Verification of Red-Black Trees in KeY – A Case Study in Deductive Java Verification

Bachelor's Thesis by

Johanna Stuber

at the KIT Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer:	Prof. Bernhard Beckert
Advisor:	Dr. Mattias Ulbrich
Second advisor:	Wolfram Pfeifer, M. Sc.

09.05.2023 – 11.09.2023

Karlsruher Institut für Technologie
KIT-Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that the work presented in this thesis is entirely my own. I confirm that I specified all employed auxiliary resources and clearly acknowledged anything taken verbatim or with changes from other sources. I further declare that I prepared this thesis in accordance with the rules for safeguarding good scientific practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, 11.09.2023

. . . .

(Johanna Stuber)

Abstract

While thorough testing of software reduces the likelihood of errors, formal verification can guarantee the correct behaviour of programs for all inputs through proofs. Consequently, for the ubiquitous data structures and algorithms like those found in standard libraries, used by thousands of applications, formal verification is especially desirable.

In this case study, we specify and verify a Java implementation of red-black trees with KeY. The KeY system is a tool for the formal verification of Java programs, and uses *dynamic frames* for memory reasoning, that is, framing. *Red-black trees* are a popular data structure for the efficient storage and retrieval of ordered elements, for example implemented in the class `java.util.TreeMap` of the Java Class Library. There exist various case studies in both areas – those in KeY, however, rarely consider tree structures, and existing red-black-tree verifications use different approaches to framing compared to KeY.

In this work, we conclude that successful reasoning over tree structures with dynamic frames is possible, yet very work-intensive compared to other framing approaches. Apart from this, we explore general strengths and limitations of KeY and give suggestions on how to improve its usability. Furthermore, we test out methods for automising and persisting proofs with the new KeY features of *JML Scripts* and *Proof Caching*.

Zusammenfassung

Während das ausführliche Testen von Software das Auftreten von Fehlern unwahrscheinlicher macht, kann mit formaler Verifikation durch Beweise garantiert werden, dass sich ein Programm für sämtliche Eingaben korrekt verhält. Besonders für tausendfach verwendete Grundelemente wie die Datenstrukturen und Algorithmen einer Standardbibliothek ist eine formale Verifikation daher erstrebenswert.

In dieser Fallstudie spezifizieren und verifizieren wir eine Java-Implementierung von Rot-Schwarz-Bäumen mit KeY. KeY ist ein Tool zur formalen Verifikation von Java-Programmen, und verwendet dabei *Dynamic Frames* für Aussagen über Speicherbereiche, also das Framing. *Rot-Schwarz-Bäume* sind eine beliebte Datenstruktur für das effiziente Speichern und Auslesen von Elementen und sind zum Beispiel in der Klasse `java.util.TreeMap` der Java Class Library umgesetzt. In beiden Bereichen existieren verschiedenste Fallstudien – die in KeY betrachten jedoch kaum Baumstrukturen und existierende Rot-Schwarz-Baum-Verifizierungen gehen auf andere Weise als KeY mit Framing um.

In dieser Arbeit kommen wir zu dem Schluss, dass die Verifikation von Baumstrukturen mit Dynamic Frames möglich ist, jedoch im Vergleich zu anderen Ansätzen viel zusätzlichen Aufwand mit sich bringt. Darüber hinaus erkunden wir generelle Stärken und Schwächen von KeY und machen einige Vorschläge zur Verbesserung der Benutzbarkeit. Außerdem testen wir mit *JML Scripts* und *Proof Caching* neue Methoden zur Beweis-Automatisierung und -Persistierung.

Contents

List of Figures and Tables

List of Listings

1 Introduction

1.1 Motivation

The standard libraries of programming languages are an indispensable part of modern software development. Immense trust is put in the correct behaviour of, among other things, basic data structures and their access operations, as tens of thousands of applications rely on them. Yet, there are no guarantees that they indeed provide what we are expecting.

This is where formal methods can be of great value. Applied to standard library implementations, they can uncover existing issues or prove intended properties to hold true. Ideally, they result in formally verified guarantees that fully legitimise the trust that is currently only based on testing – which can reduce the likelihood of errors, but not prove the absence thereof.

One tool enabling the use of formal methods for Java code is the KeY system (Ahrendt et al., 2016)[1]. Developed since 1998, it has recently been used for a number of case studies looking at implementations in the Java standard library, for example by de Gouw et al. (2015), de Boer et al. (2022), and Hiep et al. (2022). However, until now none of these more elaborate efforts have dealt with tree structures, a basis for many advanced data structures.

This thesis conducts a case study about the use of KeY for the verification of red-black trees. These are a particularly efficient version of self-balancing binary search trees and, as such, are a fundamental data structure widely used for the efficient storage and retrieval of ordered elements. With its class `java.util.TreeMap`, the Java Class Library (JCL) provides an implementation of red-black trees; in addition, it uses them internally in the class `java.util.HashMap`.[2]

Red-black trees are also a popular goal for formal verification. With VACID-0, Leino and Moskal (2010) even propose them (amongst four other data structures) as a means to benchmark verification systems. To the best of our knowledge, this work is the first to verify red-black trees with a tool that uses dynamic frames for memory reasoning.

The main goal of this thesis is thus to contribute to a growing collection of formally verified data structures and algorithms – working towards those of the JCL – while pushing the limits of KeY in order to gain insights into its strengths as well as potential for further improvements.[3]

[1] `https://www.key-project.org/`

[2] The source code of `TreeMap` and `HashMap` can be found on GitHub at `https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util`

[3] The complete source code with specifications and the proofs resulting from this thesis are available at `https://github.com/gewitternacht/rbtree-verification`

1.2 Contributions

With this thesis, we present a case study that makes the following contributions:

- We specify and successfully verify a Java implementation of the `contains` and `add` methods of red-black trees. In doing so, we work towards the verification of real-world code from the Java Class Library. Even the simplified (but comparably efficient) code we use for our study already contributes to a fully verified algorithmic "Basic Tool Box". This also paves the way for future verification efforts dealing with similar structures.

- We explore and push the limits of KeY, in particular regarding its approach to framing, and conclude that successful reasoning over tree structures with dynamic frames is possible, yet very work-intensive. We demonstrate its limitations, and strategies to deal with arising problems.

- With the heavy use of assertions and JML Scripts, and a look at Proof Caching, we test out new ways to automate and persist proofs in KeY, and give suggestions on how to further improve them.

- Generally, we gain insights into strengths, limitations, and also some bugs of KeY and give pointers to potential improvements regarding usability and functionality. By approaching one of the proposed VACID-0 data structures, we enable the benchmarking and comparison of different verification systems based on relatively complex real-world examples.

1.3 Outline

The remainder of this thesis is structured as follows: Chapter 2 introduces red-black trees on an algorithmic level, as well as the necessary basics of JML and KeY that are used in this case study. It further discusses the frame problem and its relevance for this thesis, and gives an overview of related work. Next, Chapter 3 describes the design decisions made for the implementation and the fundamental definitions for the specification. It also defines the method contracts that constitute the goals to be verified. Chapter 4 then deals with the most important steps and challenges of the verification process with KeY and illustrates this by providing a few concrete examples of proof situations. In addition, statistics are given for all proofs. The insights gained during the course of this thesis are detailed in Chapter 5, which includes some desirable new features for KeY. Chapter 6 concludes with a summary of the most important results of this work and gives an outlook for possible future work.

2 Background

This chapter starts by introducing red-black trees and the insertion operation on them on an algorithmic level. The explanations in the first section are based on *Introduction to Algorithms* (Leiserson et al., 1994, Chapter 12 and 13). The next three sections, dealing with the basics of JML and KeY, as well as the important *frame problem*, are based on *The KeY Book* (Ahrendt et al., 2016). Finally, an overview of existing related work concludes this chapter.

2.1 Red-Black Trees

2.1.1 Binary Search Trees

As red-black trees are a version of self-balancing binary search trees, we start by looking at the properties and usage of this more general concept.

Property Generally, search trees are used for ordered storage of elements, often for the entries of a map data structure, that is, a set of key-value pairs. In binary search trees, these entries are structured as a tree where each node contains one key (and possibly additional data) and two references to child nodes. The *binary-search-tree property* states that these keys must be ordered in a way that if a node n contains the key k_n

- every key k_l in the left subtree of n must not be greater than k_n, i.e. $k_l \leq k_n$
- every key k_r in the right subtree of n must not be smaller than k_n, i.e. $k_r \geq k_n$.

Search With that, we only have to traverse a tree once from the root node to a leaf to check whether it contains a specific key k: By comparing k with the key of the node we are currently looking at, we can determine if our search must be continued in the left or right subtree of the node. We do this until we have found a node with the requested key k or until we reach a leaf of the tree, in this case concluding that k is not present in the tree.

Insertion The same approach can be used to correctly insert a new element with key k' into the tree: We traverse the tree in the same manner down to a leaf where we would have expected k' in a valid binary search tree. We then simply attach a new node in this place. In order to ensure unique keys for map-like data structures, we can decide to abort the insertion or opt for a replacement if we detect that k' is already contained in the tree.

Complexity The search and insertion operations consequently run in a time proportional to the height of the tree. For balanced trees, which have a height that is logarithmic in the number of elements n they contain, the complexity of these operations thus is in $O(\log n)$. However, the naive insertion described above could lead to trees far from balanced, in the worst case resulting in a tree consisting of a single branch, resembling a linked list. It is therefore desirable for modifying operations to rearrange the elements of the tree into a balanced structure; the implementation of such operations is what makes a search tree *self-balancing*.

2.1.2 Red-Black Properties

Properties Red-black trees are binary search trees where each node is additionally coloured either red or black. Non-existent children of nodes are called leaves – depending on the concrete implementation, this could be `null` pointers or a designated `NIL`-node. The following constraints on the possible combinations of colours in the tree guarantee that a valid red-black tree is approximately balanced:

- All leaves must be black.
- The root node must be black.
- *no double red*: No two adjacent nodes may be red; in other words: both children of a red node must be black.
- *black balanced*: The black heights of both children of a node must be equal, where the *black height* of a subtree is defined as the maximum number of black nodes on a path from the root node of the subtree to any leaf. An alternative formulation of this property states that "for each node, all paths from the node to descendant leaves contain the same number of black nodes" (Leiserson et al., 1994, p. 273).

Figure 2.1 illustrates some exemplary (in)valid red-black trees.

Consequence Intuitively, the combination of *no double red* and *black balanced* ensures that the longest possible path in a tree (alternating red and black nodes) is at most twice as long as the shortest possible path in a tree (only black nodes). As a consequence, the height of a tree with n elements is in $O(\log n)$[1] and thus the tree is approximately balanced if it satisfies all red-black properties. A formal proof of this fact can be found in Leiserson et al. (1994, p. 274). For red-black trees to be self-balancing, the insertion and deletion operations therefore have to ensure that the red-black properties hold again afterwards.

2.1.3 Inserting an Element

For the insertion of an element into a red-black tree, the first step is to perform a naive insertion as described for binary search trees in Section 2.1.1. The newly added node is always coloured red. This, however, could violate the *no double red* property if the

[1] to be precise, a valid red-black tree with n nodes has a height of at most $2\log(n+1)$

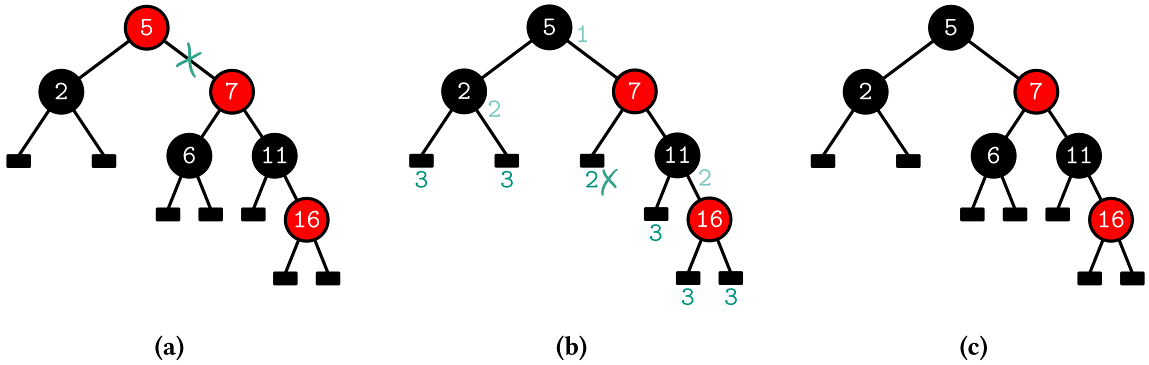


Figure 2.1: Three examples of red-black trees: (a) violates the *no double red* property and does not have the required black root; (b) violates the *black balanced* property as indicated by the numbers counting the black nodes on each branch; (c) is a completely valid red-black tree, however with maximum imbalance

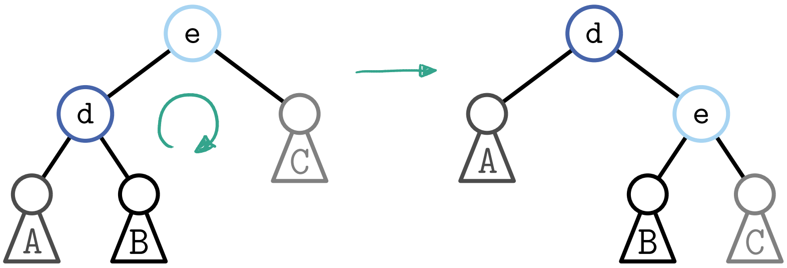
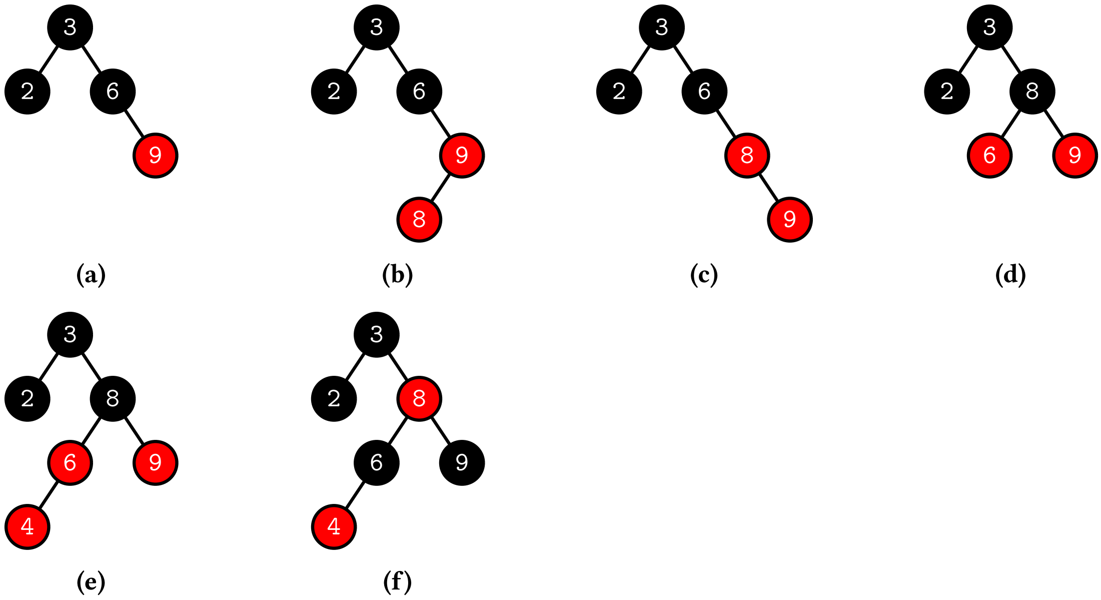


Figure 2.2: A right rotation: The nodes with keys `d` and `e` "pivot" around their link, swapping their roles of parent and child. The pointers to the subtrees `A` and `C` do not change, but `B` is reattached from the former to the new child node. One can easily see that the rotation preserves the binary-search-tree property.

parent of the new node is also red, resulting in a *double red*. Because of that, different local fix operations have to be performed to restore this property and make the tree a valid red-black tree again, while also preserving the binary-search-tree property. Depending on the specific situation, this could mean only changing the colours of nodes or actually modifying the tree structure with tree rotations.

