)	verified
53	public	removeLast()	verified
54	public	removeLastOccurrence(Object o)	verified
55	public	set(int index, Object element)	verified
56	public	size()	verified
57	public	spliterator()	skipped
58	public	toArray()	verified
59	public	toArray(T[] a)	excluded

```
1  // implements java.util.Collection.clear
2  /*@
3    @ also
4    @ public normal_behavior
5    @   ensures
6    @     nodeList == \seq_empty &&
7    @     (\forall \bigint i; 0 <= i < \old(nodeList.length);
8    @       ((Node)\old(nodeList[i])).prev == null &&
9    @       ((Node)\old(nodeList[i])).item == null &&
10   @       ((Node)\old(nodeList[i])).next == null);
11   @*/
12 public void clear() {
13     //@ ghost \bigint index;
14     //@ set index = 0;
15     /*@
16       @ maintaining
17       @   0 <= index && index <= nodeList.length;
18       @ maintaining
19       @   x != null ==> index < nodeList.length;
20       @ maintaining
21       @   index < nodeList.length ==> x == nodeList[index];
22       @ maintaining
23       @   (\forall \bigint i; index<=i<nodeList.length-1;
24       @     ((Node)nodeList[i]).next==(Node)nodeList[i+1]);
25       @ maintaining
26       @   nodeList.length > 0 ==>
27       @     ((Node)nodeList[nodeList.length-1]).next == null;
28       @ maintaining
29       @   nodeList == \old(nodeList);
30       @ maintaining
31       @   (\forall \bigint i; 0 <= i < index;
32       @     ((Node)nodeList[i]).prev == null &&
33       @     ((Node)nodeList[i]).item == null &&
34       @     ((Node)nodeList[i]).next == null);
35       @ maintaining
36       @   (\forall \bigint i; index <= i < nodeList.length;
37       @     ((Node)nodeList[i]).prev ==
38       @       \old(((Node)nodeList[i]).prev) &&
39       @     ((Node)nodeList[i]).item ==
40       @       \old(((Node)nodeList[i]).item) &&
41       @     ((Node)nodeList[i]).next ==
42       @       \old(((Node)nodeList[i]).next));
43       @ decreasing
44       @   nodeList.length - index;
45       @*/
46     for (Node x = first; x != null;) {
47         Node next = x.next;
48         x.item = null;
49         x.next = null;
50         x.prev = null;
51         //@ set index = index + 1;
52         x = next;
53     }
54     first = last = null;
55     size = 0;
56     //@ set nodeList = \seq_empty;
57     modCount++;
58 }
```

Listing 4.11: clear() (annotated with JML)

```
1  ==>
2        wellFormed(heap)
3      & !self = null
4      & self.<created> = TRUE
5      & java.util.BoundedLinkedList::exactInstance(self) = TRUE
6      & (e = null | e.<created> = TRUE)
7      & measuredByEmpty
8      & (self.nodeList.length + 1 <= java.lang.Integer.MAX_VALUE & self.<inv>)
9    -> {heapAtPre:=heap || _e:=e}
10       \<{
11           exc=null;try {
12             result=self.add(_e)@java.util.BoundedLinkedList;
13           }   catch (java.lang.Throwable e) {
14             exc=e;
15           }
16         }\> (    self.nodeList
17                = seqConcat(self.nodeList@heapAtPre,
                    ↪  seqSingleton(self.nodeList[javaSubInt(self.nodeList.length, 1)]))
18              &
                ↪  (java.util.BoundedLinkedList.Node)(self.nodeList[javaSubInt(self.nodeList.length,
                ↪  1)]).item = e
19              & result = TRUE
20              & self.<inv>
21              & exc = null
22              & \forall Field f;
23                  \forall java.lang.Object o;
24                    ((o, f) \in allLocs | !o = null & !o.<created>@heapAtPre = TRUE | o.f =
                      ↪  o.f@heapAtPre))
```

