UNIVERSITY OF TWENTE.

MSc Computer Science

Track: Software Technology

Final thesis

Formal verification of a red-black tree data structure

by

Huu-Minh Nguyen

1895508

March 25, 2019

July 2018 - February 2019

Supervisor:
Prof Dr Marieke Huisman
Dr Sebastiaan Joosten
Dr Stefan Blom

Abstract

Nowadays, although software has been integrated deeply into our society, software errors are still common. Because the failure of software can have devastating effects, being certain that a program does what it is meant to do is crucial. This thesis conducts a case study in deductive verification, which is a sub-area of formal verification. The case study involves a company in the Netherlands and their industrial red-black tree code. This thesis is intended to be an experience report to show how formal verification can be used to help proving the correctness of a program. Ultimately, we want to be able to verify the industrial red-black tree code. However, in this thesis, we only cover the verification of a standard red-black tree code. The main section presents how specifications of a red-black tree can be developed, and the obstacles that are met during the development. Finally, we conclude with the comparisons with the results of other authors and possible future work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and motivation

Over the last decade, technology adoption has been increasing rapidly. Nowadays, we can find software in almost all aspects of our lives. While errors in mobile phone's applications would most likely only cause some annoyances, there are also software errors that could cost millions of euro or cause injuries or even death when going wrong. For example, on August 21st 2018, a shoplifter ran onto the train track into Schiphol tunnel. To avoid accidents, all trains that connected to Schiphol were stopped. Manual instructions were implemented for these trains. However, they conflicted with the automated system, resulting in a software error which rendered the Dynamic Traffic Management system useless. This brought train traffic in the region to a standstill for half a day. Although the economic damage of the incident was not published, it is bound to be high considering that about 52 thousand people were affected. Sometimes, the damage cannot be measured in dollars. On March 18th 2018, a Uber self-driving car crashed into a bicyclist, killing her in the impact. The preliminary report [5] found that the accident was caused by software that was set to ignore trivial objects in the road like a plastic bag. Unfortunately, the software had not categorized the cyclist correctly before it was too late. Because of the critical nature of software, having it run correctly is essential.

When making software, developers expect their program does what it is supposed to do. However, that is not always the case. Mistakes are usually made during the implementation. Some compilers can do type checking at compile time, hence, eliminate certain run-time errors. For example, the statement

```
int[] a = Arrays.copyOf(true, 10);
```

would signal an error at compile time as boolean is not the compatible type of an integer array.

Still, it is impossible for the compiler to know the design decisions or check the compliance of the code to the design decisions. To tackle this problem, formal verification is used [19]. Formal verification is a program analysis technique where detailed design decisions can be formally specified. To support formal verification, many verifiers have been developed. They each have their own underlying techniques and serve different purposes, but in general, if a program verifier deems a program to be correct, then that program's executions do not violate the specified design decisions.

This thesis tries to tackle a part of the verification problems. We focus on the verification of trees, which are widely used data structures in computer science. Their properties make them useful in many search applications where data is constantly entered and removed. For example, they are used in 3D video games to determine what objects need to be rendered, in routers for storing routing tables, in jpeg and mp3 as a part of Huffman compression algorithm, etc. To guarantee a good performance in searching the tree, balancing is needed, as otherwise the tree could degenerate into a list. Red-black trees are particular implementations of self-balancing binary search trees. They seem to be the most popular choice of implementation because of their relatively low complexity across all insert, delete and get operations. Red-black trees are the foundation for set and dictionary implementations in many libraries. For example, the TreeSet and TreeMap classes in Java Core [1] as well as sets [3] and maps [2] in C++. Because of the popularity, being able to formally verify red-black trees is essential in many systems.

1.2 Goal

This research is done in collaboration with BetterBe, a company in Netherlands as a part of the final thesis within the master program of Computer Science at the University of Twente.

This thesis conducts a case study in deductive verification in a company in Netherlands. The study consists of tool selections, approaches and results. It focuses on the red-black tree data structure, which is being used in crucial parts of the company. The correctness of the implementation is thus very important. Formal verification technique is a possible solution to give either a proof of correctness or indication of errors.

The ultimate goal of this research is to verify the BetterBe's red-black tree code. However, due to time constrain, the goal of this thesis is set to verify the standard implementation.

It show the potential of current verification tools and to gain insights on the application process as well as the difficulties of applying the formal verification techniques and the corresponding tools.

1.3 Contributions

The main contributions of this work are proposing a verification method for a red-black tree. There are also other smaller contributions as well:

1. Give insights on the verification process when using VerCors.
2. Show how permissions on tree can be modeled.
3. Provide a verified data structure that can be used in the library.
4. Help finding bottlenecks and flaws in VerCors due to certain encodings.