Tree Rotations The concept of rotations on parts of binary search trees is also used for other variants of self-balancing search trees, e.g. AVL trees (Adel'son-Velskii and Landis, 1962). Rotations locally change a constant number of pointers in the tree and can help restore balance while keeping the search-tree structure valid. Figure 2.2 depicts a right rotation – the other kind of rotation, a left rotation, is the inverse thereof. Essentially, two nodes that are parent and child swap their roles through a rotation, with the former child becoming the new root node of the rotated subtree.

Fix Operations After the naive insertion of a new node x, the handling of the potential *double red* depends on the colour of x's uncle u, that is, the sibling of x's parent. If u is



(a) **(b)** **(c)** **(d)**

(e) **(f)**

Figure 2.3: (a) a valid red-black tree with four nodes; (b) the insertion of 8 produces a *double red* and as its uncle is `null`, thus black (see first property in Section 2.1.2), rotations are performed: (c) the right rotation on 9 is a sometimes necessary “normalisation rotation” to transform the *inner double red* (here `right` and `right.left`) to an *outer double red* (here `right` and `right.right`); (d) the *outer double red* is fixed by a left rotation on 6; (e) the insertion of 4 produces another *double red* and as the uncle of 4 (that is, 9) is red, recolouring is performed: (f) after the recolouring, the violation is fixed completely, but in other cases could have been propagated two levels up (to 3-8 being *double red*)

black, either one or two tree rotations resolve the violation completely, as illustrated in Figure 2.3(a)-(d). If u is red, *recolouring* is performed, as shown in Figure 2.3(e)-(f). This potentially propagates the *double red* up the tree, necessitating the same procedure two levels closer to the root of the tree. If the *double red* eventually reaches the root node, it is resolved by simply colouring the root node black again.

Complexity The worst case for the cost of the insertion is to traverse the tree once from the root node down to a leaf and then all the way up to the root node again while performing the fix operations. Since recolouring and rotations have a constant cost and red-black trees are balanced, the insertion thus runs in $O(\log n)$ for a tree with n elements.

2.2 JML

To be able to formally prove correctness of code, we need a way to precisely define what “correct” means for a given program. The *Java Modeling Language* (JML) is one way to provide such specifications for Java programs. It uses easy-to-understand syntax to describe the intended behaviour in annotations (starting with `/*@` or `//@`), mostly in so-called *method contracts*. The most important features and concepts of JML, with a focus on those used in this work, are described in the following.

```
 1 /*@ public normal_behaviour
 2   @ requires x > 0;
 3   @ ensures \result == 7 * x;
 4   @ measured_by x;
 5   @ assignable \strictly_nothing;
 6   @*/
 7 public int sevenTimes(int x) {
 8     if (x == 1)
 9         return 7;
10     else
11         return sevenTimes(x - 1) + 7;
12 }
```

Listing 2.1: A simple method contract

JML Expressions The expressions allowed in JML specifications can contain standard Java expressions as well as JML specific keywords and symbols. In a method contract, this includes expressions over the method's parameters and the object's fields, a `\result` keyword to access the value returned by the method and an `\old` keyword to refer to expressions in the state before the method call. In addition, side-effect-free, terminating (`/*@ pure @*/`) methods can be invoked.

Furthermore, Boolean expressions can be combined with an implication ==> or equivalence <==> besides the standard Java connectives conjunction &, disjunction | and negation `!`. They can also be quantified with `\forall` and `\exists` quantifiers. For example, to specify that an integer array `a` should only contain positive entries, we could write:

```
(\forall int i; 0 <= i & i < a.length; a[i] > 0)
```

Method Contracts Method contracts are the central concept for modular specification with JML and serve as a contract between the *caller* and the *callee* of the method. They document which preconditions have to hold prior to the method invocation in order for it to guarantee the specified postconditions after its execution. The modularity means that methods can be specified and verified independently of each other.

Listing 2.1 gives an example of a simple contract for a method that uses recursion for multiplying a given `x` by seven. The contract starts with the keyword `normal_behaviour`[2], which specifies that the method has to terminate normally (that is, not due to an exception) and ensure that all postconditions hold in the final state. Afterwards, the content of the contract is given in a number of different *clauses*. Preconditions are Boolean expressions preceded by the keyword `requires` and impose obligations on the caller of the method. In our example, line 2 specifies that the method must be called with a positive argument `x` for the contract to take effect. Postconditions similarly start with the keyword `ensures` and describe the obligations of the callee. In the example, line 3 guarantees that the result equals $7 \cdot x$.

[2] An alternative is `exceptional_behaviour`, which is never used in this thesis and not explained here.

```
class TodoList {
    private String[] items;
    private String nextTask;
    private int nextTaskPos;

    /*@ public instance invariant
            0 <= nextTaskPos && nextTaskPos <= items.length
            && nextTask == items[nextTaskPos];
      @*/
}
```

Listing 2.2: An instance invariant for a simple class `TodoList`

In (mutually) recursive methods, the `measured_by` clause (line 4) is used to specify a termination witness. This could be an integer expression or a pair or tuple thereof, in which case the lexicographical ordering is used for comparisons. In order to prevent circular reasoning, the termination witness needs to be strictly decreasing for called methods and have a lower bound of 0.

Finally, being of great importance in this work and explained in more detail in Section 2.4, there are the `accessible` and `assignable` clauses, both of type `locset` (location set). The framing clause introduced by `assignable` describes which locations on the heap the method may modify at most. The `accessible` clause, in turn, specifies which heap locations the result of a method can at most depend on – the method's *dependency contract*.

Instance Invariants Another central concept of JML are instance invariants (a type of class invariants, also called object invariants) which constrain the valid object state in the form of a Boolean expression. An example can be seen in Listing 2.2. Every constructor has to make sure it establishes these invariants and every method contract implicitly contains them as a pre- as well as a postcondition – except for methods declared as `helper`. The invariant for an object `o` can be explicitly referenced with `\invariant_for(o)`.

Model Methods Model methods[3] are specification-only class members that, similarly to instance invariants, are evaluated with respect to the current object state – they are observer symbols. Just like Java methods, they can take arguments and have an arbitrary return type, but their method body is restricted to a single return statement and must be side-effect free. Their behaviour can be specified with a method contract as described above, only that it starts with the keyword `model_behaviour`. For an example of a model method that appears in this case study, see Listing 2.3.

Ghost Fields Another kind of specification-only class members are ghost fields. They are preceded by the `ghost` keyword and defined just like normal Java fields. In contrast to model methods, they do not depend on the state of the object, but extend it and have to

[3] There are also *model fields*, a closely related concept which is not discussed further here.

```
/*@ model_behaviour
    requires \invariant_for(this);
    accessible \set_minus(footprint(), heightVariant);
    helper model boolean validRBSubtree() {
        return blackBalanced() && noDoubleRed(); // also model methods
    }
  @*/
```

Listing 2.3: The `validRBSubtree` model method in the `Tree` class

be set explicitly by a `set` statement during the execution of a method. Like regular Java fields, they are part of the heap and need to be considered for framing clauses.

Loop Specifications If a method contains a loop, usually an additional specification called a *loop invariant* is necessary for a successful verification. This invariant has to hold before entering the loop and must be maintained by a single iteration of it. As such, it allows reasoning over the loop without knowing the concrete number of executions. If the method is specified to terminate, an additional `decreasing` clause is necessary, describing a variant that decreases with every loop iteration. An `assignable` clause (`\strictly_nothing` if the loop only works on local variables) is used to preserve the knowledge about the heap.

Block Contracts Similarly to loop invariants, block contracts are not strictly necessary, but can facilitate the verification immensely. They allow for the specification of any Java block in the same manner as it is done for methods, which can prevent an unnecessary large number of case distinctions.

Assertions JML assertions are another way to assist the verification process, given as `assert` P`;` in a method body. When we reach this assert statement while proving the method, we have to show that P holds in the current state and can afterwards safely use it for further reasoning. In Section 4.4, we talk about *JML Scripts*, a way to further automise and persist proofs by giving instructions on how to prove each assertion.

2.3 KeY

The KeY system, being developed since 1998, is a formal verification tool for the Java language[4]. It uses *JML** for specifications, a KeY-specific extension of JML that supports all features described above. The properties given in method contracts define proof obligations that are translated into a *sequent*, using *Java Dynamic Logic* (JavaDL) formulas, which then can be proven with KeY in a (semi-)automatic or interactive manner. To handle the semantics of Java programs, KeY uses *symbolic execution*, and *taclets* are used to generally reason about formulas. All this is explained in more detail in the following.

[4] The supported Java features include basic Java 1.2 features such as inheritance, loops, recursion, etc., but KeY does for example not support multithreading or Java 8 features like lambdas.

JavaDL JavaDL, being an instance of dynamic logic, integrates formulas and programs within a single language. The programs must be a sequence of legal Java statements, and Java first-order logic (JFOL) is used for the formulas. JFOL is an extension of classical first-order logic enriched with a type hierarchy – including booleans, integers, class types from the Java code, location sets and heaps – for reasoning about a single state of a Java program. JavaDL formulas can then use the box and diamond modalities, with p being a program and ψ being a JFOL formula: $[p]\psi$ means that if p terminates, ψ has to hold afterwards; $\langle p \rangle \psi$ means that p terminates *and* ψ has to hold afterwards.

To express a `normal_behaviour` method contract this way, we can write $\phi \rightarrow \langle p \rangle \psi$. This means that, starting in a state where the preconditions ϕ hold, the method with body p terminates in a state where the postconditions ψ hold. The formulas ϕ and ψ in reality not only consist of the specified pre- and postconditions, but also some additional fragments taking care of things like framing and recursion or implicit pre- and postconditions.

Sequents and Taclets To prove that a method contract is valid – that a formula like the one above holds, KeY uses a sequent calculus. With this, structured implications called *sequents* are manipulated by a set of schematic rules called *taclets.* A sequent is of the form

$$\phi_1, \ldots, \phi_m \Longrightarrow \psi_1, \ldots, \psi_n$$

with JavaDL formulas ϕ_i (the antecedent) and ψ_j (the succedent) and $0 \leq m, 0 \leq n$. The semantics of the sequent is that under the assumption of all the formulas in the antecedent, one of the formulas in the succedent holds, that is, $\bigwedge_{i=1}^{m} \phi_i \rightarrow \bigvee_{j=1}^{n} \psi_j$ is valid.

The initial sequent for the proof of a method contract always has an empty antecedent and one formula like $\phi \rightarrow \langle p \rangle \psi$ as the succedent – we have to show its validity without any additional assumptions. Starting from that, different rules can be applied to rewrite the sequent, preferably simplifying it step by step and resulting in a proof tree with possibly multiple branches. The impLeft rule for example, read from bottom to top, generates two new branches from a sequent with an implication in the antecedent:

$$\text{impLeft}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \rightarrow \psi \Longrightarrow \Delta}$$

with Γ and Δ being sets of formulas. There are many more such rules, most of them a lot more sophisticated for a multitude of proof situations. Eventually, some special rules can close branches and if all branches of a proof tree are closed, the initial sequent has been successfully proven.

In KeY, every of these schematic rules is defined by a taclet, providing a way to describe the rule itself as well as some additional information like heuristics when it should be applied automatically. New taclets can even be added by the user, in which case their soundness obviously has to be considered carefully.

Symbolic Execution and Updates In order to reason about formulas containing Java statements in modalities, such as the aforementioned $\phi \rightarrow \langle p \rangle \psi$, KeY must somehow know their semantics and interpret them accordingly. This is done by *symbolic execution* taclets, which are applied to the modality and extract or simplify the statements one by one to eventually reach an empty modality that can be eliminated. During this extraction, assignment statements are transformed into parallel *updates*, denoted by curly braces. In the end, these accumulated updates can all be applied simultaneously on the remaining formula. An example of symbolic execution and update simplification could look like this, being simplified a little:

$$\begin{aligned} & \texttt{y} \doteq \texttt{x} \longrightarrow \langle \texttt{y = y + 1;} \rangle \texttt{y} \doteq \texttt{x + 1} \\ \rightsquigarrow\quad & \texttt{y} \doteq \texttt{x} \longrightarrow \{\texttt{y := y + 1}\} \langle\rangle \texttt{y} \doteq \texttt{x + 1} \\ \rightsquigarrow\quad & \texttt{y} \doteq \texttt{x} \longrightarrow \texttt{y + 1} \doteq \texttt{x + 1} \end{aligned}$$

This way, we obtain a formula that now only makes statements about a single state of execution, namely the pre-state of the method. We can then continue reasoning about first-order formulas with only arithmetic, having handled the "dynamic" part with modalities and Java code.

Automated Proof Search and Interaction KeY has an automated proof search strategy that applies taclets on the existing proof branches and tries to close them by some heuristic. How likely which kind of rules are applied can be influenced by the user through proof search strategy settings, which can strongly influence the likelihood of success, as we will see in Section 4.2. There also exist several macros, which a user can select to only perform a specific task automatically, like finishing the symbolic execution. One macro important for us is the Full Auto Pilot, which finishes the symbolic execution, separates the proof obligations, expands invariant definitions and closes all goals that are automatically provable within a given number of steps.

However, KeY's proof search strategy is not always powerful enough, and needs some guidance by the user on how to successfully close a goal. For this, manual interaction with the proof in the KeY GUI is possible: There, we can select a node from the current proof tree and manually apply taclets on formulas present in the corresponding sequent. This might for example be necessary to instantiate quantifiers, expand a needed model method definition, or *cut* the proof, in essence inserting a manual assertion.

2.4 The Frame Problem

In the context of formal methods, the *frame problem* refers to the challenge of specifying what a program does *not* do, specifically which locations on the heap are not changed by a method call. For object-oriented languages, this is challenging because references between objects create complex compound data networks and different references could be aliases to the same object. Thus, we can not assume the effects of a piece of code to be local to some object, which is problematic for modular verification. KeY addresses this problem by introducing *dynamic frames*, abstract sets of heap locations that are "first-class citizens" of the specification language.

For this work, the frame problem manifests as follows: A method called on a tree node may only change heap locations "belonging" to this node and its subtree. It must leave the heap locations of its sibling unchanged. To prove this, we need to know – that is, specify –

```
 1 //@ public instance invariant left == null || right == null ||
       \disjoint(left.footprint(), right.footprint());
 2
 3 /*@ model_behaviour
 4     requires \invariant_for(this);
 5     accessible footprint();
 6     helper model boolean validRBSubtree() { ... }
 7   @*/
 8
 9 /*@ normal_behaviour
10   @ ...
11   @ assignable footprint();
12   @*/
13 public void add(int key) {
14     ...
15     right.add()
16     //@ assert left == null || left.validRBSubtree() ==
           \old(left.validRBSubtree());
17     ...
18 }
```

Listing 2.4: Interplay of `assignable` and `accessible` clauses: Due to the `assignable` clause of `add()` in line 11, we know that `right.add()` in line 15 writes at most to heap locations in `right.footprint()`. Together with the instance invariant, which specifies that `left.footprint()` and `right.footprint()` are disjoint, the assertion in line 16 can now be proven relatively easy by invoking the dependency contract of `left.validRBSubtree()`, which restricts its result to depend on `left.footprint()` at most (see line 5).

that the location sets of the left and the right subtree of a node are always disjoint, which we achieve through the usage of dynamic frames. The desired behaviour of a method can then be proven by using the framing clauses and dependency contracts mentioned in Section 2.2. The `assignable` keyword specifies the *frame* of a method (the set of heap locations it can at most write to) and the `accessible` keyword specifies the *footprint* (the location set the method's result may at most depend on). An example with profitable interplay of these two specification fragments, a simplified version of actual specifications of this work, is shown in Listing 2.4.