Listing 4.12: add(Object), initial proof obligation

```
/*@
  @ also
  @ ...
  @ public normal_behavior
  @   requires
  @     o != null;
  @   ensures
  @     \result >= -1 && \result < nodeList.length;
  @   ensures
  @     \result == -1 ==>
  @       (\forall \bigint i; 0 <= i < nodeList.length;
  @         !o.equals(((Node)nodeList[i]).item));
  @   ensures
  @     \result >= 0 ==>
  @       (\forall \bigint i; \result < i < nodeList.length;
  @         !o.equals(((Node)nodeList[i]).item)) &&
  @       o.equals(((Node)nodeList[\result]).item);
  @*/
public /*@ strictly_pure @*/ int
lastIndexOf(/*@ nullable @*/ Object o) {
    int index = size;
    if (o == null) {
        ...
    } else {
        /*@
          @ maintaining
          @   (\forall \bigint i; index <= i < nodeList.length;
          @       !o.equals(((Node)nodeList[i]).item));
          @ maintaining
          @   0 <= index && index <= nodeList.length;
          @ maintaining
          @   0 < index && index <= nodeList.length ==>
          @     x == (Node)nodeList[index - 1];
          @ maintaining
          @   index == 0 <==> x == null;
          @ decreasing
          @   index;
          @ assignable
          @   \strictly_nothing;
          @*/
        for (Node x = last; x != null; x = x.prev) {
            index--;
            if (o.equals(x.item))
                return index;
        }
    }
    return -1;
}
```

Listing 4.13: lastIndexOf(Object) (annotated with JML). Searches the list from last to first for an element. Returns -1 if this element is not present in the list; otherwise returns the index of the node that was equal to the argument. Only the contract and branch in which the argument is non-`null` is shown due to space restrictions. Methods such as `indexOf`, `removeFirstOccurrence` and `removeLastOccurrence` are very similar.

Column abbreviations: Exc is the exception thrown, Br. is the number of proof branches, I.steps is the number of interactive steps, Q.ins is the number of quantifier instantiations, C. is the number of method contracts applied, li is the number of loop invariants, loc are lines of code, and los are lines of specification. Exception abbreviations: IOOBE is index out of bounds exception, ISE is illegal state exception, NSEE is no such element exception.

Table 4.11: Normal behavior

Nr	Visibility	Method	Rules	Br.	I.steps	Q.ins	C.	li	loc	los
1	private	checkElementIndex(int)	1,372	14	0	8	1	0	4	5
2	private	checkPositionIndex(int)	1,494	14	0	10	1	0	6	5
3	private	checkSize()	1,454	14	0	10	1	0	4	3
4	private	isElementIndex(int)	1,340	11	0	10	1	0	3	2
5	private	isMaxSize()	732	6	0	5	0	0	3	2
6	private	isPositionIndex(int)	1,340	11	0	10	0	0	3	2
7	private	linkFirst(Object)	33,650	180	665	277	0	0	9	8
11	private	unlinkFirst(Node)	11,186	114	276	30	0	0	13	7
12	private	unlinkLast(Node)	11,023	101	252	31	0	0	13	8
14	package	linkBefore(Object,Node)	28,146	222	531	109	0	0	10	13
15	package	linkLast(Object)	24,575	131	33	256	0	0	9	8
16	package	node(int)	12,952	54	226	21	0	2	13	23
17	package	unlink(Node)	65,799	261	1,604	426	0	0	19	11
18	public	add(Object)	1,675	11	185	7	2	0	5	8
19	public	add(int,Object)	5,576	41	423	14	5	0	6	10
22	public	addFirst(Object)	1,863	10	31	6	2	0	4	7
23	public	addLast(Object)	1,693	10	159	6	2	0	4	7
24	public	BoundedLinkedList()	562	3	0	0	0	0	1	2
26	public	clear()	16,340	130	1,068	35	0	1	12	42
28	public	contains(null)	960	12	51	3	1	0	4	12
28	public	contains(Object)	716	13	10	9	2	0	4	12

Table 4.11 – continued from previous page

Nr	Visibility	Method	Rules	Br.	I.steps	Q.ins	C.	li	loc	los
30	public	element()	2,044	27	0	28	1	0	3	5
31	public	get(int)	3,107	21	37	9	2	0	4	5
32	public	getFirst()	565	7	0	4	0	0	5	5
33	public	getLast()	574	7	0	4	0	0	5	5
34	public	indexOf(null)	7,114	49	99	10	0	1	12	30
34	public	indexOf(Object)	6,581	43	56	9	1	1	12	30
35	public	lastIndexOf(null)	4,144	29	51	6	0	1	12	30
35	public	lastIndexOf(Object)	4,571	26	23	7	1	1	12	30
37	public	offer(Object)	2,948	16	2	11	1	0	3	8
38	public	offerFirst(Object)	2,812	18	2	12	1	0	4	8
39	public	offerLast(Object)	2,010	10	2	6	1	0	4	8
40	public	peek()	942	8	0	5	0	0	4	4
41	public	peekFirst()	942	8	0	5	0	0	4	4
42	public	peekLast()	943	8	0	5	0	0	4	4
43	public	poll()	7,256	28	0	46	1	0	4	4
44	public	pollFirst()	21,716	115	0	362	1	0	4	5
45	public	pollLast()	3,068	13	1	2	1	0	4	5
46	public	pop()	10,195	40	4	105	1	0	3	6
47	public	push(Object)	4,215	24	3	17	2	0	3	7
48	public	remove()	3,397	27	4	26	1	0	3	6
49	public	remove(int)	1,862	26	958	4	3	0	4	8
50	public	remove(null)	7,042	72	201	15	1	1	11	36
50	public	remove(Object)	4,216	63	211	13	2	1	11	36
51	public	removeFirst()	632	8	11	1	1	0	5	6
52	public	removeFirstOccurrence()	967	12	60	3	1	0	3	15
52	public	removeFirstOccurrence(Object)	698	14	6	3	1	0	3	15