The thesis is organized as follows: Chapter 1 provides a brief introduction to the problem. Chapter 2 talks about the problem and its challenge; then, comes up with a strategy and explicitly states the research questions to be answered. Chapter 3 goes through the definition of formal verification and the implementation of red-black tree. Chapter 4 discusses a common formal verification language and some of the verifiers for Java. Chapter 5 describes the specification process of a binary search tree and the result. Likewise, Chapter 6 considers the specification process of a red-black tree and the outcome. Then, the similar and related works are reviewed in Chapter 7. Finally, a conclusion and future work is given in Chapter 8.

Chapter 2

Problem definition

2.1 Approach

Because of the recursive nature and having a lot of cases in the implementation, verifying a red-black tree is not easy. To tackle a complex problem, it is best to divide it into multiple simpler problems. A red-black tree at its core is a binary search tree with additional properties. A red-black tree neither change nor remove any properties of a binary search tree. So, a specification of the binary search tree can be transformed to verify some properties of the red-black tree.

There are many implementations of a tree. The standard implementation that can be found in many tutorials uses references to link nodes together to form a tree. Another implementation that is also used by the BetterBe encodes the tree as an array. Because both binary search tree and red-black tree are trees, both implementation methods work for them. So, there are two ways of getting to the BetterBe's red-black tree from the standard binary search tree. We encode the binary search tree as an array first, then we transform it into a red-black tree; or we transform the binary search tree to a red-black tree first, then we encode it as an array. We want to eventually be able to verify the BetterBe's implementation. The red-black tree is typically harder to implement and verify than the binary search tree. So, we took the second path, which involves a standard implementation of red-black tree.

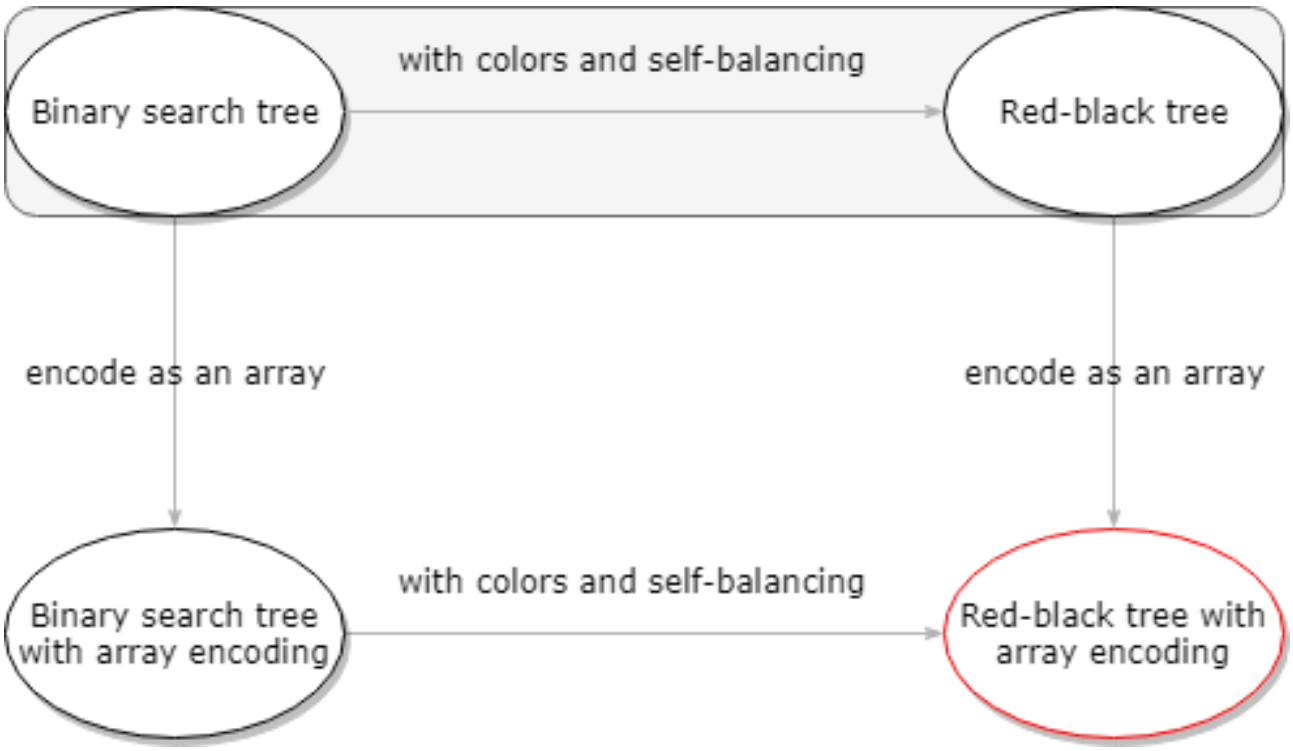

Figure 2-1: Research approach.