There are various alternative approaches to dealing with the frame problem, which are described in more detail in *The KeY Book* (Ahrendt et al., 2016, Section 9.6): Standard JML supports *data groups*, but these are not suitable for modular verification. Another approach is the concept of *ownership*, which introduces a type system with *universe types*. This prevents complex aliasing by restricting the topology of a system to a hierarchical tree structure with at most one *owner* per object – a more scalable but also more restrictive approach compared to dynamic frames. Similarly restricted to hierarchical structures is *separation logic*, an alternative that allows explicit reasoning about the heap by blending separation properties with the functional specification. VerCors is a verification system that uses such separating conjunctions and implications among other constructs, generally making the specification more concise but less straightforward.

2.5 Related Work

Permission-Based Verification of Red-Black Trees and Their Merging Very similar to this work is the case study of Armborst and Huisman (2021), a continuation of the master thesis of Nguyen (2019). They were able to successfully specify and verify a Java implementation of red-black trees, including an insertion and a deletion operation and a parallel merging algorithm for two trees. However, the verification tool used by them is VerCors (Blom et al., 2017), which uses permission-based separation logic and thus a fundamentally different approach to the frame problem described in Section 2.4. Also, some basic design decisions differ from those of our implementation, which are described in Section 3.1. Nonetheless, we were able to adapt many of the specifications concerned with the actual red-black properties and the reasoning over them.

Deletion in a Tree A work that deals with tree structures in KeY is that of Bruns, Mostowski, and Ulbrich (2015). As part of the verification competition held at the Formal Methods 2012 conference, it includes, among other things, a solution to the *Deletion in a Tree* challenge. This challenge asked to specify and verify a method performing the deletion of the minimum element in a binary search tree. We were able to use some specifications from that work as a starting point for the representation of the tree structure and extended them to our needs.

Specification of Red-Black Trees in KeY With the work of Bruns (2011), there exists an attempt on a specification of red-black trees for KeY. However, this was done more than ten years ago, with very limited features of KeY compared to what was developed since, and Bruns concludes that “actual (feasible) verification using KeY is not yet possible”. Although we now were able to verify an insertion operation on red-black trees with KeY, we suspect that the use of parent pointers in Bruns’ implementation still might present a major challenge today due to framing issues (cf. Section 3.1.2). Due to these and other fundamental differences in the implementation, and the incompleteness of the specification, we unfortunately could not build on this work at all.

Other Works on Red-Black Tree Verification Red-black trees are a popular goal for formal verification, as indicated by the significant number of other works that exist in this area. However, these differ more from ours than the ones mentioned above in that they use other programming and specification languages, and other verification tools with different logical foundations. Also, some of them consider non-standard variants of red-black trees.

One such work is that of Peña (2020), where a variant of red-black trees is verified in an assertional proof with the tool *Dafny* (Leino, 2010). Dafny supports object-oriented programs and for such makes use of dynamic frames like KeY– the work of Peña, however, avoids complex memory reasoning altogether by using a purely functional subset of Dafny for its implementation. It instead uses *abstract algebraic data types* (ADTs) to focus on algorithmic correctness.

Other verified functional red-black tree implementations exist in *Coq* (Bertot and Castéran, 2004) by Filliâtre and Letouzey (2004) and, a more efficient version, by Appel (2011), and in Haskell by Kahrs (2001), which also employs ADTs. On an even more abstract level are the works of Nipkow (2016), proving functional correctness (only regarding the search tree invariant) of different functional search trees with *Isabelle/HOL* (Nipkow, Paulson, and Wenzel, 2002), of Manna, Sipma, and Zhang (2007), working with decidable first-order theories of term algebras with Presburger arithmetic, and several others. All these functional works do not have to consider memory issues with pointer structures, in contrast to our work, where we aim to gain insights about exactly these issues by looking at a more practical object-oriented implementation.

One work that aims to bridge the gap between such verifications working with abstract data types, and real-world implementations, is that of Schellhorn et al. (2022a). It uses data refinement to consider functional correctness on an algebraic level separate from reasoning about pointer structures, using the theorem prover *KIV* (Schellhorn et al., 2022b). Like the work in VerCors described above, this is based on separation logic, while our work, to the best of our knowledge, is the first to verify red-black trees with a tool that uses dynamic frames for memory reasoning.

3 Implementation and Specification

For this thesis, the overall goal is to define and prove method contracts for the `contains` and `add` methods of a Java implementation of red-black trees. In this chapter, we look at our implementation of these and other necessary methods, and how they are specified. The complete source code without any specification can be found in the appendix, Section A.1; code including specifications can be found in the corresponding subsections of this chapter and on GitHub[1].

3.1 Design Decisions

This section describes some fundamental design decisions we made for our implementation of red-black trees, and how these differ from the JCL implementation and those of related works. These decisions were taken in order to make the specification and verification feasible, for example regarding framing aspects. It is important to note that, despite these differences, the core algorithm for rebalancing the tree remains the same.

3.1.1 General Simplifications

Compared to the `java.util.TreeMap` in the JCL, we made a number of simplifications. Firstly, due to the limited time available, we naturally couldn't look at all the numerous methods provided by this class and concentrate on one simple constructor, a `contains` method checking the presence of a given element and an `add` method inserting a new one.

Also, while JCL's `TreeMap<K,V>` can store generic keys and values, we only allow Java's primitive `int` for keys. Allowing arbitrary objects as keys would require a lot of additional reasoning regarding framing, because these objects are part of the heap and potentially hold references to other objects. The restriction to integers also frees us from having to deal with custom `Comparators` – whose treatment in KeY would require some fundamental considerations. We further do not consider any values at all so that the data in a node consists only of the key. Due to them not interacting with the tree structure in any way, we do not deem them necessary for a formal verification, but merely distracting.

Some additional decisions consistent with the JCL but different from the implementation of Bruns (2011) are the uniqueness of keys in the tree, the use of `null` instead of special `NIL` nodes and the absence of default values.

[1] `https://github.com/gewitternacht/rbtree-verification`

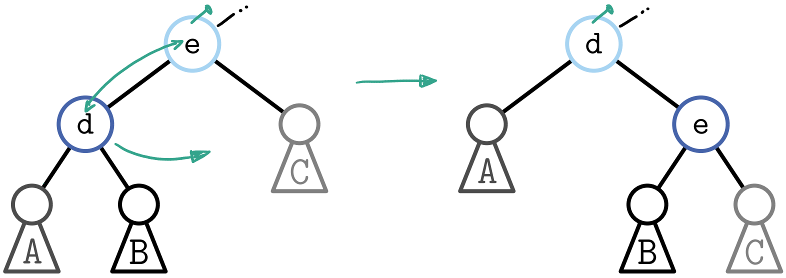


Figure 3.1: Our version of a right rotation (see Figure 2.2 for the standard version): As indicated by the colours that represent the object identities, the parent of the light blue node does not have to change the reference to its child. The rotation is realised by swapping the values d and e and moving the dark blue node from the left to the right. The pointers to the subtrees A, B and C have to be modified accordingly.

3.1.2 Iterative vs. Recursive Add Method

One major difference of our add method to the corresponding one in `java.util.TreeMap` is the mechanism by which the tree is traversed upwards during the rebalancing operations. The implementation of the JCL, a mostly exact adaption of the pseudocode in Leiserson et al. (1994), uses a loop to iteratively follow parent pointers up the tree. However, we suspect this to impose major challenges on the handling of framing, as these parent pointers present circular references. It would not be possible for all objects (indirectly) referenced in a node to “belong” to this node, and the interplay of dependency contracts and framing clauses would not work out as nicely as it does without parent pointers.

Hence, we decided for an equivalent recursive version where the rebalancing operations are performed while returning from recursive calls, so no references to parent nodes are needed. Since the tree height always stays comparatively small, this should not cause any problems with recursion depth.

3.1.3 Preservation of the Root Node by Rotations

The last adjustment made by us is one in the implementation of tree rotations (cf. Section 2.1.3). Normally, by performing a rotation on a subtree, the root node of this subtree changes. The parent of the subtree therefore needs to adapt its corresponding child reference to point to a different object.

However, we found a way[2] to preserve the root node of the subtree during a rotation by swapping the keys of the two “rotated” nodes and only changing some references below the root node. Figure 3.1 illustrates this process. Our approach makes the rotations self-contained, which facilitates the reasoning regarding framing and also avoids the need for a return value of the rotation method to inform callers about the changed root. This avoidance of return values is what distinguishes our implementation from that of Armborst and Huisman (2021), where they also opted for a recursive insertion, but used “standard” rotations, resulting in return values being necessary for all methods.

[2] inspired by the use of swap operations in a Rust implementation: `https://github.com/dbyr/rb_tree`

```
public class Client {
    public static void main(String[] args) {
        RBTree t = new RBTree();

        /*      -3      -3             5              5
                          \           / \            / \
                    ~>     5   ~>   -3   9    ~>   -3   9
                                                       /
                                                      7         */
        t.add(-3);
        t.add(5);
        t.add(9);   // rotation
        t.add(7);   // recolouring

        /*@ assert (\forall int k; t.contains(k)
                        <==> (k == -3 || k == 5 || k == 9 || k == 7)); *@/
        //@ assert t.validRBTree();
    }
}
```

Listing 3.1: The `Client` class that uses an `RBTree`

3.2 Classes

With this section, we begin looking at the concrete code of our implementation, starting with the basic structure. We use three classes, two of them for the actual red-black tree implementation, and one client class.

3.2.1 `Client`

The `Client` class is a simple example of a client that uses our red-black tree implementation. It contains a main method that creates an `RBTree` and calls `add` a few times. The purpose of this class is to make sure that the implementation is usable in practice and that our method contracts are strong enough to show desirable features from a client's point of view. In our case, the overall goal is to provide the client with contracts of `add` and `contains` that together can ensure the following: A red-black tree consists of exactly those elements that were inserted before, and it fulfils the binary-search-tree and red-black properties from Section 2.1.

The two assertions shown in Listing 3.1 can be considered to be the actual specification of the client, ensuring that the aforementioned properties can be shown. To be able to load the main method into KeY, we additionally annotate it with an "empty" contract that requires and ensures `true`.

```java
final public class RBTree {
    /*@ nullable @*/ Tree root;

    public boolean contains(int key) {
        return root != null && root.contains(key);
    }

    public void add(int key) {
        if (root == null) {
            root = new Tree(key);
        } else {
            root.add(key);
        }
        fixRootColour();
    }

    private void fixRootColour() {
        root.isRed = false;
    }
}
```

Listing 3.2: The `RBTree` class

3.2.2 RBTree

The `RBTree` class provides the interface to create empty red-black trees, add elements to existing ones, and query them for specific keys. It consists of just a few lines of code, which are shown in Listing 3.2. Being essentially a wrapper for the `Tree` class, it only has one field, the `root` of the red-black tree, which is of type `Tree`. This field needs to be annotated with the JML keyword `nullable`, since JML assumes all fields to be non-nullable by default, but the `root` might be `null` if the tree is empty. The `RBTree` class is necessary in addition to `Tree` to handle this special case for the root. It also makes sure that the colour of `root` is always black, as required by the second red-black property listed in Section 2.1.2.

The specification of `RBTree`'s methods is postponed to Section 3.4.2, until after we have seen some basic definitions of used model methods.

3.2.3 Tree

An object of the `Tree` class corresponds to a node in a red-black tree.[3] The fields of a `Tree` consist of its `left` and `right` child (also of type `Tree`), an integer `key` and a boolean `isRed`, as shown in Listing 3.3. Missing children are represented as `null` and therefore, the `nullable` annotation is necessary for `left` and `right`. The `isRed` field stores the node's colour, with the value `true` representing red and `false` black.

[3] `Node` might have been a better name, but we stuck with `Tree` for "historical reasons".

```java
final public class Tree {
    /*@ nullable @*/ Tree left;
    /*@ nullable @*/ Tree right;
    int key;
    boolean isRed;

    public Tree(int key) { ... }
    public boolean contains(int key) { ... }
    public void add(int key) { ... }

    private void rightRotate() { ... }
    ...
}
```

Listing 3.3: The `Tree` class with its fields and most important methods

`Tree` implements the `contains` and `add` methods we already encountered in Listing 3.2 and additional internal methods, for example to handle recolouring or rotations. All methods and their specifications are explained in detail in Section 3.4.

3.3 Modelling the Tree Structure and Properties

This section deals with the fundamental definitions for the representation of the structure and properties of a red-black tree. These definitions abstract from the concrete data structure and allow us to reason about the properties of a tree more comfortably[4]. We use them in method contracts, for example to talk about "the set of all keys contained in the tree" or to ensure that the binary-search-tree or red-black properties hold.

As the `Tree` class is where "the magic happens" (where the algorithms are actually implemented), the things described below are all part of `Tree`. The last subsection then shows how the `RBTree` lifts these definitions, similarly to the normal Java methods, up to the top level view of complete red-black trees.

The definitions concerned with the tree structure – `footprint` and `treeSet` – are based on the work of Bruns, Mostowski, and Ulbrich (2015), though modified and expanded to our needs. For the definitions of the red-black properties, we were able to adapt a lot from the VerCors specification of red-black trees by Armborst and Huisman (2021).

3.3.1 Recursion Measure `heightVariant`

As the `add` method as well as most of the model methods described below are defined recursively, they need to specify a decreasing variant in a `measured_by` clause (see Section 2.2). The height of a subtree is the obvious choice for a value decreasing towards children: It is zero for leaves and increases by at least one for each step upwards to the root node.

[4] Frankly, "comfortable" here means that these abstractions make the verification feasible at all.

Thus, we introduce an integer ghost field `heightVariant`, that extends the object state and keeps track of an overapproximation of a subtree's height:

```
//@ ghost instance int heightVariant;
```

Always keeping track of the actual height would have been unnecessarily complex, as this field needs to be set explicitly during the execution of an insertion or rotation. How this is done is explained in the detailed descriptions of the according methods in Section 3.4. The properties we require of the `heightVariant` are captured in an instance invariant, shown in Listing 3.8, line 3, 4 and 7.

3.3.2 Framing Specification `footprint`

Being the very central concept of this work, the definition of a node's `footprint` might be surprisingly simple. It is a model method specifying all heap locations belonging to the node, and thus is of the built-in type `\locset`. It consists of the fields of the node itself, denoted by `this.*`, and recursively the `footprints` of its children:

```
/*@ helper model \locset footprint() {
        return \set_union( this.*,
               \set_union( left  == null ? \empty: left.footprint(),
                           right == null ? \empty: right.footprint()));
    }
  @*/
```

Listing 3.4: Definition of a `Tree`'s `footprint`

Like Section 2.4 explains, we need this `footprint` for framing purposes, to reason that operations on subtrees only make local changes. For this, we define the disjointness of `footprints` in an instance invariant, see Listing 3.8, line 5, 8 and 12. It generally ensures the well-formedness of the subtree by also requiring that the left and right child of the node must not be equal (line 2).

3.3.3 Abstract Tree Representation `treeSet`

```
/*@ helper model \free treeSet() {
      return \dl_iSet_union( left  == null ? \dl_iSet_empty() : left.treeSet(),
             \dl_iSet_union( \dl_iSet_singleton(this.key),
                             right == null ? \dl_iSet_empty() : right.treeSet()));
  }
@*/
```

Listing 3.5: Definition of a `Tree`'s `treeSet`

The set of all keys contained in the tree is defined by the model method `treeSet`. Syntactically, the method has the return type `\free`, which is a built-in "customisable" data type. In our case, we define a type `iSet` for integer sets that was already used in previous works and the source code of which can be found in the appendix in Section A.2. We refrain from using the built-in \seq type used by Bruns, Mostowski, and Ulbrich (2015) for a similar method because we want to abstract from the concrete place of an element in the tree.

The definition of treeSet is similar to that of footprint in consisting of the key of the node itself and recursively the treeSets of its children.