Table 4.11 – continued from previous page

Nr	Visibility	Method	Rules	Br.	I.steps	Q.ins	C.	li	loc	los
53	public	removeLast()	10,840	25	10	63	1	0	5	7
54	public	removeLastOccurrence()	4,225	36	27	9	1	1	11	34
54	public	removeLastOccurrence(Object)	4,497	38	26	9	2	1	11	34
55	public	set(int,Object)	1,509	29	41	17	4	0	8	7
56	public	size()	85	4	32	0	0	0	3	3
58	public	toArray()	21,221	102	420	115	1	1	7	37
		Total	371,386	2,316	7,801	2,214	51	12	340	644

Table 4.12: Exceptional behavior

Nr	Visibility	Method	Exc	Rules	Br.	I.steps	Q.ins	C.	li	loc	los
1	private	checkElementIndex(int)	IOOBE	294	9	184	2	3	0	6	5
2	private	checkSize()	ISE	155	7	90	2	2	0	4	4
3	private	checkPositionIndex(int)	IOOBE	279	10	160	2	3	0	6	5
18	public	add(Object)	ISE	134	6	54	5	1	0	5	5
19	public	add(int,Object)	ISE	665	11	79	9	1	0	6	5
19	public	add(int,Object)	IOOBE	173	8	89	5	2	0	6	7
22	public	addFirst(Object)	ISE	133	6	53	5	1	0	4	5
23	public	addLast(Object)	ISE	135	6	55	5	1	0	4	5
30	public	element()	NSEE	146	6	74	5	1	0	3	5
31	public	get(int)	IOOBE	158	6	82	6	1	0	4	5
32	public	getFirst()	NSEE	219	10	77	5	1	0	5	5
33	public	getLast()	NSEE	217	10	58	5	1	0	5	5
37	public	offer(Object)	ISE	173	7	91	6	1	0	3	5
38	public	offerFirst(Object)	ISE	171	6	90	6	1	0	4	5
39	public	offerLast(Object)	ISE	171	6	90	6	1	0	4	5

Table 4.12 – continued from previous page

Nr	Visibility	Method	Exc	Rules	Br.	I.steps	Q.ins	C.	li	loc	los
46	public	pop()	NSEE	155	6	75	6	1	0	3	5
47	public	push(Object)	ISE	171	6	90	6	1	0	3	5
48	public	remove()	NSEE	155	6	75	6	1	0	3	5
49	public	remove(int)	NSEE	147	6	65	5	1	0	4	6
51	public	removeFirst()	NSEE	226	10	64	5	1	0	5	5
53	public	removeLast()	NSEE	154	7	49	1	1	0	5	5
55	public	set(int,Object)	NSEE	147	6	64	5	1	0	8	5
			Total	4,378	161	1,808	108	28	0	100	112

Proof statistics. The below table summarizes the main proof statistics for all methods. The last two columns are not metrics of the proof, but they indicate the total lines of code (LoC) and the total lines of specifications (LoSpec). The number of interactive steps as a percentage of the number of rules is less than 3 percent.

We found the most difficult proofs were for the method contracts of: `clear()`, `linkBefore(Object,Node)`, `unlink(Node)`, `node(int)` and `remove(Object)`. The number of interactive steps seem a rough measure for effort required. But, we note that it is not a reliable representation of the difficulty of a proof: an experienced user can produce a proof with very few interactive steps, while an inexperienced user may take many more steps. The proofs we have produced are by no means minimal.

Table 4.13: Overall

Rules	Branches	Interactive	Quant.ins	Contract	LoopInv	LoC	LoSpec
375,764	2,477	9,609	2,322	79	12	440	756

Chapter 5

Discussion

In this section we discuss some of the main challenges of verifying the real-world Java implementation of a `LinkedList`, as opposed to the analysis of an idealized mathematical linked list. We will end with some concluding remarks.