A binary search tree is a special tree. By applying the same logic, we can make the problem even smaller. A binary search tree has the exact same structure as a normal tree. So, the memory access permission of a tree should be identical to a binary search tree in concurrent software.

There are 2 differences between a binary search tree and a red-black tree: the colors and the balance. Although red-black tree's self-balancing mechanism depends on the colors, the colors themselves are quite independent. So, the transformation process could be chopped up further.

With that, our research strategy could be summarized as follows: First, we handle the permission of the tree. Then, it is used to provide permission to the binary search tree. After that, we specify the properties of the binary search tree. Next, we implement the red-black tree on top of the binary search tree so that those properties hold. Finally, we specify the properties of the red-black tree. In the future work, we will change the implementation of the specified red-black tree to use the array encoding so that all properties hold. Lastly, we will apply those specification to BetterBe's code.

2.2 Research questions

The following research questions have been formulated:

How can a red-black tree data structure be formally verified?

To know whether the existing tools are sufficient for the verification of red-black trees, we have to gain information about the red-black tree's properties and the tools' capability. There are different implementations of red-black trees, but they share most of their properties. It will help the future verifications if we can transform the specification of an implementation to be usable with another implementation. Therefore, the main research question is supported by 4 sub-questions. These underlining questions have to be answered before we can answer the main questions:

1. What are the properties of a red-black tree? Which set of properties defines a red-black tree?
2. Which tools are suitable for verifying red-black trees?
3. How can the selected tools be used to specify a red-black tree?

   A red-black tree shares the properties with other data structures like a tree, a binary search tree, a colored tree and a balanced binary search tree. This question could be answered easier by asking the same question to those data structures and gradually build up the specification for red-black trees from the specification of those data structures.

   (a) How can these tools be used to specify a tree?
   (b) How can these tools be used to specify a binary search tree?
   (c) How can these tools be used to specify a colored binary search tree?
   (d) How can these tools be used to specify a colored balanced binary search tree?

Research question 1 is first answered during the explanation of red-black tree in chapter 3, and is reminded in chapter 6. Chapter 4 discusses about verification tools and gives the answer for research question 2. The answer for research question 3a and 3b can be found in chapter 5. Lastly, chapter 6 answers research question 3c and 3d.

Chapter 3

Background knowledge

3.1 Formal verification

Formal verification checks for the conformance between the algorithms of a system and certain specifications or properties using mathematics. It is done by giving a formal proof on an abstract mathematical model of the system. The mathematical model has to be constructed to show the behavior of the system. Finite state machines, labelled transition systems, Petri nets, vector addition systems, timed automata, hybrid automata, process algebra, formal semantics of programming languages such as operational semantics, denotational semantics, axiomatic semantics and Hoare logic are frequently used to model systems.

Deductive software verification [13], also known as program proving, is a form of formal verification that is based on Hoare logic. It expresses the intended behavior of a program through a set of mathematical statements called pre-conditions and post-conditions. Pre-conditions and post-conditions together form contracts. Pre-conditions are conditions or predicates that must be true prior to the execution of a section of code or operation. Likewise, post-conditions are conditions or predicates that must be true after the execution. Then, either interactive or automated theorem provers are used to prove the conformance of the code block to these conditions. Those contracts are written in a specification language, which is a formal language used to describe a system at high level. Java Modeling Language is the most common specification language that is tailored to Java.

3.2 Binary search tree

3.2.1 Definition

Binary search trees are sorted data structures. They consist of data nodes linked together, illustrating trees with at most 2 branches. Because they are sorted, they allow fast lookup, addition and removal of items. By definition, binary search trees have the following properties:

1. The key in every node on the left subtree has to be smaller than the key in the current node.
2. The key in every node on the right subtree has to be larger than the current node.
3. The left and right subtrees are also binary search trees.
4. All leaves (final nodes) contain no key.

3.2.2 Implementation

The implementations of binary search trees usually involve recursion to traverse the trees. We traverse the tree by comparing the new key with the current key. If the new key is smaller, we move to the left branch and vice versa.

Insertion

A new node is always inserted at a leaf. First, we traverse the tree until we hit a leaf node. Once we find a leaf node, the new node is added there.

Deletion

First, we traverse the tree to find the node to be deleted. Once it is found, there are 3 cases:

1. It has no child: Remove it from the tree.
2. It has one child: Point the parent directly to the child instead of the node. This action remove it from the tree.
3. It has two children: Find the in-order successor of the node. The successor is the closest node in value to the current node to the right. It is found by finding the smallest node in the right branch of the current node. Copy the contents of the successor to the current node and delete the successor instead.