3.3.4 Binary-Search-Tree Property

There are different ways to formulate the binary-search-tree property given in Section 2.1.1. Depending on the specific proof situation, either of them might be more suited for easy reasoning. This is why we define two versions, both as model methods of a node that take an integer key k as a parameter and return a boolean:

```
/*@ helper model boolean invLessNotInRight(int k) {
        return k < key ==> (right == null || !\dl_in(k, right.treeSet()));
    }
  @*/

/*@ helper model boolean invLessInTreeIffLeft(int k) {
        return k < key ==> (left == null ||
                (\dl_in(k, treeSet()) <==> \dl_in(k, left.treeSet())));
    }
  @*/
```

Listing 3.6: Definition of the binary-search-tree property

The first one, invLessNotInRight, specifies that a k smaller than the node's key cannot be contained in the right subtree. The second one, invLessInTreeIffLeft, specifies that such a key is contained in the treeSet of the node if and only if it is contained in the treeSet of the left subtree. Both methods exist symmetrically for the case k > key (called invGreaterNotInLeft and invGreaterInTreeIffRight).

These methods are then universally quantified to fully express the binary-search-tree property. However, only the first versions are used in an instance invariant, as can be seen in Listing 3.8, line 10 and 11. We do this to avoid the extra effort of always having to prove both versions to hold whenever we have to show the preservation of the instance invariant. Instead, to be able to use the second versions if needed, we make use of model methods again and define a lemma invLemmaLess:

```
/*@ model_behavior
    requires (\forall int k; invLessNotInRight(k));
    ensures (\forall int k; invLessInTreeIffLeft(k));
    helper model boolean invLemmaLess() {
        return true;
    }
  @*/
```

Listing 3.7: Modelling of a lemma for the binary-search-tree property

If invLessNotInRight holds for all k, then – under the implicit assumption of the complete instance invariant, including the disjointness of the footprints of left and right – the invLessInTreeIffLeft version also holds for all k. Again, this is symmetrically defined for the case k > key. The contracts of these "lemma model methods" can then be proven individually and used in another proof as described in Section 4.3.

3.3.5 Instance Invariant

```
 1  /*@ public instance invariant
 2            (left != right || left == null || right == null)
 3         && 0 < heightVariant
 4         && (left  == null || ( left.heightVariant < heightVariant
 5                             && \disjoint(this.*, left.footprint())
 6                             && \invariant_for(left)))
 7         && (right == null || ( right.heightVariant < heightVariant
 8                             && \disjoint(this.*, right.footprint())
 9                             && \invariant_for(right)))
10         && (\forall int k; invLessNotInRight(k))
11         && (\forall int k; invGreaterNotInLeft(k))
12         && (left == null || right == null || \disjoint(left.footprint(),
               right.footprint()));
13    @*/
```

Listing 3.8: The instance invariant of Tree

All the invariants mentioned above are summarised in a single instance invariant, as shown in Listing 3.8, together with one important addition in line 6 and 9: The complete invariant, that is, decreasing of the heightVariant, well-formedness and binary-search-tree property, has to hold recursively for the node's children. This makes sure that a node is only in a valid state if both its children are too.

This instance invariant is virtually never violated in any intermediate state, and implicitly is part of the pre- and postcondition of every method contract (see Section 2.2). Separate from this, the red-black properties are summarised in their own "invariant" that is explained below. This is because these properties may be temporarily violated during the insertion, and separating them from the really invariant properties allows us to precisely define what violations are possible in which state.

3.3.6 Red-Black Properties

Several further model methods specify the *black balanced* and *no double red* properties from Section 2.1.2.

Black Height The black height of a node is determined by a static model method taking a nullable Tree t as its parameter and returning an integer. It is not an instance method because that would require lengthy case distinctions during the calculation, checking if one or both of the children are null. With the static version and in the case that t is non-null, its black height is always calculated as the maximum black height of its children, increased by one if `t` is black. Evaluating `blackHeight` for a `null` child is allowed and defined as `1` – remember that leaf nodes are always black.

Although this static implementation saves some case distinctions, we refrain from using it for the other model methods below due to a bug in key concerning `nullable` arguments of model methods (see Section 5.3).

```
/*@ helper model static int blackHeight(nullable Tree t) {
        return t == null ? 1
                         : (t.isRed ? 0 : 1)
                            + (blackHeight(t.left) > blackHeight(t.right)
                            ? blackHeight(t.left)
                            : blackHeight(t.right));
    }
  @*/
```

Listing 3.9: Definition of a `Tree`'s `blackHeight`

Black Balanced The `blackBalanced` model method is rather simple: It is an instance method, takes no arguments and returns a boolean – like all model methods that follow below. It returns `true` if the black height of its children are equal and they recursively are black balanced themselves (if existing), and `false` otherwise.

No Double Red Similarly, the `noDoubleRed` method is not complicated either: For this method to return `true`, both the left and the right child (if existing) must not be red if the node itself is red, and they have to fulfil the `noDoubleRed` property themselves.

However, as this is the property that might be violated during an insertion, we also define alternative `doubleRedLeft` and `doubleRedRight` methods that describe the specific violations that might occur:

```
/*@ helper model boolean doubleRedLeft() {
        return isRed
            && left != null && left.isRed && left.noDoubleRed()
            && (right != null ==>
                (!right.isRed && right.noDoubleRed()));
    }
  @*/
```

Listing 3.10: Definition of the exceptionally allowed `doubleRedLeft`

This method returns `true` if the node and its left child are both red while all other nodes continue to fulfil the `noDoubleRed` property; `doubleRedRight` is defined symmetrically. Additionally, `doubleRedTop` summarises the two previous violations by being `true` if either of them is `true`.

(Almost) Valid Red-Black Subtree Finally, the `validRBSubtree` model method (which is shown in Listing 2.3) is the conjunction of `blackBalanced` and `noDoubleRed`, describing a

```
/*@ model_behaviour
    requires \invariant_for(root);
    accessible footprint();
    helper model boolean validRBTree() {
        return root == null || (
              \disjoint(this.*, root.footprint())
            && \invariant_for(root)
            && root.validRBSubtree()
            && !root.isRed );
    }
@*/

/*@ helper model \locset footprint() {
        return \set_union(this.*, (root == null) ? \empty : root.footprint());
    }
  @*/

/*@ helper model \free treeSet() {
        return (root == null) ? \dl_iSet_empty(): root.treeSet();
    }
  @*/
```

Listing 3.11: The model methods of `RBTree`

fully valid red-black subtree regarding the constraints on its colours. As mentioned earlier in Section 3.3.5, this method can be seen as an additional invariant that can be weakened in certain situations. For this, `validRBSubtreeExceptRedTop` is defined as the subtree either being completely valid or having the one violation specified through `doubleRedTop`.

3.3.7 `RBTree` Model Methods

Similarly to the normal Java methods, `RBTree` lifts the model methods of the `Tree` class to the level of a complete red-black Tree, as shown in Listing 3.11. From this top-level view, a valid red-black tree must always fulfil the `validRBTree` model method: The root must satisfy its invariant and be a `validRBSubtree` and, in accordance with the second property in Section 2.1.2, it must be black. Also, `this.*` must be disjoint from the root's `footprint`.

The `footprint` and `treeSet` of `RBTree` are wrappers for those of `Tree` with special handling for the case `root = null`.

3.4 Methods and their Contracts

This section describes the code and contracts of all methods. Section 3.4.1 starts with the contracts for the model methods, of which the definitions were already given in Section 3.3. Next, Section 3.4.2 deals with the methods of `RBTree`. Each following section then explains one of the more complex methods of `Tree`.

```
/*@ model_behavior
    requires \invariant_for(this);
    accessible \set_minus(footprint(),
                    \set_union(isRed,
                               (left == null? \empty: \singleton(left.isRed)),
                               (right == null? \empty: \singleton(right.isRed)),
                               heightVariant));
    measured_by heightVariant;
    helper model \free treeSet() { ... }
  @*/
```

Listing 3.12: Contract of the `treeSet` model method in `Tree`

3.4.1 Contracts for Model Methods

The instance invariant, the `footprint` and `treeSet` model methods, and those for the *black balanced* and *no double red* properties in the `Tree` class all have approximately the same contract: An `accessible` clause restricts the result to the `footprint` of the node – depending on the specific method, this is further refined, for example by excluding the colour of the node and other fields for `treeSet` (see Listing 3.12). This makes it as comfortable as possible to reason about model methods' results not changing through specific method calls and assignments. Because the methods are defined recursively, a `measured_by` clause specifies the `heightVariant` as the recursion measure. The `\invariant_for(this)` is required because only then is it guaranteed that the `heightVariant` is in fact decreasing towards children nodes.

The `validRBSubtree(ExcpetRedTop)` model methods have similar contracts as well, but are not recursive and therefore do not need a `measured_by` clause. In `RBTree`, each model method may access the tree's complete `footprint` and requires the `\invariant_for(root)` for this.

3.4.2 `RBTree` Method Contracts

The method contracts of the `RBTree` methods are what is visible from an outside perspective like our `Client` (see Section 3.2.1). As they are only called on fully valid red-black trees and not in any intermediate state, they are relatively simple. For the implementation of the methods, see Listing 3.2. The specification is given in Listing 3.13.

Constructor `RBTree`'s constructor is simply a default constructor, taking no arguments and creating an empty red-black tree with a `null` root (see line 6 in Listing 3.13). It is made explicit in order to be able to specify a method contract. According to this contract, the constructor has to ensure that the resulting red-black tree is valid and empty, and consists only of newly allocated heap locations (denoted by the `fresh` keyword).

`RBTree::contains` The `contains` method takes an integer `key` as an argument and forwards the query to the root node if it is currently not `null`. It may not modify the heap

```
/*@ public normal_behavior
  @ ensures validRBTree();
  @ ensures treeSet() == \dl_iSet_empty();
  @ ensures \fresh(footprint());
  @*/
public RBTree() {}

/*@ normal_behavior
  @ requires validRBTree();
  @ ensures validRBTree();
  @ ensures \dl_in(key, treeSet()) <==> \result == true;
  @ accessible footprint();
  @ assignable \strictly_nothing;
  @*/
public boolean contains(int key) { ... }

/*@ normal_behavior
  @ requires validRBTree();
  @ ensures validRBTree();
  @ ensures treeSet() == \dl_iSet_union(\old(treeSet()), \dl_iSet_singleton(key));
  @ ensures \new_elems_fresh(footprint());
  @ assignable footprint();
  @*/
public void add(int key) { ... }

/*@ normal_behavior
  @ requires root != null;
  @ requires \invariant_for(root);
  @ requires root.validRBSubtreeExceptRedTop();
  @ ensures validRBTree();
  @ ensures treeSet() == \old(treeSet());
  @ ensures footprint() == \old(footprint());
  @ assignable root.isRed;
  @*/
private void fixRootColour() {
    root.isRed = false;
}
```

Listing 3.13: Mehtod contracts for RBTree

at all, and its result may only depend on the heap locations contained in the `footprint` of the red-black tree. Said result must be true if and only if the queried `key` is part of the tree's `treeSet`. The `validRBTree` model method is added to the contract as an "invariant".

`RBTree::add` The `add` method also wraps that of `Tree` and afterwards makes sure the root colour is always black by calling `fixRootColour()`. It may write to heap locations in the `footprint` and, denoted by `\new_elems_fresh(footprint())`, extend it by newly allocated locations. As expected of an `add` method, the given `key` should be added to the existing `treeSet`, which otherwise should remain the same. The `validRBTree` invariant is added here, too.

`fixRootColour` The root node potentially violates the *no double red* property with one of its children after calling `Tree::add` and thus satisfies `validRBSubtreeExceptRedTop`. A `validRBTree` can be restored by simply colouring the root node black, which `fixRootColour` does. This method in addition requires the invariant of the non-null `root` to hold and ensures that nothing besides the root colour changes.

Together, the `ensures` clauses talking about the `treeSet` (line 11 and 20 of Listing 3.13) make sure that a client can prove a red-black tree to contain exactly those elements that were previously added. It is also ensured that the tree is in a valid state at all times. Additionally, the `accessible` clauses of `contains` and of the model methods, together with the `fresh` clauses regarding the `footprint`, allow a client to create multiple red-black trees and know that they do not influence each other.

3.4.3 `Tree::contains`

The contract of the `contains` method in `Tree` (see Listing 3.14) is almost exactly the same as that of its `RBTree` counterpart, only omitting the `validRBTree` invariant. The instance invariant that is implicitly part of the contract is all the method needs to work correctly.

The algorithm described in Section 2.1.1 is implemented with a `while` loop, traversing the tree downwards until the given `key` or a leaf is reached. For this loop, we provide a loop invariant to guide the prover: While the current `node` is not `null`, its invariant holds and the `key` is part of the corresponding subtree if and only if it also is part of the whole tree – we successively narrow down its possible position in the tree. If we reach `null`, the `key` is not contained in the tree at all. The `decreasing` clause specifies the `node`'s `heightVariant` as the loop variant.

3.4.4 `Tree::add`

The `add` method and its contract are a lot more complex than its `RBTree` wrapper and the part of this work that was the most time-consuming to prove. Because all the fix operations described in Section 2.1.3 need to be done symmetrically for a "left" and a "right" case, we split it up into two methods `addLeft` and `addRight`, as can be seen in Listing 3.15. This introduces more modularity, which has the practical advantages of decreased proof size

```
/*@ normal_behavior
  @ requires true;
  @ ensures \dl_in(key, treeSet()) <==> \result == true;
  @ accessible footprint();
  @ assignable \strictly_nothing;
  @*/
public boolean contains(int key) {
    Tree node = this;

    /*@ maintaining node == null ==> !\dl_in(key, treeSet());
      @ maintaining node != null ==> (\invariant_for(node) &&
      @             (\dl_in(key, treeSet()) <==> \dl_in(key, node.treeSet())));
      @ decreasing node == null ? 0 : node.heightVariant;
      @ assignable \strictly_nothing;
      @*/
    while (node != null && node.key != key) {
        if (key < node.key) {
            node = node.left;
        } else {
            node = node.right;
        }
    }
    return node != null;
}
```

Listing 3.14: Tree's contains with its contract

```
/*@ normal_behavior
@ ... same requires and ensures clauses as addLeft and addRight ...
@ measured_by heightVariant, 1;
@*/
public void add(int key) {
  if (key == this.key) {
      return;
  } else if (key < this.key) {
      addLeft(key);
  } else {
      addRight(key);
  }
}
```

Listing 3.15: Tree's add with its contract

with greater clarity and easier redoing of proofs. Also, due to the two methods being completely symmetric, once a proof is found for one of them, the other can be proven analogously. In this work, we actually only prove addRight, but we are confident that the addLeft proof would be very doable, though laborious, with all the gained knowledge.

Therefore, we will now look at the addRight method and its contract in more detail, which is shown in Listing 3.16. It has the exact same requires and ensures clauses as add, with the additional precondition that the key to be inserted must be greater than the node's key. The only other explicit requirement is that the method must be called on a validRBSubtree. Then, the contract ensures that the resulting tree is a validRBSubtreeExceptRedTop, maintains its blackHeight, and adds the given key to the treeSet, while only changing heap locations in the footprint or newly allocating some. The measured_by clause of this method is (heightVariant, 0), a pair of integers. This is because it is in mutual recursion with add, which does not descend to a child node but only decreases the second integer from 1 to 0.

In the case that there currently is no right child, a new node is created and added in its place, the heightVariant of the node is increased by one, and we return. Otherwise, right.add is called recursively. As this could increase the heightVariant of right to a value greater than that of the node itself, setHeight (see Section 3.4.7) is called afterwards to restore the instance invariant.

Then, the fix operations illustrated in Figure 2.3 are executed, but only at a level "above" the actual *no double red* violation: If the current node is red, we immediately return and let the parent node (or the RBTree wrapper) do the work. If the current node is black and its right child is red, the violation could occur with right.left or right.right and the appropriate measure (recolouring or one to two rotations) is taken. It is not possible that this, right and a grandchild are all red at the same time, which the "helper" clause in line 6f. ensures – either the colour of right did not change during the recursive call (in which case this and right can't be both red), or the colour changed to red, but then both grandchildren are known to be black. After the first normalisation rotation, setHeight is called again to restore the invariant, as rotations can increase the heightVariant of a tree.