5.1 Challenges

Extensive use of Java language constructs The `LinkedList` class uses a wide range of Java language features. This includes nested classes (both static and non-static), inheritance, polymorphism, generics, exception handling, object creation and foreach loops. To load and reason about the real-world `LinkedList` source code requires an analysis tool with high coverage of the Java language, including support for the aforementioned language features.

Support for intricate Java semantics The Java `List` interface is position based, and associates with each item in the list an index of Java's `int` type. The bugs described in Section 3.2 were triggered on large lists, in which integer overflows occurred. Thus, while an idealized mathematical integer semantics is much simpler for reasoning, it could not be used to analyze the bugs we encountered! It is therefore critical that the analysis tool faithfully supports Java's semantics, including Java's integer (overflow) behavior.

Collections have a huge state space A Java collection is an object that contains other objects (of a reference type). Collections can typically grow to an arbitrary (but in practice, bounded) size. By their very nature, collections thus intrinsically have a large state. To make this more concrete: triggering the bugs in `LinkedList` requires at least 2^{31} elements (and 65 gigabytes of memory), and each element, since it is of a reference type, has at least 2^{32} values. This poses serious problems to fully automated analysis methods that explore the state space.

Interface specifications Several of the `LinkedList` methods contain an interface type as parameter. For example, the `addAll` method takes two arguments, the second one is of the `Collection` type, which is shown in Listing 5.1. As KeY follows the design

```
public boolean addAll(int index, Collection c) {
  ...
  Object[] a = c.toArray();
  ...
}
```

Listing 5.1: AddAll() uses Collection interface

by contract paradigm, verification of `LinkedList`'s `addAll` method requires a contract

for each of the other methods called, including the `toArray` method in the `Collection` *interface*. How can we specify interface methods, such as `Collection.toArray`? The stub generator generates a conservative contract: it may arbitrarily modify the heap and return *any* array. Simple conditions on parameters or the return value are easily expressed, but meaningful contracts that relates the behavior of the method to the contents of the collection require some notion of state to capture all mutations of the collection, so that previous calls to methods in the interface that contributed to the current contents of the collection are taken into account. Model fields/methods [11, Section 9.2] are a widely used mechanism for abstract specification. A model field or method is represented in a concrete class in terms of the concrete state given by its fields. In this case, as only the interface type `Collection` is known rather than a concrete class, such a representation cannot be defined. Thus the behavior of the interface cannot be fully captured by specifications in terms of model fields/variables, including for methods such as `Collection.toArray`. Ghost variables cannot be used either, since ghost variables are updated by adding set statements in method bodies, and interfaces do not contain method bodies. This raises the question: how to specify behavior of interface methods?[87]

Verifiable code revisions We fixed the `LinkedList` class by explicitly bounding its maximum size to `Integer.MAX_VALUE` elements, but other solutions are possible. Rather than using integers indices for elements, one could change to an index of type `long` or BigInteger. Such a code revision is however incompatible with the general `Collection` and `List` interfaces (whose method signatures mandate the use of integer indices), thereby breaking all existing client code that uses `LinkedList`. Clearly this is not an option in a widely used language like Java, or any language that aims to be backwards compatible.

It raises the challenge: can we find code revisions that are compatible with existing interfaces and client classes? We can take this challenge even further: can we use our workflow to find such compatible code revisions, *and are also amenable to formal verification?* The existing code in general is not designed for verification. For example, the `LinkedList` class exposes several implementation details to classes in the java.util package: i.e., all fields, including `size`, are package private (not private!), which means they can be assigned a new value directly (without calling any methods) by other classes in that package. This includes setting `size` to negative values. As we have seen, the class malfunctions for negative `size` values. In short, this means that the `LinkedList` itself cannot enforce its own invariants anymore: its correctness now depends on the correctness of other classes in the package. The possibility to avoid calling methods to access the lists field may yield a small performance gain, but it precludes a modular analysis: to assess the correctness of `LinkedList` one must now analyze all classes in the same package (!) to determine whether they make benign changes (if any) to the fields of the list. Hence, we recommend to encapsulate such implementation details, including making at least all fields `private`.

Proof reuse The total effort of our case study was about 7 man months. The largest part of this effort is finding the right specification. KeY supports various ways to specify Java code: model fields/methods, pure methods, and ghost variables. For example, using pure methods, contracts are specified by expressing the content of the

[87] Since the representation of classes that implement the interface is unknown in the interface itself, a particularly challenging aspect here is: how to specify the footprint of an interface method, i.e.: what part of the heap can be modified by the method in the implementing class?

list before/after the method using the pure method get(i), which returns the item at index i. This led to rather complex proofs: essentially it led to reasoning in terms of relational properties on programs (i.e. `get(i)` before vs get(i) after the method under consideration). After 2.5 man months of writing partial specifications and partial proofs in these different formalisms, we decided to go with ghost variables as this was the only formalism in which we succeeded to prove non-trivial methods.