3.3 Red-black tree

3.3.1 Definition

Red-black trees are one of the implementations of self-balancing binary search trees. The name comes from the red or black colors that is assigned for each of the nodes. By definition, red-black trees have the following properties:

1. The key in every node on the left subtree has to be smaller than the key in the current node.
2. The key in every node on the right subtree has to be larger than the current node.
3. The left and right subtrees are also red-black trees.
4. All leaves (final nodes) contain no key.
5. Each node is either red or black.
6. The root is black.
7. All leaves are black.
8. If a node is red, then both its children are black.
9. Every path from a given node to any of its descendant leaf nodes contains the same number of black nodes.

These properties imply another essential property of red-black trees: *"The path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf"*. As a result, the tree is roughly balanced.

3.3.2 Implementation

The implementations of red-black trees usually use recursion to traverse the trees and a number of rotations to maintain their properties for every call of insert or delete.

Insertion

The insertion of a new node into a red-black tree starts with a normal binary search tree insertion. The newly inserted node is marked as red initially. If both it and its parent are red, we have a double red problem. When this happens, the grandparent cannot be red. If the grandparent is red, the tree before the insertion is not a valid red-black tree. The double red problem is then resolved by recoloring and rotating the nodes. If we call the newly inserted node is x, x's uncle is the other child of x's grandparent. If x does not have a grandparent, x's parent is the root. We can change it into black to remove the double red in this case. We

remain with two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring depending on the position of the double red.

1. If the uncle is red.
    (a) Change the colors of parent and uncle to black.
    (b) Color the grandparent red.
    (c) Repeat for the grandparent: let the grandparent be the new x and check for the double red problem.

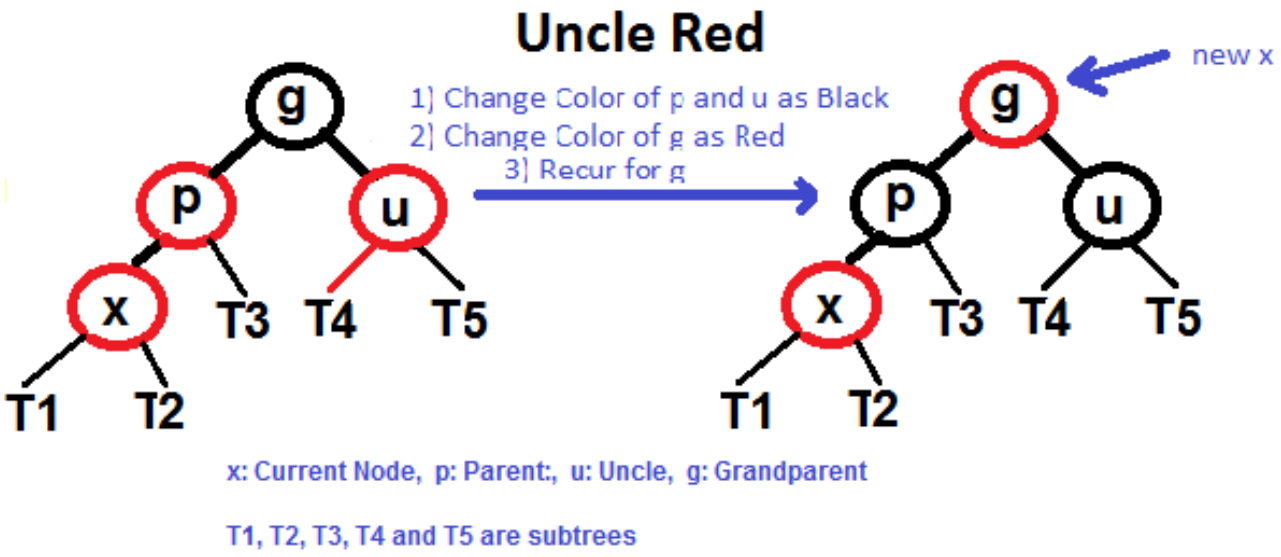

Figure 3-1: Uncle is red.

2. If the uncle is black and x is the left child of parent, whom is the left child of grandparent.
    (a) Right rotate at grandparent. To right rotate at a target, we point the left branch of the target to the right node of the left node of the target. Then, we point the right branch of the original left node of the target to the target itself.
    (b) Swap the colors of grandparent and parent.