A remark about the checks for different colour combinations in the if statements: We use non-short-circuiting logical connectives here, which might make the implementation a little less efficient. However, this facilitates the proof of the method a great deal, because short-circuiting connectives introduce additional proof branches we don't need here. The part of the checks that *needs* short-circuiting is separated into the isRed method (see Section 3.4.5) for this reason.

3.4.5 isRed

The static isRed method is used for case distinctions during addRight (see Listing 3.16). It takes a nullable Tree t as its parameter and summarises two checks: As the null leaves of a tree are regarded to be black, it returns true if and only if t is non-null and red. The contract is a copy of the statement in the method body, with canonical accessible and assignable clauses, see Listing 3.17.

```
/*@ normal_behavior
  @ requires key > this.key;
  @ requires validRBSubtree();
  @ ensures validRBSubtreeExceptRedTop();
  @ ensures blackHeight(this) == \old(blackHeight(this));
  @ ensures isRed == \old(isRed)
  @         || (isRed && !isRed(left) && !isRed(right));
  @ ensures treeSet() == \dl_iSet_union(\old(treeSet()), \dl_iSet_singleton(key));
  @ ensures \new_elems_fresh(footprint());
  @ assignable footprint();
  @ measured_by heightVariant, 0;
  @*/
private void addRight(int key) {
    if (this.right == null) {
        Tree newRight = new Tree(key);  // separated creation and assignment of new
        this.right = newRight;          // node for easier handling of assertions
        //@ set heightVariant = heightVariant + 1;

    } else {
        this.right.add(key);
        setHeight();

        // ----------- fix operations -----------
        if (!isRed & right.isRed) {

            // recolouring if uncle red -> potentially moves double red up the tree
            if (isRed(left) & (isRed(right.left) | isRed(right.right))) {
                recolour();
                return;
            }

            // rotations if uncle black -> fixes double red completely
            // rotation for normalisation (make double red "outer")
            if (isRed(right.left)) {
                right.rightRotate();
                setHeight();
            }
            // rotation to fix double red
            if (isRed(right.right)) {
                leftRotate();
                return; // return statement for easier handling of assertions
            }
        }
    }
}
```

Listing 3.16: addRight with its contract

```
/*@ normal_behaviour
  @ ensures \result == (t != null && t.isRed);
  @ accessible t == null ? \empty : t.footprint();
  @ assignable \strictly_nothing;
  @*/
private static boolean isRed(/*@ nullable @*/ Tree t) {
    return t != null && t.isRed;
}
```

Listing 3.17: `isRed` with its contract

3.4.6 `recolour`

The `recolour` method is straightforward, both regarding its implementation and specification, as shown in Listing 3.18: The node itself has to be black and `blackBalanced`, each child has to be red and a `validRBSubtreeExceptRedTop`, and one grandchild must violate the *no double red* property. Under these circumstances, the method guarantees a `validRBSubtree` with a red root and black children, not changing the `blackHeight`, `treeSet` or `footprint`.

3.4.7 `setHeight`

Although the `setHeight` method is present in the normal Java code, it exists only for specification purposes. As already seen in Listing 3.16, it is called twice in `addRight` (and `addLeft`) to restore the invariant regarding the `heightVariant`. Like shown in Listing 3.19, `setHeight` realises this by setting the `heightVariant` of the node to the maximum of those of its children, increased by one.

There are two reasons for factoring this out in its own method: Firstly, the calculation includes several case distinctions – implemented with ternary operators here – for which the symbolic execution during a proof introduces several new branches. However, the only thing we really need to know is that the invariant is restored, which is equally provided by all branches. The method contract of `setHeight` specifies this (line 2-6)[5], summarising all branches in a single one from a caller's perspective. Secondly, we address a framing issue: Though it may be obvious to humans, KeY needs to somehow realise that a change of the `heightVariant` has no impact on the results of all our model methods. This is what the rest of the contract specifies, facilitating the proof of methods that call `setHeight` further.

Because the aforementioned multiple branches would now be an issue for proving all the `ensures` clauses regarding results of model methods, we additionally enclose the three lines of the method in a block contract, essentially repeating line 2-6.

3.4.8 Rotations

There are two rotation methods, for a left and a right rotation, of which we also only verified the `rightRotate` version shown in Listing 3.20. As described in Section 3.1.3, this

[5] Also note the `helper` annotation, which prevents the invariant from being implicitly part of the contract.

```
/*@ normal_behaviour
  @ requires right != null && left != null;
  @ requires left.validRBSubtreeExceptRedTop() &&
      right.validRBSubtreeExceptRedTop();
  @ requires blackBalanced();
  @ requires !isRed && left.isRed && right.isRed &&
  @             (isRed(left.left) || isRed(left.right) || isRed(right.left) ||
      isRed(right.right));
  @ ensures validRBSubtree();
  @ ensures isRed && !isRed(left) && !isRed(right);
  @ ensures blackHeight(this) == \old(blackHeight(this));
  @ ensures treeSet() == \old(treeSet());
  @ ensures footprint() == \old(footprint());
  @ assignable isRed, left.isRed, right.isRed;
  @*/
private void recolour() {
    isRed = true;
    left.isRed = false;
    right.isRed = false;
}
```

Listing 3.18: `recolour` wit its contract

implementation preserves the root node of the subtree by reattaching some nodes beneath it and swapping keys. In addition, we need to update the `heightVariant` of the two rotated nodes, increasing that of the root by one.

Besides `left` being non-null, the method contract only requires `blackBalanced` to hold, as rotations are performed in an intermediate state where `noDoubleRed` is violated. However, one of two conditions specified in line 4f. must be met, corresponding to the two specific situations in which a rotation might be performed. Depending on which of these actually holds, a different property can be guaranteed afterwards: The first option is that the rotation serves as a normalisation from an “inner” to an “outer” double red – `doubleRedLeft` to `doubleRedRight` – during the `addRight` method (see line 13). The alternative is that the rotation completely fixes the violation during an `addLeft` call, resulting in `noDoubleRed` (see line 11f.).

The `blackBalanced` property as well as the colour, `blackHeight`, `footprint` and `treeSet` of the node are always preserved. Furthermore, the following is ensured: `right` is now non-null and has the former colour of `left`, the `heightVariant` is increased by at most one and only heap locations contained in the `footprint` may be assigned.

```
/*@ normal_behaviour
  @ requires ... instance invariant except for part about heightVariant ...
  @ ensures heightVariant > 0;
  @ ensures left == null || heightVariant > left.heightVariant;
  @ ensures right == null || heightVariant > right.heightVariant;
  @ ensures \invariant_for(this);
  @ ensures footprint() == \old(footprint());
  @ ensures treeSet() == \old(treeSet());
  @ ensures blackHeight(this) == \old(blackHeight(this));
  @ ensures blackBalanced() == \old(blackBalanced());
  @ ensures left == null || left.noDoubleRed() == \old(left.noDoubleRed());
  @ ensures right == null || right.noDoubleRed() == \old(right.noDoubleRed());
  @ ensures noDoubleRed() == \old(noDoubleRed());
  @ ensures doubleRedTop() == \old(doubleRedTop());
  @ ensures left == null || left.validRBSubtreeExceptRedTop() ==
      \old(left.validRBSubtreeExceptRedTop());
  @ ensures right == null || right.validRBSubtreeExceptRedTop() ==
      \old(right.validRBSubtreeExceptRedTop());
  @ ensures validRBSubtreeExceptRedTop() == \old(validRBSubtreeExceptRedTop());
  @ ensures validRBSubtree() == \old(validRBSubtree());
  @ assignable heightVariant;
  @ helper
  @*/
private void setHeight() {

    /*@ requires left == null || (\disjoint(this.*, left.footprint()) &&
        \invariant_for(left));
      @ requires right == null || (\disjoint(this.*, right.footprint()) &&
          \invariant_for(right));
      @ ensures heightVariant > 0;
      @ ensures left == null || heightVariant > left.heightVariant;
      @ ensures right == null || heightVariant > right.heightVariant;
      @ signals_only \nothing;
      @ assignable heightVariant;
      @*/
    {
        //@ ghost int leftHeight = left == null ? 0 : left.heightVariant;
        //@ ghost int rightHeight = right == null ? 0 : right.heightVariant;
        //@ set heightVariant = 1 + (leftHeight > rightHeight ? leftHeight :
            rightHeight);
    }
}
```

Listing 3.19: `setHeight` with its contract

```
/*@ normal_behavior
  @ requires left != null;
  @ requires blackBalanced();
  @ requires !isRed && left.doubleRedLeft() && (right == null || !right.isRed &&
      right.noDoubleRed())
  @          || doubleRedLeft();
  @ ensures footprint() == \old(footprint());
  @ ensures treeSet() == \old(treeSet());
  @ ensures heightVariant <= \old(heightVariant) + 1;
  @ ensures isRed == \old(isRed);
  @ ensures right != null && right.isRed == \old(left.isRed);
  @ ensures \old(!isRed && left.doubleRedLeft() && (right == null || !right.isRed
      && right.noDoubleRed()))
  @             ==> noDoubleRed();
  @ ensures \old(doubleRedLeft()) ==> doubleRedRight();
  @ ensures blackBalanced();
  @ ensures \old(blackHeight(this)) == blackHeight(this);
  @ assignable footprint();
  @*/
private void rightRotate() {
    Tree l = left;
    Tree ll = left.left;
    Tree lr = left.right;
    Tree r = right;

    left = ll;
    right = l;
    right.left = lr;
    right.right = r;

    int t = key;
    key = right.key;
    right.key = t;

    //@ set right.heightVariant = heightVariant;
    //@ set heightVariant = heightVariant + 1;
}
```

Listing 3.20: `rightRotate` with its contract

4 Verification

As already mentioned in Section 2.3, KeY can find proofs of method contracts with an automated proof search. Its strategy can only work with general heuristics, though, while a human prover tends to have an intuition about how a proof can be closed successfully. This is especially true for more complex proof situations, where the automatic can get easily "distracted" by the sheer number of available formulas to work with. In this chapter, we therefore describe the most important aspects of our interactive verification process with KeY for the contracts specified in Section 3.4. In addition, Section 4.6 presents some statistics on the proofs. All proofs are available on GitHub[1] together with the source code and specifications.

4.1 General Approach

Our general approach was to do the following: First, after loading a method contract into KeY, we would run the Full Auto Pilot (see Section 2.3) so that all proof obligations get neatly separated. Some of them could be further split manually, for example an invariant into all of its parts. On this finer level of granularity, we experimented with different settings for the proof search strategy to close as many goals as possible automatically, which is detailed in Section 4.2. We would then analyse the goals still left open, looking for what information the automatic is missing, and guide the prover interactively with our knowledge. This could for example be the expansion of needed definitions, manual cuts or the use of dependency contracts.

However, specifications – and with them their proofs – usually go through several iterations while working on them: Sometimes, the missing information to close a goal is really an additional `requires` clause; or a called method has to ensure more properties about its final state. This often necessitates a complete redo of the proof. Also, we intentionally chose a bottom-up approach, in which we first proved that the binary-search-tree property is preserved by a naive insertion, and only then started to look at the fix operations and the subsequent restoration of the red-black properties. This allowed us to establish a solid foundation of basic specification fragments, before refining it for more complex reasoning tasks, without having to deal with everything at once.

To be able to properly build on the knowledge we gained in earlier iterations of the proof and to not "throw away" all the manual interactions carefully applied there, we used some (only recently developed) techniques to persist and automise proofs. We mainly worked with assertions and *JML Scripts*, described in more detail in Section 4.3 and 4.4, and also tried out *Proof Caching*, detailed in Section 4.5.

[1] `https://github.com/gewitternacht/rbtree-verification`

4.2 Proof Search Strategy Settings

The success of KeY's automated proof search turned out to be highly sensitive to the settings chosen for the proof search strategy. As a basis, we used the predefined Java verif. std. strategy, with settings that are generally sensible for Java verification. We had to make some adjustments, though:

Class Axiom Rule This setting controls when the definitions of class axioms, such as invariants or model methods, are expanded. It is set to Delayed by default, which is unsuitable for us. As we use many model methods and most of them are defined recursively, the automatic very quickly inflates the sequent by repeatedly expanding the definition of e.g. `footprint`, instead of expanding other definitions that would advance the proof. Therefore, we set the Class Axiom Rule to Off most of the time and manually expanded exactly those definitions that were needed in the specific situation. Occasionally, though, switching the setting back to Delayed could close a few goals automatically.

Proof Splitting Sometimes, goals that closed automatically in an earlier iteration of the proof would no longer close after small (unrelated) modifications in the sequent. In some of these situations, the proof was found again after setting the Proof Splitting option – by default Delayed – to Off. This option controls the applications of rules that split the proof, that is, lead to several proof branches, for example making a case distinction for an if-then-else expression.

Settings while Running the Auto Pilot For long runs of the Auto Pilot, we found that at some point – mid-symbolic execution – the automatic strategy would completely focus on the use of dependency contracts and the expansion of local queries. Queries are methods used as a function in the logic, and our "local queries" getting expanded would typically be calls to isRed. This cluttered the sequent with unnecessary formulas and prevented real progress. Therefore, we set the Dependency Contracts and Expand Local Queries to Off during the Auto Pilot execution. As they are generally needed in some situations, though, we switched them back to their default of On afterwards.

4.3 Assertions

We added many assertions to almost all methods for the verification of their contracts. They serve various purposes, which are described below.

Structuring the Proof The use of assertions allows us to structure a proof more clearly, since we can create one designated branch in the proof tree for each proof obligation and keep track of them more easily. Also, we can order the individual goals in a specific way that facilitates things – the "easy" ones first, the "harder" ones building thereon later. Similarly, if we observe that the same cut is necessary on several of these branches – for example, this could be the fact that `right == \old(left)` in `rightRotate` – we can capture this in its own `assert` and access it in all following assertions. Finally, they are also useful for breaking down complex obligation into smaller subgoals: In the `rightRotate` method, for instance (see Figure 3.1 and Listing 3.20), we need to show that the rotation preserves the `blackHeight` of the tree. This is done by incrementally proving that

1. `left` was red in the prestate, and `right` is red now
2. the `blackHeights` of `left.left`, `left.right` and `right` were all equal in the prestate
3. the `blackHeights` of the three subtrees starting at these nodes didn't change
4. therefore, now `left`, `right.left` and `right.right` all have the same `blackHeight`
5. based on all of this, we can conclude that the overall `blackHeight` is preserved

With an approach like this, it is more likely that the automated proof search strategy finds proofs on its own, as we guide it by providing desired intermediate results that facilitate the subsequent reasoning. In the appendix in Section A.3.1, the complete list of assertions for the `rightRotate` method is given to illustrate their heavy usage.

Capturing the State of Model Methods Most of the assertions are for framing purposes, though, stating that "untouched" parts of the tree did not change. After the recursive `right.add` call in `addRight` (see Listing 3.16), for example, it is important to know for the rest of the proof, that none of `left`'s model methods were affected by this. Therefore, we use a number of assertions like

```
//@ assert left == null || left.footprint() == \old(left.footprint());
```

to capture the result of model methods in such an intermediate state. In the end, this allows us to "chain" equalities stemming from these assertions, together with information that the contracts of called methods provide (for example that of `setHeight`, see Listing 3.19), in order to reason about the overall change that model method's results have gone through. The three assertions that are needed for item 3 above are also of this kind – in such a case, they are used to express that subtrees "further down" didn't change and therefore, we can concentrate on the local changes for subsequent reasoning.

Using Lemmas Lastly, the contracts of the "lemma model methods" defined in Section 3.3.4 can be "invoked" in another proof with the use of `assert` statements:

```
//@ assert invLemmaLess();
```

When the prover reaches this assertion, it can easily prove its validity by expanding the method's definition, which is simply `true`. For the subsequent reasoning, it can then use the additional information provided by the method contract. In this case, the lemma provides an alternative formulation of the binary-search-tree property, which facilitates closing some of the branches in the proof of `Tree::contains`.