It then took $\approx$ 4 man months of iterating in our workflow through (failed) partial proof attempts and refining the specs until they were sufficiently complete. In particular, changes to the class invariant were "costly", as this typically caused proofs of all the methods to break (one must prove that all methods preserve the class invariant). The possibility to interact with the prover was crucial to pinpoint the cause of a failed verification attempt, and we used this feature of KeY extensively to find the right changes/additions to the specifications.

It reveals that while the total effort was 6-7 person months, once the specifications are in place after many iterations of the workflow, producing the actual final proofs took only 1-2 weeks! But minor specification changes often require to redo nearly the whole proof, which causes much delay in finding the right specification. Other program verification case studies [11, 28, 16, 2] show similarly that the main bottleneck today is specification, not verification. This calls for techniques to optimize proof reuse when the specification is slightly modified, allowing for a more rapid development of specifications.

5.1.1 Status of the Challenges

Most of these challenges are still open. The challenge concerning "Interface specifications" could perhaps be addressed by defining an abstract state of an interface by using/developing some form of a trace specification that map a sequence of calls to the interface methods to a value, together with a logic to reason about such trace specifications.

The challenges related to code revisions and proof reuse are compounded for analysis tools that use very fine-grained proof representations. For example, proofs in KeY consist of actual rule applications (rather than higher level macro/strategy applications), and proof rule applications explicitly refer to the indices of the (sub) formulas the rule is applied to. This results in a fragile proof format, where small changes to the specifications or source code (such as a code refactoring) break the proof.

The KeY system covered the Java language features sufficiently to load and statically verify the `LinkedList` source code. KeY also supports various integer semantics, allowing us to analyze LinkedList with the actual Java integer overflow semantics. As KeY is a theorem prover (based on deductive verification), it does not explore the state space of the class under consideration, thus solving the problem of the huge state space of Java collections. We could not find any other tools that solved these challenges, so we decided at that point to use KeY.

However, other state-of-the-art systems such as Coq, Isabelle and PVS support proof *scripts*. Those proofs are described at a typically much more coarse-grained level when compared to KeY. It would be interesting to see to what extent Java language features and semantics can be handled in (extensions of) such higher level proof script languages.

5.2 Reflection

Initially we started writing specifications using the query method `get(int index)`. As an example we consider Listing 5.2, which shows method `addFirst(Object)`. It inserts the specified element at the beginning of the list. This preliminary con-

```
/*@
  @ public normal_behavior
  @   requires
  @     0 <= size() < Integer.MAX_VALUE;
  @   ensures
  @     size() == (\bigint)\old(size()) + 1 &&
  @     get(0) == e &&
  @     (\forall int i; 1 <= i < size(); get(i) == \old(get(i-1)));
  @*/
public void addFirst(/*@ nullable @*/ Object e) {
    checkSize(); // new
    linkFirst(e);
}
```

Listing 5.2: JML addFirst(), preliminary, bounded

tract is expressed in terms of methods `size()` and `get(int index)`. By calling `checkElementIndex(index)`, method `get(int index)` (see Listing 5.3) first checks whether the given index is in its intended range ($0 \leq$ `size` $\wedge$ `size` $<$ `size()`), and if it is, it calls method `node(int index)` (otherwise `checkElementIndex(index)` throws an error). Method `node(int index)` (see Listing 5.4) searches for the position in the list based on the given index and then returns the corresponding node to its caller. The postcondition of `addFirst(Object)` states that the size has been incremented

```
/*@
  @ also
  @ public exceptional_behavior
  @   requires
  @     index < 0 || index >= size();
  @   signals_only IndexOutOfBoundsException;
  @   signals (IndexOutOfBoundsException e) true;
  @*/
public /*@ nullable strictly_pure @*/ Object get(int index) {
    checkElementIndex(index);
    return node(index).item;
}
```

Listing 5.3: JML get(int index), preliminary. Only annotated w.r.t. exceptional behavior.