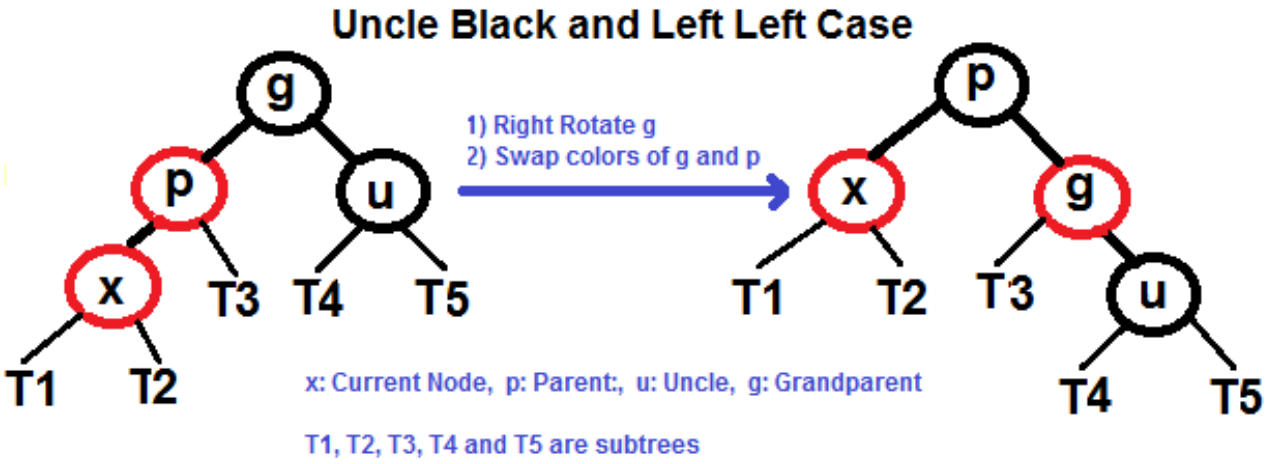

Figure 3-2: Uncle is black. Node is the left child of parent, whom is the left child of grandparent.

3. If the uncle is black and x is the right child of parent, whom is the left child of grandparent.
    (a) Left rotate at parent. To left rotate at a target, we point the right branch of the target to the left node of the right node of the target. Then, we point the left branch of the original right node of the target to the target itself.
    (b) It becomes case 2 now. Repeat the steps of case 2 to resolve this.

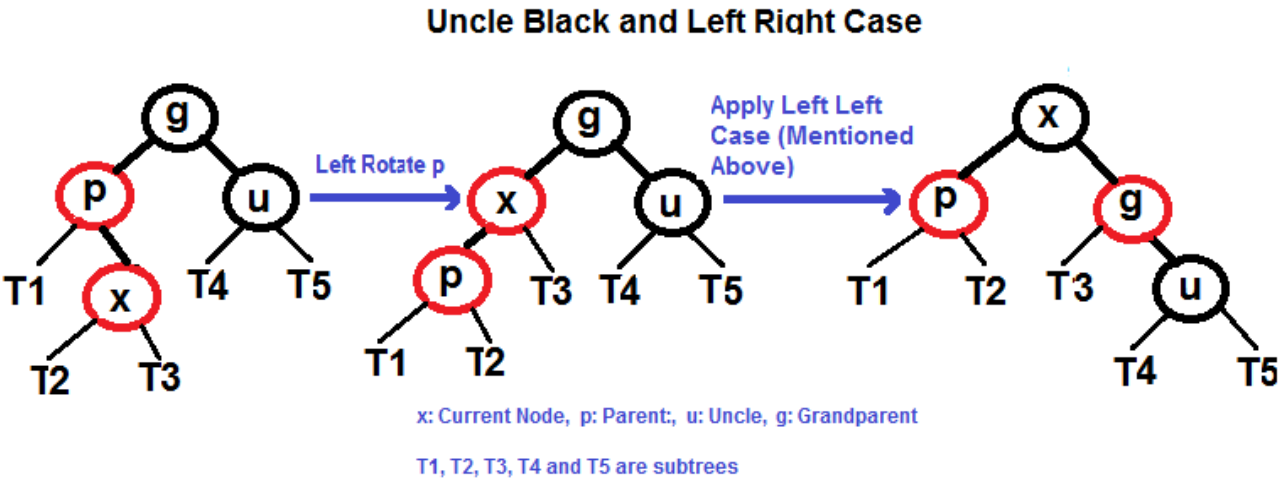

Figure 3-3: Uncle is black. Node is the right child of parent, whom is the left child of grandparent.

4. If the uncle is black and x is the right child of parent, whom is the right child of grandparent.
    (a) Left rotate at grandparent.
    (b) Swap the colors of grandparent and parent.