4.4 JML Scripts

Motivation All the assertions described in the previous section also serve one purpose not explicitly mentioned there: They help in persisting proofs between several revisions of the specification. Without them, it would be a lot harder to recreate the already successful parts of the proof that were found interactively. Still, manual interaction is often necessary to prove the validity of these assertions, which poses the challenge of, for example, remembering which definition expansions were needed in which situation or what was changed in the strategy settings to close a branch automatically. This, as well as the fact that many of the "framing assertions" have very similar proofs, is addressed by *JML Scripts*.

Concept *JML Scripts* are a new and, as of now, experimental addition to KeY (they are not available in a released KeY version yet), and were used on a larger scale for the first time in this work. They existed prior to this case study, but were shaped by the needs arising over the course of it, in constant dialogue with the KeY developers. Manually executed proof steps can be documented directly in the source code with one JML Script per `assert`. A script is introduced by the keyword `\by`, surrounded with curly braces, and consists of a semicolon-separated list of prover directives indicating how to prove the validity of the corresponding assertion. It can be invoked during a proof by running the Script-Aware Auto Pilot macro.

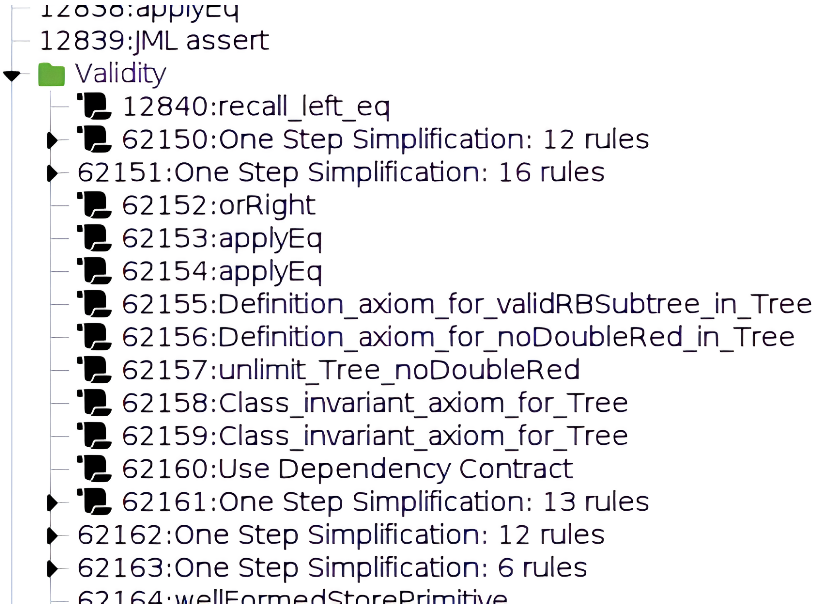
Assertion Labels Listing 4.1 shows an example of two JML Scripts that are part of the `addRight` method (in line 18 of Listing 3.16) and together prove that `left.noDoubleRed` still holds after the insertion of a new `right` node. The short one in line 1 simply invokes the automated search strategy with the `auto` command, which suffices in this situation. Additionally, this assertion uses another new feature of KeY that was introduced by the KeY developers for this work: An *assertion label*, which allows us to name the assertion (here `left_eq`) and reference it again later with a dynamically generated taclet (here recall_left_eq). This is important because assertions that are "obvious" enough have often disappeared from the sequent by the time they are needed, due to KeY's simplifications.

Commands In the following, we will explain the available script commands that were most important for us by stepping through the more complicated script of the assertion `left_nodoublered` that starts in line 3 of Listing 4.1. The result of the executed script as shown in the proof tree in KeY can be seen in Figure 4.1. First, the `rule` command is used to state an arbitrary taclet, in this case the aforementioned recall_left_eq, that should be applied on the current sequent. Next, the `oss` command performs One Step Simplifications where applicable (two in this case) and the `macro` command calls a predefined macro[2]. These preparations allow us to apply the `left_eq` equality on the formula of `left_nodoublered` (which is needed for the `dependency` command later), again using the `rule` command, but this time specifying additional necessary parameters, namely the

[2] `nosplit-prop` decomposes propositional top-level formulas, but without splitting the goal

```
//@ assert left_eq:              left == \old(left)                 \by { auto; }
/*@ assert left_nodoublered:     left == null || left.noDoubleRed() \by {
        rule recall_left_eq;
        oss;
        macro nosplit-prop;
        rule applyEq on="self.left@heapAT" occ=1;
        rule applyEq on="self.left@heapAT" occ=1;
        expand on="self.validRBSubtree()";
        expand on="self.noDoubleRed()";
        rule unlimit_Tree_noDoubleRed on="Tree::noDoubleRed$lmtd(heap,
            Tree::select(heap, self, Tree::$left))";
        expand on="self.<inv>";
        expand on="self.left.<inv>" occ=0;
        dependency on="(self.left@heap).noDoubleRed()@heapAT";
        auto classAxioms=false steps=500;
    }
  @*/
```

Listing 4.1: An example of JML Scripts with assertion labels



```
12838:applyEq
12839:JML assert
Validity
    12840:recall_left_eq
    62150:One Step Simplification: 12 rules
    62151:One Step Simplification: 16 rules
    62152:orRight
    62153:applyEq
    62154:applyEq
    62155:Definition_axiom_for_validRBSubtree_in_Tree
    62156:Definition_axiom_for_noDoubleRed_in_Tree
    62157:unlimit_Tree_noDoubleRed
    62158:Class_invariant_axiom_for_Tree
    62159:Class_invariant_axiom_for_Tree
    62160:Use Dependency Contract
    62161:One Step Simplification: 13 rules
    62162:One Step Simplification: 12 rules
    62163:One Step Simplification: 6 rules
    62164:wellFormedStorePrimitive
```

Figure 4.1: Executed second script from Listing 4.1 as shown in the proof tree in KeY

formula to apply the rule on and its occurrence in the sequent.[3] After this, we use the `expand` command four times, which can expand the definition of given model methods or invariants. The first two expansions "dig up" the `left.noDoubleRed` from the prestate – the `rule` application thereafter is a technicality that we won't examine further here – and the latter two expose necessary information contained in the instance invariants. Finally, we can apply the dependency contract of `left.noDoubleRed` on the now present `(self.left@heap).noDoubleRed()@heapAT` and `self.left.noDoubleRed()`. The branch can then be closed by an invocation of the automated proof search with a sufficient number of steps, and with the Class Axiom Rule temporarily turned off (cf. Section 4.2).

This can be considered a script of average length for this work, others ranging from 1 to 50 lines, of which an example can be found in the appendix, Section A.3.2.

4.5 Proof Caching

Proof Caching is another new feature of KeY with the purpose of reducing the amount of work that is necessary for proving a slightly modified specification. It was developed not long before this case study and we were the first to test it on a larger scale. A typical use case would be the following: We successfully closed (part of) a proof, but some adaptions needed to be made that necessitate a redo of most of the proof. Now, if there is an already closed branch in the old proof that is (in some specific way) similar to a new, open one, Proof Caching can automatically detect this and reuse the corresponding rule applications to close the new branch analogously to the old one. This is based on the observation that, with some restrictions, the following holds for sequents:

$$\text{if} \quad \Gamma \Longrightarrow \Delta \quad \text{is valid, then} \quad \Gamma, \Phi \Longrightarrow \Delta, \Psi \quad \text{is valid,}$$

with $\Gamma, \Delta, \Phi, \Psi$ all being sets of JavaDL formulas. Meaning: A sequent in the new proof is "similar" to one in the old proof if it consists of the same, and possibly additional formulas.

For the `addRight` method, there were situations where we would have liked to make use of Proof Caching, but couldn't do so because of the aforementioned restrictions: If there are modalities or queries on the sequent, which was the case due to several `isRed` queries, the observation above about the validity of sequents does not necessarily hold, and Proof Caching cannot be applied.

For the proof of the `setHeight` method, however, we were in a situation ideal for Proof Caching and tested it out. Specifically, we added some of the `ensures` clauses seen in Listing 3.19 only after proving the method contract for the first time. Proving all the ensured properties wasn't too complicated, but nevertheless required quite a few manual interactions that all would have had to be applied again. Luckily, Proof Caching worked very well and saved a lot of time. It detected virtually all of the already previously existing proof obligations as "cached", and we only needed to close the additional branches by hand.

[3] For better readability, `heapAT` represents the heap in the current state here, being an abbreviation for `heapAfter_Tree[self.right := self_0][self.heightVariant := 1 + self.heightVariant]`. Abbreviations were very recently added to JML Scripts, but not yet tested in this thesis.

		code	**spec**	**#asserts**	**script**	**total JML**
`Tree`	model methods etc.	-	149	-	-	149
	`Tree` constructor	4	12	0	0	12
	`contains`	11	11	6	6	25
	`add`	9	11	1	2	16
	`addRight`	23	13	44	247	370
	`addLeft`	21	13	-	-	13
	`rightRotate`	13	19	39	331	447
	`leftRotate`	13	19	-	-	19
	`recolour`	5	13	29	127	204
	`setHeight`	4	37	21	0	64
	`isRed`	3	5	0	0	5
	total `Tree`	**112**	**302**	**140**	**713**	**1 324**
`RBTree`	model methods etc.	-	26	-	-	26
	`RBTree` methods	19	29	1	0	30
	total `RBTree`	**19**	**55**	**1**	**0**	**56**
`Client`		**9**	**4**	**2**	**0**	**6**
total		**140**	**361**	**143**	**713**	**1 386**
estimated total		140	361	143	1 291	2 171

Table 4.2: Lines of code and specification, number of assertions, lines of script, and total lines of JML for every method

4.6 Proof Statistics

As mentioned earlier, we successfully proved all contracts needed for the verification of the `contains` and `add` methods of `RBTree` – with an exception of the `addLeft` and `leftRotate` methods, because they are completely symmetric to their "right" counterparts and would have required a lot of additional, repetitive work. This section presents statistics for all proven methods. First, we look at the lines of code and specification, and then at the number of rule applications.

Lines of Code and Specification Figure 4.2 shows the number of lines of each kind for every method, grouped by their class. The first column **code** counts the lines of normal Java code – the actual implementation of red-black trees. The second column **spec** gives the lines of specification, including method contracts etc., but no assertions or scripts. Those are given in the next two columns, **#asserts** counting the assertions, and **script** the lines of JML Scripts. Lastly, **total JML** summarises all lines for specification purposes.

The rows "model methods etc." include all lines of JML that do not belong to one specific Java method, like the definitions of model methods and instance invariants. We can see that these (together 175 lines) make up about half of the specification, which consists of 361 lines. This, in turn, is more than twice as much as the 148 lines of Java code, which is not uncommon for formal specification. In addition to that, there are 143 assertions, and 713 lines of JML Scripts attached to them for prover guidance, resulting in a total of 1386 lines of JML – nearly ten times the amount of Java code. Most of them were written for the addRight, rightRotate and recolour methods, which were the most difficult and time-consuming to prove. All other methods were either a lot simpler, or the effort necessary to create scripts would have exceeded the expected benefits.

As we did not prove addLeft and leftRotate, there are no assertions and scripts there, but we estimate the necessary effort to be equal to that of the "right" versions. Consequently, for proving these methods as well, the estimated total of JML lines increases to about 15 times the lines of Java code.

Number of Rule Applications Figure 4.3 contains the statistics for all completed proofs, again grouped by class. The column **rule apps** lists the number of rule applications and **manual** how many of them were applied manually. Next, the column **sliced** shows the number of rule applications after *proof slicing*, a feature of KeY that allows to reduce proofs to their necessary steps. The percentage of those **useful** steps in the initial number of steps is given in the last column.

The rows "model method accs" summarise all indented rows below them, which contain the statistics for the accessible clauses of model methods. Also, both contains methods and isRed have an additional accessible to prove, all other methods only have proofs for their normal_behaviour. By far the largest and also most time-consuming proof is that of addRight with over 80,000 rule applications, 361 of them manually executed. Following are the proofs for rightRotate, setHeight and recolour, and, not to be underestimated, that of the accessible clauses of the Tree instance invariant and treeSet. All accessible clauses together make up about one fourth of the overall number of applied rules.

The manually applied rules amount to a total of 880 rules. As the majority of scripts were developed from previously manually executed steps, and one line of script can be considered the equivalent of one manual rule application, we reach a total of nearly 1600 "guided steps" by adding the number of script lines from Figure 4.2 to the manual steps from 4.3. We estimate that about 100 additional manual steps were "lost" through the application of Proof Caching, which does not preserve this information. The top rules applied manually include: expansion of the footprint definition, of other model methods, and of the instance invariant; applyEq – mostly in preparation for Use Dependency Contract; and different branching rules, like cut_direct, eqTermCut or ifthenelsesplit, for example to make case distinction like self.left = null.

In contrast to the automatically applied rules, these manual interactions are almost all considered "useful" by the proof slicing. For most of the larger proofs, the application thereof results in proofs decreased to about a third of their original size. Unfortunately, for the addRight method and also the client, proof slicing cannot be applied due to some bugs (marked with ? in the table). However, we can still run a dependency analysis that lets us estimate the percentage of useful applications in addRight to be about 25%. The top "useless" rules of all proofs include: One Step Simplification; the polySimp_mulComm0 rule, which commutes the factors of a multiplication (one of which is always −1 in our case, originating from previous normalisations); applyEq, replaceKnownSelect, replaceKnownAuxiliaryConstant and replace_known_left/_right; andLeft; true_left; and for some of the accessible proofs elementOfSetMinus, elementOfUnion and elementOfSingleton.

proof		rule apps	manual	sliced	useful
Tree	model method accs	52 021	311	15 542	30%
	– blackBalanced	2 734	44	912	33%
	– blackHeight	1 453	28	614	42%
	– doubleRedLeft	2 960	34	581	20%
	– doubleRedRight	2 291	34	603	26%
	– doubleRedTop	142	3	108	76%
	– footprint	4 775	28	1 228	26%
	– inv	20 686	74	5 036	24%
	– noDoubleRed	5 390	35	2 076	39%
	– treeSet	10 824	27	3 912	36%
	– validRBSubtree	218	2	169	78%
	– validRBSubtreeERT	548	2	303	55%
	invLemmaGreater	152	3	147	97%
	invLemmaLess	147	3	142	97%
	Tree constructor	1 354	4	1 181	87%
	contains	9 636	4	2 952	31%
	– normal_behaviour	8 599	0	2 388	28%
	– accessible	1 037	4	564	54%
	add	3 570	0	2 722	76%
	addRight	83 506	361	?	?
	rightRotate	33 819	57	12 671	37%
	recolour	12 149	31	4 612	38%
	setHeight	23 540	13	5 855	25%
	isRed	486	0	452	93%
	– normal_behaviour	180	0	173	96%
	– accessible	306	0	279	91%
	total Tree	**199 232**	**787**		
RBTree	model methods accs	2 675	20	1 180	44%
	– footprint	357	4	201	56%
	– treeSet	280	4	153	55%
	– validRBTree	2 038	12	826	41%
	RBTree constructor	222	0	222	100%
	add	8 121	15	2 609	32%
	contains	1 130	7	767	68%
	– normal_behaviour	422	0	300	71%
	– accessible	708	7	467	66%
	fixRootColour	3 035	45	1 024	34%
	total RBTree	**15 183**	**87**	**5 802**	**38%**
Client		**2 101**	**6**	?	?
total		**216 516**	**880**		

Table 4.3: Number of rule applications, before and after *proof slicing*, with the percentage of “useful” rule applications, and the number of manually applied rules

5 Insights

This chapter presents the insights that we gained throughout the course of this work. This includes our experience with dynamic frames for tree structures, some findings about KeY's automated proof search strategy, a few bugs we detected and, lastly, several features of KeY that were or would have been of great value for this work.