with 1, that the specified element is the first item, and that all other elements are unaffected, i.e., their order and content are unaltered. In short: the list in the post-state is the same as in the prestate, except a new element (the specified one) has been inserted at the front of the list. In other words: the list in the poststate is the concatenation of the specified element and the list in the prestate. The use of the `get` method in the postcondition is such that it will not throw an error; i.e., the `ensures` clause is well-defined. Listing 5.5 shows a more compact form of the poststate: compared to Listing 5.2, it shows a rewritten `ensures` clause using a *model* field, viz., `theList`. This field uses the query-method as shown in Listing 5.6. The query method `get(int index)` is incorporated in the contract of `addFirst(Object)`, either direct (Listing 5.2) or indirect (Listing 5.5). This rises the question: how can we specify the query method it self? As it is easy to specify `checkElementIndex(index)`, this boils down to the question: how can we specify method `node(int index)`? Only `first` and `last` are directly accessible, i.e., only when `index == 0` or `index == size() -`

```
/*@ strictly_pure @*/ Node node(int index) {
    // assert isElementIndex(index);
    if (index < (size >> 1)) {
        Node x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

Listing 5.4: JML node(int index), preliminary. Only annotated w.r.t. purity level.

1 we can directly relate `index` to the node that is returned by `node(int index)`. The seemingly impossibility to directly access whatever node without using the query method imposed a problem in specifying `BoudedLinkedList`. A substantial part of the research period was already behind us when we got on the right track. What we actually needed was a mathematical abstraction of the linked list that is *not* directly related to its state. Thus: we needed a ghost field. We already were trying to use a model field of type `\seq`, and we realized that we could use this type also for a ghost field. Modifications in the linked list could be managed by the `set` command, and the type also offers functions that can be used in postconditions (and in the `set` command). We then introduced the field `nodeList`: we wanted this field to represent the nodes in the linked list, in the same order. Then we would be able to use this field in method contracts and in the class invariant as well. More over, the need to use the query method for specifications had vanished. The first step now was to verify method `node(int index)`, to see if things went as we were hoping. Method `node(int index)` annotated with JML using the ghost field is shown in Listing 4.8, and verifying the method went well. We then used the ghost field everywhere we thought it was appropriate and useful. Normal behavior JML for `get(index int)` is shown in Listing 5.7. Once we used the ghost field, specifying went more smoothly, and verification became relatively easy. This emphasises the importance of correct specification, and that incorrect specification is the bottleneck in formal verification, as stated in a previous section.

```
/*@
  @ public normal_behavior
  @   requires
  @     0 <= size() < Integer.MAX_VALUE;
  @   ensures
  @     theList == \seq_concat(\seq_singleton(e),\old(theList));
  @*/
```

Listing 5.5: JML addFirst(), preliminary, bounded, model field

```
/*@
 *@ public instance model \seq theList;
  @   represents theList \such_that
  @     (\forall int i; 0 <= i < size(); theList[i] == get(i));
  @*/
```

Listing 5.6: JML model field theList

```
/*@
  @ public normal_behavior
  @   requires
  @     0 <= index < theList.length;
  @     // this implies theList.length > 0, i.e., theList != \seq_empty
  @   ensures
  @     \result == ((Node)theList[index]).item;
  @*/
```

Listing 5.7: JML get(int index), normal behaviour. Using ghost field.

5.2.1 Future Work

There is a blind spot in what we have done. When we look at section 4.2.3, where method `set(int index, Object element)` is treated, we observe that `nodeList` is unaffected. This is since of a specific node its field `item` is altered. The node as such, i.e., its place in the list and the instance of type `Node` it refers to, are unaffected. It would be better if a contract would made it explicit which items have been changed, and more over, which ones not. Let's put it this way: what if method `add(Object e)` (see section 4.2.1) would secretly clear field `item` of pre-existing nodes? In the current constellation, the contract would still be valid. More over, let us realize that type `Node` is not publicly visible in the first place. This type is an internal construct that is needed to link nodes to each other. From a client perspective, only the data items and their mutual order are of importance. As a (possible) future enhancement we propose an additional model or ghost field, e.g., like shown in Listing 5.8. Listing 5.9 shows a (possible) contract of the `set` method where items have been taken into account.[88] Since `itemList` is a model field directly related to `nodeList`, altering the class invariant may be not necessary. It may turn out to be easier to make `itemList` a ghost field as well. If it is, the JML `set` command should be added in the `set` method.

```
//@ public model \seq itemList;
//@ represents itemList = (\seq_def \bigint i; 0; nodeList.length; ((Node)nodeList[i]).item);
```

Listing 5.8: Specification field itemList. Here shown as a model field.

5.3 Remarks and Conclusion

A number of methods have been verified by Hans-Dieter A. Hiep. Jinting Bian has re-verified method `unlinkLast(Node)` with less interaction than in a previous version of the proof. This has been recorded in a video of a 30 minute proof session [29]. Table 4.13 and the text above it have been taken from [24], the TACAS paper mentioned in section 1.5. Section 4.3.4.1, which describes the proof of `lastIndexOf(Object)`, is based on a similar part of the article, but has been rewritten in this thesis (i) to have a proof with less interaction, and (ii) to write the proof down a bit more low-level. The proof files, the annotated source code, and the KeY version used in this thesis coincide with that of the article. An archive is available on-line in the Zenodo repository [30]. The test cases used in the article are also in that archive.

[88] KeY 'loads' this contract, i.e., the syntax is correct. No warranties are given that this contract is correct. It does however give an idea of what it takes to reason about the items.