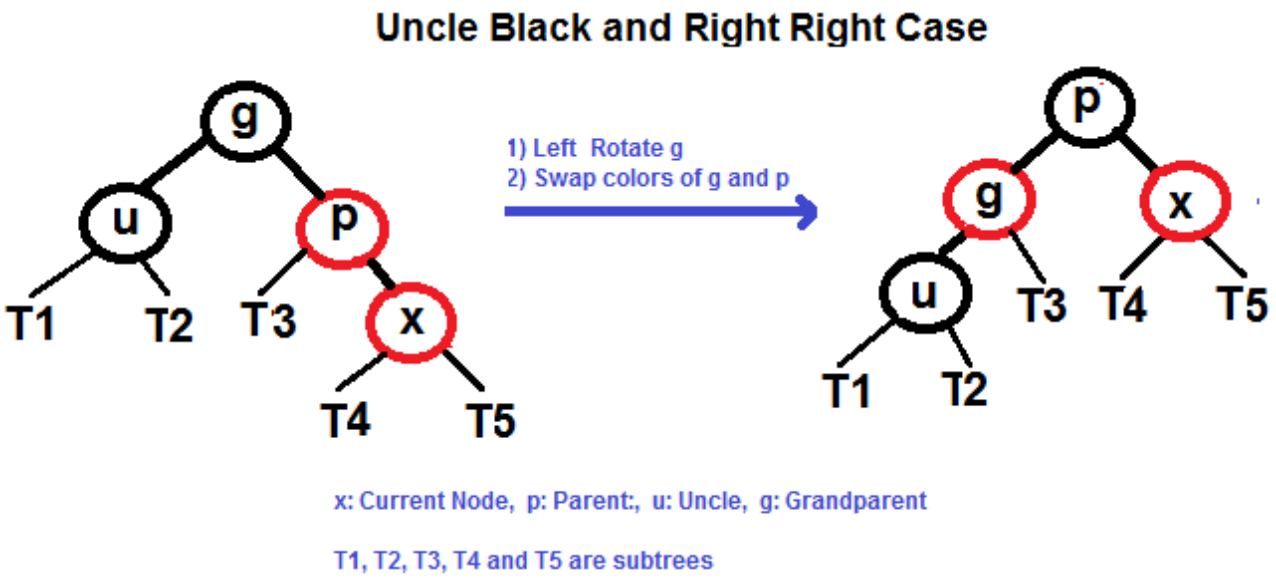

Figure 3-4: Uncle is black. Node is the right child of parent, whom is the right child of grandparent.

5. If the uncle is black and x is the left child of parent, whom is the right child of grandparent.
    (a) Right rotate at parent.
    (b) It becomes case 4 now. Repeat the steps of case 4 to resolve this.

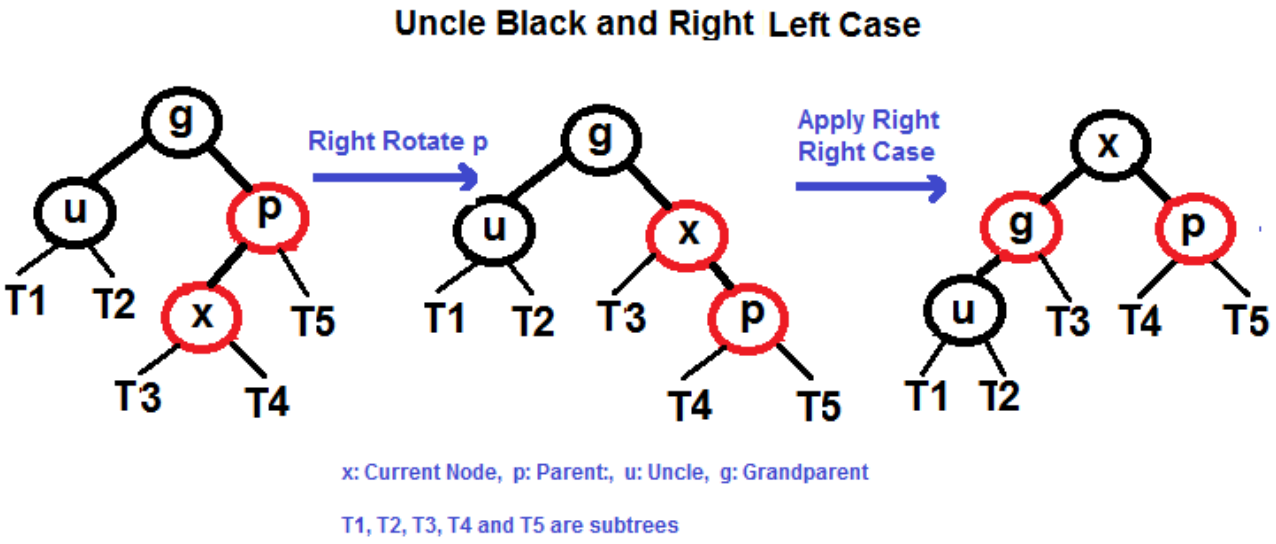

Figure 3-5: Uncle is black. Node is the left child of parent, whom is the right child of grandparent.