5.1 Framing

Regarding KeY's approach to framing, we conclude that successful reasoning over tree structures with dynamic frames is possible, yet very work-intensive. During the course of this work, a significant amount of time and effort went into determining at what time we should assert or ensure that which model methods' results are unchanged – to make sure that no information is missing later on, but at the same time avoid unnecessary effort proving superfluous assertions.

With `setHeight`, there even is a method contract that exists almost exclusively for framing purposes. This reflects our realisation that it can be beneficial to capture as many non-changing model methods as possible in a contract, instead of asserting them one by one after calling the method. This way, the preconditions for using the model method's dependency contract have to be shown only once, which often requires some manual steps despite matching `assignable` and `accessible` clauses. However, these additional `ensures` clauses (as e.g. in `RBTree::fixRootColour`, see Section 3.4.2) are not a panacea either. They clutter the method contracts and, more importantly, are not always possible as would be needed. For example, there is no way for the `add` method to ensure that the sibling of the node it was called on is unaffected – that is, `right.add` does not affect `left` – as siblings do not hold references to each other.

To illustrate the amount of work going into framing, we gathered the statistics in Figure 5.1. There, we compare the lines of assertions and scripts (for `setHeight` the number of assertions) by their purpose, which is either framing related or for the actual reasoning about red-black- or binary-search-tree properties. To allow for a fair comparison, we only consider methods where all parts of the proof are equally automised. For `addRight`,

	framing	**red-black trees**
`rightRotate()`	248	188
`recolour()`	137	54
`setHeight()`: #asserts	20	1

Table 5.1: Lines of assertions and scripts by purpose: framing clearly dominates the actual reasoning about red-black trees

for example, this is not the case, as we found that writing scripts for framing assertions was easier due to their similarity, compared to the always different reasoning about keys and colours.

Still, the numbers in Figure 5.1 match the gut feeling that has emerged over the course of this thesis: Our estimate is that about three quarters of the total work was spent on framing issues, compared to one quarter for the reasoning over actual red-black properties. Writing scripts for framing assertions might be easier and parts of them can be reused for similar ones – however, problems with framing were also the primary reason for revisions of the JML annotations being necessary and proofs needing to be redone.

5.2 KeY's Proof Search Strategy

As described in Section 4.2, the success of the automated proof search can be notably influenced by the chosen strategy settings. We first had to learn that turning Dependency Contracts and Expand Local Queries off was the key to success for long Auto Pilot runs, and that playing around with settings like Proof Splitting could lead to branches closing automatically. Furthermore, KeY's not-so-useful automatic expansion of model methods caused us to set Class Axiom Rule to off, which had the effect that the manual expansion of needed model methods represented a significant part of the effort that went into the verification. Besides this, there were two more notable findings about the proof search strategy.

Reasoning about Sets In several situations where reasoning about location sets (type LocSet) was necessary to close a branch, KeY's proof search didn't work as well as we would have expected or liked it to. For such branches it is especially true that turning off Proof Splitting can help to close them automatically, but there also are some that, while being obviously provable for a human, are not closed with any chosen strategy settings in a reasonable amount of time.

One such situation is a goal that can be reduced to

$$\neg\texttt{self.right = null} \Longrightarrow \neg\texttt{self.right.footprint() = }\emptyset$$

or even further to

$$\Longrightarrow \neg\texttt{self.right.}{*}\texttt{ = }\emptyset$$

for which no proof is found – only after applying the taclet equalityToElementOf manually, 13 steps suffice to close the goal automatically. The sequents above contain many additional formulas both in the antecedent in succedent in the original proof situation. This potentially contributes to KeY not finding a proof due to too many "distractions" – but at least for the examples above, we went through the trouble of hiding all other formulas and found that equalityToElementOf, part of the semantics_blasting heuristic, is actually *never* applied by the automatic and the branch never closed, even if no other rules are available for application.

That the heuristics for applying `LocSet`-related taclets could probably be improved also shows in the proofs for the `accessible` clauses of `treeSet` and `footprint`.[1] While `footprint` works with the `LocSet` type, `treeSet` uses the custom integer set definition from Section A.2. The methods' definitions are very similar, and both their proofs were prepared with the same three manual steps before running the automated proof search strategy. However, while the proof for `treeSet` closes with less than 6,000 rule applications, the one for `footprint` needs about 36,000. With proof slicing, the `treeSet` proof can be decreased to 900 rule applications, and the `footprint` proof to about 1,800.

Invisible Internal State Another peculiarity of working with KeY is its invisible internal state, determining which rules should have which priority for the next proof step. Depending on how a specific sequent originated, this internal state leads to different rules being applied by the automated search. Generally, this is important and guarantees a "fair" application of rules, but in practice this can have frustrating results. For example, there were several occasions where "Validity" branches of assertions that formerly closed automatically wouldn't do that any more, just because a few additional assertions were inserted before them. Even after hiding all new formulas to obtain the exact same sequents, KeY's internal state of rule priorities lead to one of them closing after a few hundred steps and the other one not even after several thousands.

Moreover, we noticed a situation in which some internal state apparently differs when this should actually not be the case: For some goals, running the automated proof search would result in a very branched proof tree, with no success after several thousand steps. However, pruning this part of the proof and simply running the automated search again with the exact same settings could close the goal in a few hundred steps without any branching.

5.3 Bugs in KeY

This section describes some bugs that we found in KeY during the course of this thesis, at times hindering productive work or complicating things.

Ignored Updates to Heap Variables after Pruning This bug lead to branches suddenly not closing automatically any more and was quite challenging to identify: If a "Validity" branch for an assertion is generated by symbolic execution, and afterwards it is pruned and generated a second time, taclets like `simplifyUpdate1` ignore updates to heap variables. Something like

```
{ ... || heapBefore_foo:=heap || ... }heapBefore_foo
```

is incorrectly simplified to `heapBefore_foo` instead of `heap` in this case. We created issue #3206 in KeY's GitHub repository[2], that describes this bug in more detail.

[1] These proofs were done for previous versions of the `accessible` clauses, where the complete `footprint` was specified to be accessible. The new versions additionally exclude some fields (see Section 3.4.1), and because of that need more manual interaction overall, making a comparison less meanigful.

[2] `https://github.com/KeYProject/key/issues`

Ignored nullable annotation for Model Method Parameters Due to this bug, we refrained from making all model methods `static` with `nullable` arguments like we did with `blackHeight` (see Section 3.3.6): When defining a model method with a `nullable` argument like

```
helper model int foo(nullable Object o) { ... }
```

KeY completely ignores the `nullable` annotation. For example, this can be observed when loading the contract for the model method itself, an accessible clause thereof or when applying the rule Use Dependency Contract on it – they all look the same as if `nullable` is omitted and contain `(¬o = null)«impl»` somewhere. Because of this, even though `setHeight` avoids case distinctions for `null` in its definition, it probably resulted in increased effort overall, due to the case distinctions needed before applying Use Dependency Contract. For more information on this bug, see our issue #3186 on GitHub.

Other Bugs There were or are several further bugs that did not interfere with our work as much as the two described above and therefore are only shortly mentioned here. One that is already fixed (issue #3149) consisted of nested ternary operators with integers – legal from Java's perspective – not loading due to mismatched types of `int` and KeY's internal `\bigint`. Another fixed issue (#3192) manifested in proof slicing or reloading of proofs sometimes failing when they used a dependency contract. Very frustrating, but occurring only twice, was a bug that was supposedly fixed several years ago (see #806): Due to some internal `ClassCastException`, proofs could not be saved and therefore all manual interactions since the last saving had to be redone. And lastly, in discussion with Mattias, a soundness issue was detected by him, regarding the well-foundedness of invariants and model methods, see #3155.

5.4 Desirable Features

This section presents a collection of KeY features that either already exist and were of great value during the course of this work, some with the potential for further improvements, or that we missed and in our eyes could be valuable additions.

Model Methods As indicated by the number of model methods described in Section 3.3, they are an indispensable part of this work – for abstracting from concrete object structures, modelling properties and even as a way to formulate lemmas. We suspect that without them, a successful verification of red-black trees would not have been feasible. Due to their relevance for more complex verification efforts, we consider it important to fix existing issues regarding model methods, for example the ignored `nullable` annotations or problems with well-foundedness mentioned above.

Assertions, Labels and JML Scripts The possibility to guide a proof with assertions was crucial for the verification process. Assertion labels and JML Scripts (see Section 4.4) are additions that have great potential to further automise and persist proofs.

However, as of now, creating these scripts is very work-intensive and error-prone, because they have to be manually put together, e.g. by copying taclet names and formulas, determining the `occ` parameter, and extracting the used strategy setting for `auto` – all of which was already performed interactively and KeY "knows about". In our eyes, it would therefore be both feasible and of great value to provide a *script generator* that automatically extracts a script from a manually closed branch.

Proof Caching For the instances where Proof Caching was applicable, it was very useful and worked well. Further testing has the potential to find and resolve little quirks to improve the usability – for instance, we noticed that differing term labels prevent the detection of a cached goal. The applicability of Proof Caching would profit from the relaxation of some of the current restrictions, for example allowing queries on the sequent[3], or enabling the user to manually mark sequents that should be added to the cache.

Proof Search Strategy Settings The proof search strategy settings (see Section 4.2 and 5.2) would profit from more transparency. This could be in the form of better (available) explanations for the settings and what their effects are, and some hints about which settings might help in which practical situation. Also, it could be convenient to have a macro that tries to close a goal with several setting combinations one after the other – for this work, we could have tried combinations of Class Axiom Rule and Proof Splitting set to Off and Delayed without having to change the settings manually each time.

Reasoning about Sets As described in Section 5.2, there were situations in which we were not satisfied with the automatic reasoning about sets, even more so because the custom defined integer sets sometimes seemed to work better than the built-in location sets. It is already planned to address this in a future work.

Branching for Assumes A significant amount of manually applied rules only "prepared" formulas, or generally the sequent, for the application of a rule we actually wanted to use. Expanding definitions of terms like `self.right.footprint()`, for example, is only possible if the right side of the sequent contains ¬`self.right = null`, which is specified by an `assumes` clause for the corresponding taclet. To expand the definition in formulas like

$$\texttt{self.right = null} \;\lor\; \texttt{self.right.footprint() = ...} \quad ,$$

a common situation for us, we consequently have to make a cut for ¬`self.right = null` first. To avoid these tiresome preparations being necessary in situations like this, we propose to provide a way to apply rules despite missing `assumes` clauses. Their correct usage can be ensured by generating a new branch at this point, which has the proof obligation to show the missing requirements necessary for the rule application.

[3] motivated by the insights of this work, this is actually in progress at the time

Focus on Formulas On several occasions, we knew that (or wanted to know if) a small subset of the formulas present on the sequent is sufficient to close the goal. In situations like this, it can be useful to hide all other "distracting" formulas, which is a valid operation on a sequent, but only possible by cumbersomely selecting and hiding each of those formulas individually. A way to select formulas that KeY should focus on during the proof search, or some "batch-hiding" functionality, would greatly improve the usability.

Unchangeable Formulas Similarly, we sometimes knew that specific formulas on the sequent should not be modified or even simplified away by the automated proof search strategy, so that they are available later on when we need them. For this, a way to mark formulas as unchangeable would be useful.

Interactive Proof Loader If something in the specification changes, many proofs cannot be (completely) loaded into KeY any more and large parts of them have to be redone. Sometimes, though, the corresponding `.proof` file would only need changes in, or the addition of, a few lines. Some kind of interactive proof loader could help to reuse the old proofs in such cases: This could mean to stop loading the proof if a non-applicable rule is detected and give the user different options on how to resolve this issue, interactively adapting the old proof to the new situation. Based on the chosen resolvement option, KeY could then try to automatically adapt the rest of the proof, stopping again if an issue occurs.

Enhanced Approach to Framing As discussed in detail in Section 5.1, KeY's dynamic frames are suboptimal for reasoning about tree structures. Looking at the work of Armborst and Huisman (2021), separation logic, in contrast, seems to be an ideal fit for the representation of and reasoning about trees. For future works considering tree structures, an enhanced approach to framing could reduce the necessary effort significantly. Possible options could be a combination of dynamic frames and separation logic, like the *dynamic separation logic* proposed by Hans-Dieter Hiep at the 19th KeY Symposium 2023[4], or the introduction of an ownership concept into KeY, as already explored by Pfeifer (2018) and Scheurer (2020).

[4] `https://www.key-project.org/wp-content/uploads/2023/08/dsl.pdf`

6 Conclusion and Future Work

The result of this thesis is a successfully specified and verified Java implementation of red-black trees.[1] We provide an example for future case studies regarding the profitable usage of model methods for the specification, and of assertions for the verification, as well as some findings about what effects the chosen proof strategy settings can have. The JML Scripts and Proof Caching tested in this work are promising features for automising and persisting proofs in KeY. As one of the first case studies in KeY about tree structures, and the first using dynamic frames for the verification of red-black-trees, we gained valuable insights about this combination: Successful reasoning over tree structures with dynamic frames is possible, yet very work-intensive, and KeY could profit from an enhanced approach to framing for future efforts in this area. By finding some bugs and usability issues of KeY, and giving suggestions for further enhancements, we contribute to the constant improvement of KeY as a powerful and useable tool.

Future work could use the results of this thesis and that of other red-black-tree verifications to draw a comparison between the used verification tools, possibly based on metrics suggested for VACID-0 by Leino and Moskal (2010).

Apart from that, one could look at the usage of ownership concepts to reprove the `contains` and `add` methods, comparing the necessary effort to that of the dynamic frames approach in this work. In this context, the additional verification of a `delete` method would also be interesting. To further push the limits of KeY, one could approach the JCL implementation again with its iterative `add` method and the use of parent pointers.

Finally, as this work only proves that the red-black properties hold true, assuming that this automatically leads to a balanced tree and efficient operations, it would also be interesting to actually verify the logarithmic complexity of, for example, the `add` method.

[1] available at `https://github.com/gewitternacht/rbtree-verification`

Bibliography

Georgii Maksimovich Adel'son-Velskii and Evgenii Mikhailovich Landis (1962). "An algorithm for organization of information". In: *Doklady Akademii Nauk*. Vol. 146. 2. Russian Academy of Sciences, pp. 263–266.

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. (2016). *Deductive Software Verification - The KeY Book: From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer. DOI: `10.1007/978-3-319-49812-6`.

Andrew W. Appel (2011). "Efficient verified red-black trees". In: URL: `https://www.cs.princeton.edu/~appel/papers/redblack.pdf`.

Lukas Armborst and Marieke Huisman (2021). "Permission-Based Verification of Red-Black Trees and Their Merging". In: *9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021*. Ed. by Simon Bliudze, Stefania Gnesi, Nico Plat, and Laura Semini. IEEE, pp. 111–123. DOI: `10.1109/FormaliSE52586.2021.00017`.

Yves Bertot and Pierre Castéran (2004). *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer. ISBN: 978-3-642-05880-6. DOI: `10.1007/978-3-662-07964-5`.

Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn (2017). "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings*. Ed. by Nadia Polikarpova and Steve A. Schneider. Vol. 10510. Lecture Notes in Computer Science. Springer, pp. 102–110. DOI: `10.1007/978-3-319-66845-1_7`.

Daniel Bruns (2011). "Specification of red-black trees: Showcasing dynamic frames, model fields and sequences". In: *10th KeY Symposium, Nijmegen, the Netherlands*, p. 296.

Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich (2015). "Implementation-level verification of algorithms with KeY". In: *International journal on software tools for technology transfer* 17.6, pp. 729–744. DOI: `10.1007/s10009-013-0293-y`.

Martin de Boer, Stijn de Gouw, Jonas Klamroth, Christian Jung, Mattias Ulbrich, and Alexander Weigl (June 1, 2022). "Formal Specification and Verification of JDK's Identity Hash Map Implementation". In: *17th International Conference on integrated Formal Methods (iFM 2022)*. Ed. by Maurice H. Beek and Rosemary Monahan. Vol. 13274. Lecture Notes in Computer Science. Springer, pp. 45–62. ISBN: 978-3-031-07727-2. DOI: `10.1007/978-3-031-07727-2_4`. published.

Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle (2015). "OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and

Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, pp. 273–289. DOI: 10.1007/978-3-319-21690-4_16.

Jean-Christophe Filliâtre and Pierre Letouzey (2004). "Functors for Proofs and Programs". In: *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings.* Ed. by David A. Schmidt. Vol. 2986. Lecture Notes in Computer Science. Springer, pp. 370–384. DOI: 10.1007/978-3-540-24725-8_26.

Hans-Dieter A. Hiep, Olaf Maathuis, Jinting Bian, Frank S. de Boer, and Stijn de Gouw (2022). "Verifying OpenJDK's LinkedList using KeY (extended paper)". In: *Int. J. Softw. Tools Technol. Transf.* 24.5, pp. 783–802. DOI: 10.1007/s10009-022-00679-7.

Stefan Kahrs (2001). "Red-black trees with types". In: *Journal of Functional Programming* 11.4, pp. 425–432. DOI: 10.1017/S0956796801004026.

K. Rustan M. Leino (2010). "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers.* Ed. by Edmund M. Clarke and Andrei Voronkov. Vol. 6355. Lecture Notes in Computer Science. Springer, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

K. Rustan M. Leino and Michał Moskal (2010). "VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0". In: *Proceedings of Tools and Experiments Workshop at VSTTE.*

Charles Eric Leiserson, Ronald L. Rivest, Thomas H. Cormen, and Clifford Stein (1994). *Introduction to algorithms.* Vol. 3. MIT press Cambridge, MA, USA.

Zohar Manna, Henny B. Sipma, and Ting Zhang (2007). "Verifying Balanced Trees". In: *Logical Foundations of Computer Science, International Symposium, LFCS 2007, New York, NY, USA, June 4-7, 2007, Proceedings.* Ed. by Sergei N. Artëmov and Anil Nerode. Vol. 4514. Lecture Notes in Computer Science. Springer, pp. 363–378. DOI: 10.1007/978-3-540-72734-7_26.

Huu-Minh Nguyen (2019). "Formal verification of a red-black tree data structure". MA thesis. University of Twente.

Tobias Nipkow (2016). "Automatic Functional Correctness Proofs for Functional Search Trees". In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings.* Ed. by Jasmin Christian Blanchette and Stephan Merz. Vol. 9807. Lecture Notes in Computer Science. Springer, pp. 307–322. DOI: 10.1007/978-3-319-43144-4_19.

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* Vol. 2283. Lecture Notes in Computer Science. Springer. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

Ricardo Peña (2020). "An Assertional Proof of Red-Black Trees Using Dafny". In: *J. Autom. Reason.* 64.4, pp. 767–791. DOI: 10.1007/s10817-019-09534-y.

Wolfram Pfeifer (2018). *A New Verifcation Methodology Based on Dynamic Frames and Universe Types.* Project Report. KIT.

Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif (2022a). "Separating Separation Logic - Modular Verification of Red-Black Trees". In: *Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022,* *Trento, Italy, October 17-18, 2022, Revised Selected Papers*. Ed. by Akash Lal and Stefano Tonetta. Vol. 13800. Lecture Notes in Computer Science. Springer, pp. 129–147. DOI: 10.1007/978-3-031-25803-9_8.

Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif (2022b). "Software & System Verification with KIV". In: *The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen. Vol. 13360. Lecture Notes in Computer Science. Springer, pp. 408–436. DOI: 10.1007/978-3-031-08166-8_20.

Tim Scheurer (2020). *Modelling and Exploiting Ownership-Annotations Using Dynamic Frames in KeY*. Bachelor Thesis. KIT.

A Appendix

A.1 Java Source Code

This section contains the complete Java source code of our implementation of red-black trees, without any specification. For the complete specification, refer to the GitHub repository.[1]

A.1.1 Client.java

```java
public class Client {

    public static void main(String[] args) {
        RBTree t = new RBTree();

        /*      -3      -3             5              5
                          \           / \            / \
                    ~>     5   ~>   -3   9    ~>   -3   9
                                                       /
                                                      7         */
        t.add(-3);
        t.add(5);
        t.add(9);   // rotation
        t.add(7);   // recolouring
    }

    // method printing a tree to help understand the behaviour of red-black trees
    public static void printTree(RBTree t) {
        System.out.println(treeString(t.root));
    }

    public static String treeString(Tree t) {
        return (t.left == null ? "" : ("(" + treeString(t.left) + ")-"))
            + t.key + (t.isRed ? "r" : "b")
            + (t.right == null ? "" : ("-(" + treeString(t.right) + ")"));
    }
}
```

[1] https://github.com/gewitternacht/rbtree-verification

A.1.2 RBTree.java

```java
final public class RBTree {

    public RBTree() {
        // default constructor only made explicit for specification purposes
    }

    public boolean contains(int key) {
        return root != null && root.contains(key);
    }

    public void add(int key) {
        if (root == null) {
            root = new Tree(key);
        } else {
            root.add(key);
        }
        fixRootColour();
    }

    private void fixRootColour() {
        root.isRed = false;
    }
}
```

A.1.3 Tree.java

```java
final public class Tree {
    Tree left;
    Tree right;
    int key;
    boolean isRed;

    public Tree(int key) {
        this.key = key;
        isRed = true;
    }

    public boolean contains(int key) {
        Tree node = this;
        while (node != null && node.key != key) {
            if (key < node.key) {
                node = node.left;
            } else {
                node = node.right;
            }
        }
        return node != null;
    }

    public void add(int key) {
        if (key == this.key) {
            return;
        } else if (key < this.key) {
            addLeft(key);
        } else {
            addRight(key);
        }
    }

    private void addLeft(int key) {
        if (this.left == null) {
            this.left = new Tree(key);
        } else {
            this.left.add(key);
            setHeight();

            if (!isRed && left.isRed) {
                if (isRed(right) & (isRed(left.left) | isRed(left.right))) {
                    recolour();
                    return;
                }
                if (isRed(left.right)) {
                    left.leftRotate();
                    setHeight();
                }
                if (isRed(left.left)) {
                    rightRotate();
                }
            }
        }
    }

    private void addRight(int key) {
        if (this.right == null) {
            Tree newRight = new Tree(key);
            this.right = newRight;
        } else {
            this.right.add(key);
            setHeight();

            if (!isRed & right.isRed) {
                if (isRed(left) & (isRed(right.left) | isRed(right.right))) {
                    recolour();
                    return;
                }
                if (isRed(right.left)) {
                    right.rightRotate();
                    setHeight();
                }
                if (isRed(right.right)) {
                    leftRotate();
                    return;
                }
            }
        }
    }

    private void leftRotate() {
        Tree r = right;
        Tree rr = right.right;
        Tree rl = right.left;
        Tree l = left;

        right = rr;
        left = r;
        left.right = rl;
        left.left = l;

        int t = key;
        key = left.key;
        left.key = t;
    }

    private void rightRotate() {
        Tree l = left;
        Tree ll = left.left;
        Tree lr = left.right;
        Tree r = right;

        left = ll;
        right = l;
        right.left = lr;
        right.right = r;

        int t = key;
        key = right.key;
        right.key = t;
    }

    private static boolean isRed(Tree t) {
        return t != null && t.isRed;
    }

    private void setHeight() { /* method for specification purposes only */ }

    private void recolour() {
        isRed = true;
        left.isRed = false;
        right.isRed = false;
    }
}
```

A.2 iSet.key – Custom Integer Set

This definition of an integer set type and rules thereon was provided by the KeY developers. We extended it with one additional rule for the equality of sets.

```
\sorts {
    Free; // 'Free' is used for technical reasons and represents an integer set here
}

\functions {
    Free iSet_empty;
    Free iSet_singleton(int);
    Free iSet_minus(Free, Free);
    Free iSet_union(Free, Free);
    Free iSet_intersect(Free, Free);
}

\predicates {
    in(int, Free);
    subseteq(Free, Free);
}

\schemaVariables {
    \term int x, y;
    \term Free setA, setB;
    \variable int iv;
}

\rules {
    inEmpty {
        \find(in(x, iSet_empty))
        \replacewith(false)
        \heuristics(concrete)
    };

    inSingleton {
        \find(in(x, iSet_singleton(y)))
        \replacewith(x = y)
        \heuristics(concrete)
    };

    inSetMinus {
        \find(in(x, iSet_minus(setA, setB)))
        \replacewith(in(x, setA) & !in(x,setB))
        \heuristics(simplify)
    };

    inSetUnion {
        \find(in(x, iSet_union(setA, setB)))
        \replacewith(in(x, setA) | in(x,setB))
        \heuristics(simplify)
    };

    inSetIntersect {
        \find(in(x, iSet_intersect(setA, setB)))
        \replacewith(in(x, setA) & in(x, setB))
        \heuristics(simplify)
    };

    // newly added for this work
    setEq {
        \find(setA = setB)
        \varcond(\notFreeIn(iv, setA, setB))
        \replacewith(\forall iv; (in(iv, setA) <-> in(iv, setB)))
    };
}
```

A.3 Verification Examples

A.3.1 Assertions in rightRotate

```
private void rightRotate() {
    Tree l = left;
    ...

    //@ assert right == \old(left)      // for a prettier heap

    //@ assert height_variant:          heightVariant <= \old(heightVariant) + 1;
    //@ assert is_red:                  isRed == \old(isRed);
    //@ assert right_not_null:          right != null && right.isRed ==
        \old(left.isRed);

    //@ assert left_eq:                 left == \old(left.left);
    //@ assert right_eq:                right == \old(left);
    //@ assert right_left_eq:           right.left == \old(left.right);
    //@ assert right_right_eq:          right.right == \old(right);

    // ---------- footprint ------------
    //@ assert left_footprint:          left == null || left.footprint() ==
        \old(left.left.footprint());
    //@ assert right_left_footprint:    right.left == null ||
        right.left.footprint() == \old(left.right.footprint());
    //@ assert right_right_footprint:   right.right == null ||
        right.right.footprint() == \old(right.footprint());
    //@ assert right_footprint:         right.footprint();
    //@ assert footprint:               footprint() == \old(footprint());

    // ----------- treeSet -------------
    //@ assert left_tree_set:           left == null || left.treeSet() ==
        \old(left.left.treeSet());
    //@ assert right_left_tree_set:     right.left == null || right.left.treeSet()
        == \old(left.right.treeSet());
    //@ assert right_right_tree_set:    right.right == null ||
        right.right.treeSet() == \old(right.treeSet());
    //@ assert right_tree_set:          right.treeSet();
    //@ assert tree_set:                treeSet() == \old(treeSet());

    // ------------ inv ----------------
    //@ assert right_inv:               \invariant_for(right);
    //@ assert inv:                     \invariant_for(this);

    // ---------- double red -----------
    //@ assert left_double_red:         left == null || left.noDoubleRed() ==
        \old(left.left.noDoubleRed());
    //@ assert right_left_double_red:   right.left == null ||
        right.left.noDoubleRed() == \old(left.right.noDoubleRed());
    //@ assert right_right_double_red:  right.right== null ||
        right.right.noDoubleRed() == \old(right.noDoubleRed());
    //@ assert right_double_red:        right.noDoubleRed();
    //@ assert no_double_red:           \old(!isRed && left.doubleRedLeft() &&
        (right == null || !right.isRed && right.noDoubleRed()));
    //@ assert double_red:              \old(doubleRedLeft()) ==> doubleRedRight();

    // --------- black height ----------
    //@ assert right_is_red:            right.isRed && \old(left.isRed);
    //@ assert old_black_heights_same:  \old(blackHeight(left.left) ==
        blackHeight(left.right) && blackHeight(left.right) == blackHeight(right));
    //@ assert left_black_height:       blackHeight(left) ==
        \old(blackHeight(left.left));
    //@ assert right_left_black_height: blackHeight(right.left) ==
        \old(blackHeight(left.right));
    //@ assert right_right_black_height: blackHeight(right.right) ==
        \old(blackHeight(right));
    //@ assert new_black_heights_same:  blackHeight(left) ==
        blackHeight(right.left) && blackHeight(right.left) ==
        blackHeight(right.right);
    //@ assert black_height:            \old(blackHeight(this)) ==
        blackHeight(this);

    //@ assert left_balanced:           left == null || left.blackBalanced() ==
        \old(left.left.blackBalanced());
    //@ assert right_left_balanced:     right.left == null ||
        right.left.blackBalanced() == \old(left.right.blackBalanced());
    //@ assert right_right_balanced:    right.right== null ||
        right.right.blackBalanced() == \old(right.blackBalanced()) ;
    //@ assert right_balanced:          right.blackBalanced();
    //@ assert black_balanced:          blackBalanced();

    // --------------------------------
    //@ assert \dl_assignable(\old(\dl_heap()), \old(footprint()));
}
```

A.3.2 Script for Proving \invariant_for(right) in rightRotate

This script helps to prove that the instance invariant of right holds after a right rotation, and belongs to the assertion in line 31 of A.3.1. It makes use of nested assertions inside the JML Script, which allow for a clearer structure. The heap in the poststate of rightRotate is abbreviated with h here for better readability, but in reality looks like this:

```
heap[self.left := self.left.left]
    [self.right := self.left]
    [self.left.left := self.left.right]
    [self.left.right := self.right]
    [self.key := self.left.key]
    [self.left.key := self.key]
    [self.left.heightVariant := self.heightVariant]
    [self.heightVariant := 1 + self.heightVariant]
```

```
/*@ assert right_inv:              \invariant_for(right) \by {
        expand on="self.<inv>" occ=0;
        expand on="self.left.footprint()" occ=0;
        expand on="self.left.footprint()" occ=0;
        rule unlimit_java_lang_Object__inv_ on="java.lang.Object::<inv>$lmtd(heap,
            Tree::select(heap, self, Tree::$left))";
        rule unlimit_java_lang_Object__inv_ on="java.lang.Object::<inv>$lmtd(heap,
            Tree::select(heap, self, Tree::$right))";
        expand on="self.left.<inv>";
        oss;
        assert "(self.left.right != self.right | self.left.right = null) |
            self.right = null" \by {
            assert "self.right.footprint() != empty" \by {
                assert "self.right != null" \by auto;
                rule notLeft on="self.right != null";
                expand on="self.right.footprint()" occ=0;
                rule equalityToElementOf on="...expanded footprint... = empty";
                auto classAxioms=false steps=1000;
            }
            auto classAxioms=false steps=1000;
        }
        assert "self.left.right = null | self.right.left.<inv>@h" \by {
            rule recall_right_left_eq;
            oss;
            rule applyEq on="self.right.left@h"  occ=1;
            rule unlimit_java_lang_Object__inv_
                on="java.lang.Object::<inv>$lmtd(heap, Tree::select(heap,
                Tree::select(heap, self, Tree::$left), Tree::$right))";
            dependency on="(self.left.right@heap).<inv>@h";
            auto classAxioms=false steps=1000;
        }
```

```
      assert "self.right = null | self.right.right.<inv>@h" \by {
          rule recall_right_right_eq;
          oss;
          rule applyEq on="self.right.right@h" occ=1;
          dependency on="(self.right@heap).<inv>@h";
          auto classAxioms=false steps=1000;
      }
      assert "self.left.right = null | self.right.left.heightVariant@h
              < self.heightVariant" \by {
          assert "self.left != self.left.right" \by auto classAxioms=false;
          assert "self != self.left.right" \by auto classAxioms=false;
          assert "self.left.right.heightVariant =
              self.right.left.heightVariant@h" \by auto classAxioms=false;
          auto classAxioms=false;
      }
      assert "\forall int k_0; self.right.invLessNotInRight(k_0)@h = TRUE" \by {
          leave; // JML Scripts are not powerful enough yet to comfortably
              document manual steps done here
      }
      assert "\forall int k; self.right.invGreaterNotInLeft(k)@h = TRUE" \by {
          leave;
      }
      assert "self.right@h != null" \by auto;
      auto steps=1;
      expand on="self.right.<inv>@h";
      auto classAxioms=false steps=5000;
  }
@*/
```