```
/*@
  @ also
  @ public normal_behavior
  @   requires
  @     0 <= index < itemList.length;
  @   ensures
  @     itemList[index] == element &&
  @     (\forall \bigint i; 0 <= i < itemList.length && i != index;
  @      itemList[i] == \old(itemList[i])) &&
  @     \result == \old(itemList[index]);
  @ public exceptional_behavior
  @   requires
  @     index < 0 || index >= itemList.length;
  @   signals_only IndexOutOfBoundsException;
  @   signals (IndexOutOfBoundsException e) true;
  @*/
public /*@ nullable @*/ Object
set(int index, /*@ nullable @*/ Object element) {
    checkElementIndex(index);
    Node x = node(index);
    Object oldVal = x.item;
    x.item = element;
    return oldVal;
}
```

Listing 5.9: set(int index, Object element) (annotated with JML, item based)

Bug tracker. A fixed version of `LinkedList` has been submitted to Oracle's bug tracker. This includes test cases that show the absence of the integer overflow bug for `size` for all methods that originate this bug.

Risks. KeY has a steep learning curve; especially when a user has little or no background in the field of formal methods, like the student. This implied a risk of not getting enough things done. This risk was mitigated when the decision was made to collaborate with Hans-Dieter A. Hiep, who proved a number of methods of `BoundedLinkedList`, and who was willing to share his knowlegde.

5.3.1 Conclusion

We were able to find bugs in `LinkedList` (RQ1); initially when writing specifications and additionally when examining the source code in conjunction with the informal Java documentation. The resulting analysis enabled us to synthesize test cases that expose the identified bugs (RQ2). It also directed us in thinking about a revision that no longer contains the identified bugs (RQ3). Finally, we were able to verify the fixed version to a large extent (RQ4). Moreover, a fixed version was tested on the absence of bugs, and submitted to the bug tracker of Oracle. As a future enhancement, taking items into account is recommended w.r.t. verification of `LinkedList`.

Bibliography

[1] S. de Gouw, F. S. de Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel. "Verifying OpenJDK's Sort Method for Generic Collections". In: *Journal of Automated Reasoning* (Aug. 2017). ISSN: 1573-0670. DOI: 10.1007/s10817-017-9426-4. URL: https://doi.org/10.1007/s10817-017-9426-4 (cit. on p. 1).

[2] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. "OpenJDK's java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case". In: *CAV 2015: Computer Aided Verification*. Vol. 9206. LNCS. Springer, 2015, pp. 273–289. DOI: 10.1007/978-3-319-21690-4_16 (cit. on pp. 1, 3, 73).

[3] "Systems and software engineering – Vocabulary". In: *ISO/IEC/IEEE 24765:2010(E)* (Dec. 2010), pp. 1–418. DOI: 10.1109/IEEESTD.2010.5733835 (cit. on p. 1).

[4] I. Sommerville. *Software Engineering*. 9th ed. Harlow, England: Addison-Wesley, 2010. ISBN: 978-0-13-703515-1 (cit. on p. 2).

[5] *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990. URL: https://standards.ieee.org/standard/610_12-1990.html (cit. on p. 2).

[6] D. Bjørner and K. Havelund. "40 years of formal methods". In: *International Symposium on Formal Methods*. Springer. 2014, pp. 42–61 (cit. on p. 2).

[7] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. "Beyond assertions: Advanced specification and verification with JML and ESC/Java2". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 342–363 (cit. on p. 2).

[8] D. Cok and G. T. Leavens. "Extensions of the theory of observational purity and a practical design for JML". In: *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*. Vol. 4000. 2008, pp. 43–50 (cit. on p. 2).

[9] G. T. Leavens, A. L. Baker, and C. Ruby. "JML: A Notation for Detailed Design". In: *Behavioral Specifications of Businesses and Systems*. 1999, pp. 175–188 (cit. on p. 2).