Deletion

The deletion of a node in a red-black tree starts with a normal binary search tree deletion. Using this approach, we always delete a node with at most 1 child. Let v be the node to be deleted, u be the child that replaces v, s be the sibling of v and r be the red child of s. If both v

and u are black, we have double black problem. After removing v, we mark u as double black. It means that node u is currently representing black color twice. The double black problem is then resolved by recoloring and rotating the nodes. In delete operation, we check color of sibling to decide the appropriate case. Those two cases are further split depending upon the position of sibling and the color of sibling's children:

1. If the sibling is black and it is left child of its parent and r is left child of s. We do right rotation at parent.

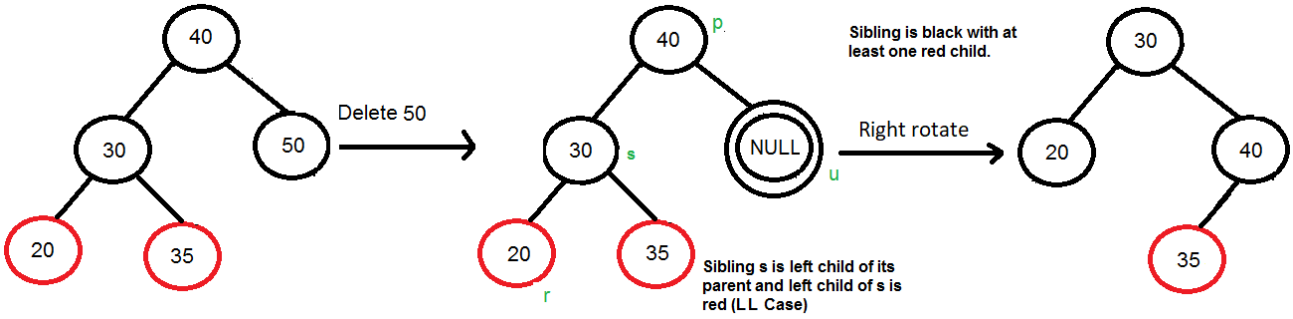

Figure 3-6: Sibling is black. s is left child of its parent and r is left child of s.

2. If the sibling is black and it is right child of its parent and r is right child of s. We do left rotation at parent.

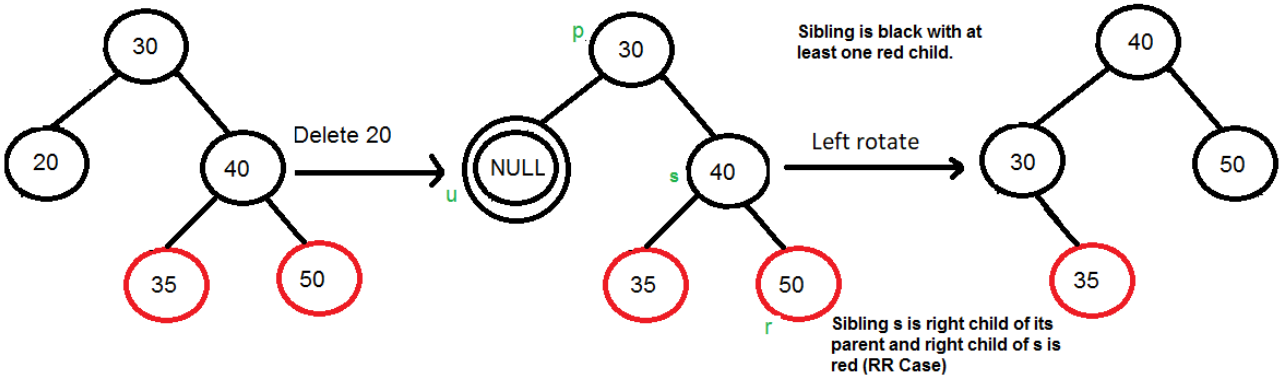

Figure 3-7: Sibling is black. s is right child of its parent and r is right child of s.

3. If the sibling is black and it is left child of its parent and r is right child of s.
   (a) Left rotate at sibling.
   (b) Right rotate at parent.