[10] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. "The KeY system: Integrating object-oriented design and formal methods". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2002, pp. 327–330 (cit. on p. 2).

[11] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6 (cit. on pp. 2, 3, 45, 47, 72, 73).

[12] B. Beckert. "A Dynamic Logic for the Formal Verification ofJava Card Programs". In: *Java on Smart Cards: Programming and Security*. Springer, 2001, pp. 6–24 (cit. on p. 3).

[13] D. Harel, D. Kozen, and J. Tiuryn. "Dynamic logic". In: *Handbook of philosophical logic*. Springer, 2001, pp. 99–217 (cit. on p. 3).

[14] J. C. King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394 (cit. on p. 3).

[15] B. Beckert and R. Hähnle. "Reasoning and Verification: State of the Art and Current Trends". In: *Intelligent Systems, IEEE* 29.1 (Feb. 2014), pp. 20–29. ISSN: 1541-1672. DOI: 10.1109/MIS.2014.3 (cit. on p. 3).

[16] S. de Gouw, F. S. de Boer, and J. Rot. "Proof Pearl: The KeY to Correct and Stable Sorting". In: *J. Autom. Reasoning* 53.2 (2014), pp. 129–139. DOI: 10.1007/s10817-013-9300-y (cit. on pp. 3, 73).

[17] C. David, D. Kroening, and M. Lewis. "Propositional reasoning about safety and termination of heap-manipulating programs". In: *European Symposium on Programming Languages and Systems*. Springer. 2015, pp. 661–684 (cit. on p. 3).

[18] M. Brain, C. David, D. Kroening, and P. Schrammel. "Model and proof generation for heap-manipulating programs". In: *European Symposium on Programming Languages and Systems*. Springer. 2014, pp. 432–452 (cit. on p. 4).

[19] V. D'Silva, L. Haller, and D. Kroening. "Abstract conflict driven learning". In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 143–154 (cit. on p. 4).

[20] M. Lewis and D. Phil. "Precise Verification of C Programs". PhD thesis. University of Oxford, 2014 (cit. on p. 4).

[21] A. Knüppel, T. Thüm, C. Pardylla, and I. Schaefer. "Experience Report on Formally Verifying Parts of OpenJDK's API with KeY". In: *F-IDE 2018: Formal Integrated Development Environment*. Vol. 284. EPTCS. OPA, 2018, pp. 53–70. DOI: 10.4204/EPTCS.284.5 (cit. on p. 4).

[22] N. Polikarpova, J. Tschannen, and C. A. Furia. "A fully verified container library". In: *FM 2015: Formal Methods*. Vol. 9109. LNCS. Springer, 2015, pp. 414–434. DOI: 10.1007/978-3-319-19249-9_26 (cit. on p. 4).

[23] V. Klebanov, P. Müller, et al. "The 1st Verified Software Competition: Experience Report". In: *FM 2011: Formal Methods*. Vol. 6664. LNCS. Springer, 2011, pp. 154–168. DOI: 10.1007/978-3-642-21437-0_14 (cit. on p. 4).

[24] H.-D. A. Hiep, O. Maathuis, J. Bian, F. S. de Boer, M. van Eekelen, and S. de Gouw. "Verifying OpenJDK's LinkedList using KeY". In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, to appear*. 2020 (cit. on pp. 5, 76).

[25] O. Maathuis. *Verifying OpenJDK's LinkedList using KeY: Additional artefacts belonging to master thesis*. 2020. DOI: 10.5281/zenodo.3700776 (cit. on pp. 6, 25, 26).

[26] R. Hähnle, N. Wasser, and R. Bubel. "Array abstraction with symbolic pivots". In: *Theory and Practice of Formal Methods*. Springer, 2016, pp. 104–121 (cit. on p. 47).

[27] R. Bubel, R. Hähnle, and B. Weiß. "Abstract interpretation of symbolic execution with explicit state updates". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2008, pp. 247–277 (cit. on p. 47).

[28] C. Baumann, B. Beckert, H. Blasum, and T. Bormer. "Lessons Learned From Microkernel Verification—specification is the New Bottleneck". In: *SSV 2012: Systems Software Verification*. Vol. 102. EPTCS. OPA, 2012, pp. 18–32. DOI: 10.4204/EPTCS.102.4 (cit. on p. 73).

[29] J. Bian and H. A. Hiep. *Verifying OpenJDK's LinkedList using KeY: Video*. 2019. DOI: 10.6084/m9.figshare.10033094.v2 (cit. on p. 76).

[30] H. A. Hiep, O. Maathuis, J. Bian, F. S. de Boer, M. van Eekelen, and S. de Gouw. *Verifying OpenJDK's LinkedList using KeY: Proof Files*. 2019. DOI: 10.5281/zenodo.3517081 (cit. on p. 76).