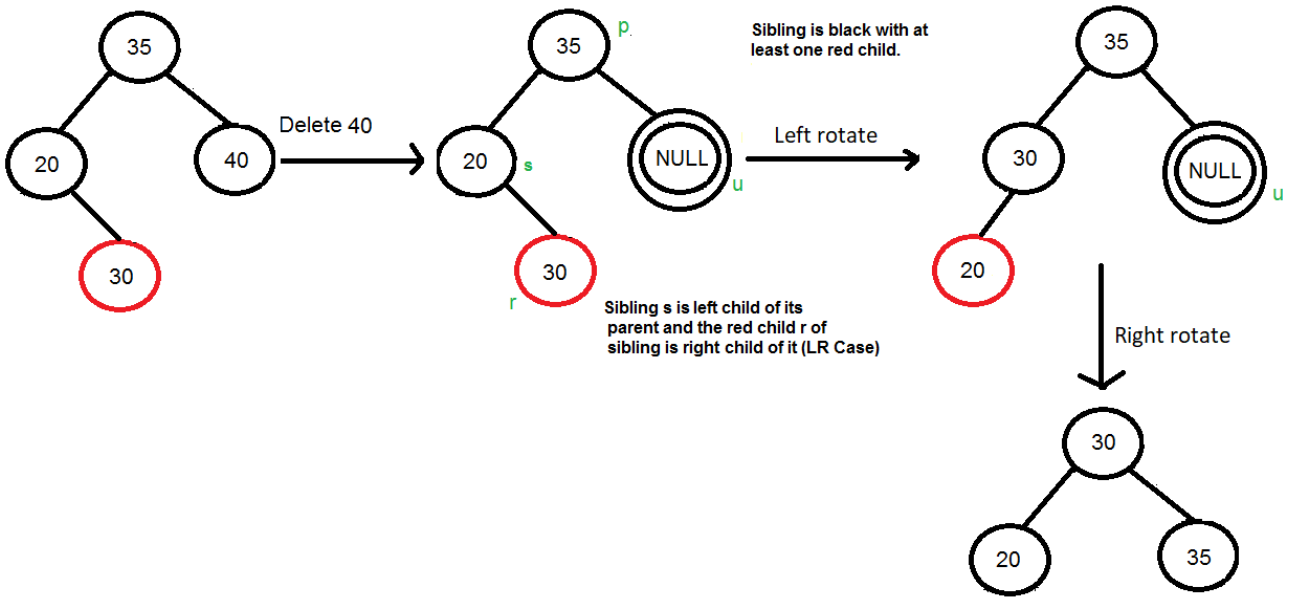

Figure 3-8: Sibling is black. s is left child of its parent and r is right child of s.

4. If the sibling is black and it is right child of its parent and r is left child of s.
   (a) Right rotate at sibling.
   (b) Left rotate at parent.

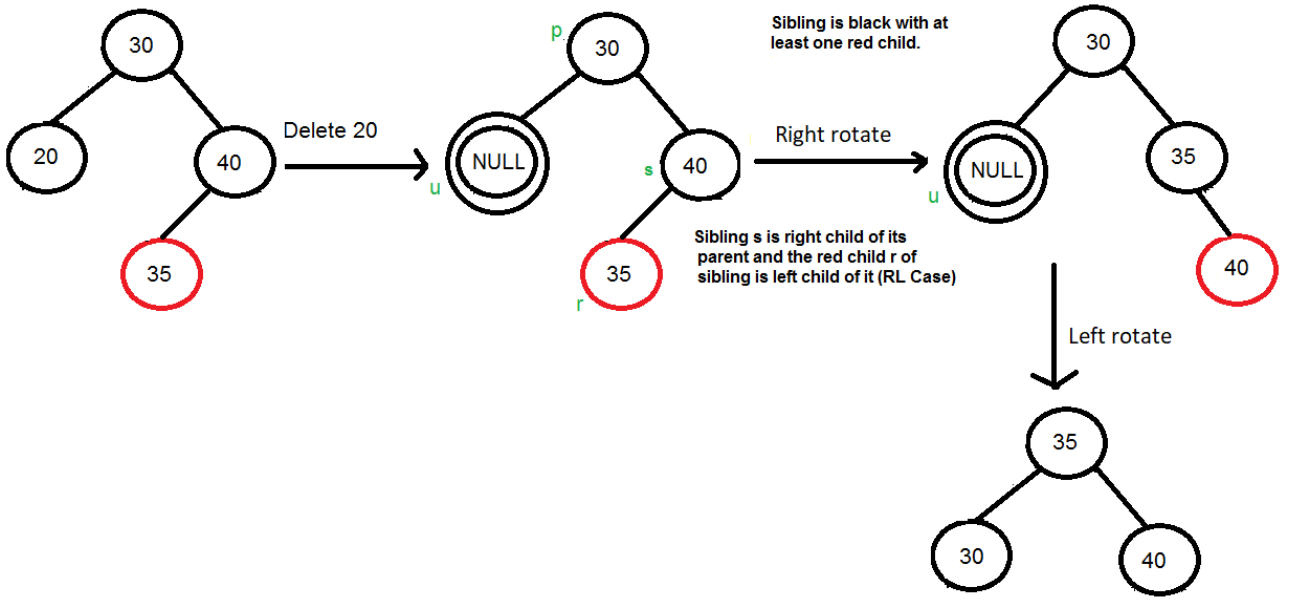

Figure 3-9: Sibling is black. s is right child of its parent and r is left child of s.

5. If the sibling is black and its both children are black.
   (a) Color the sibling red.
   (b) Mark the parent as double black and attempt to fix it by recursing one of the 7 cases.

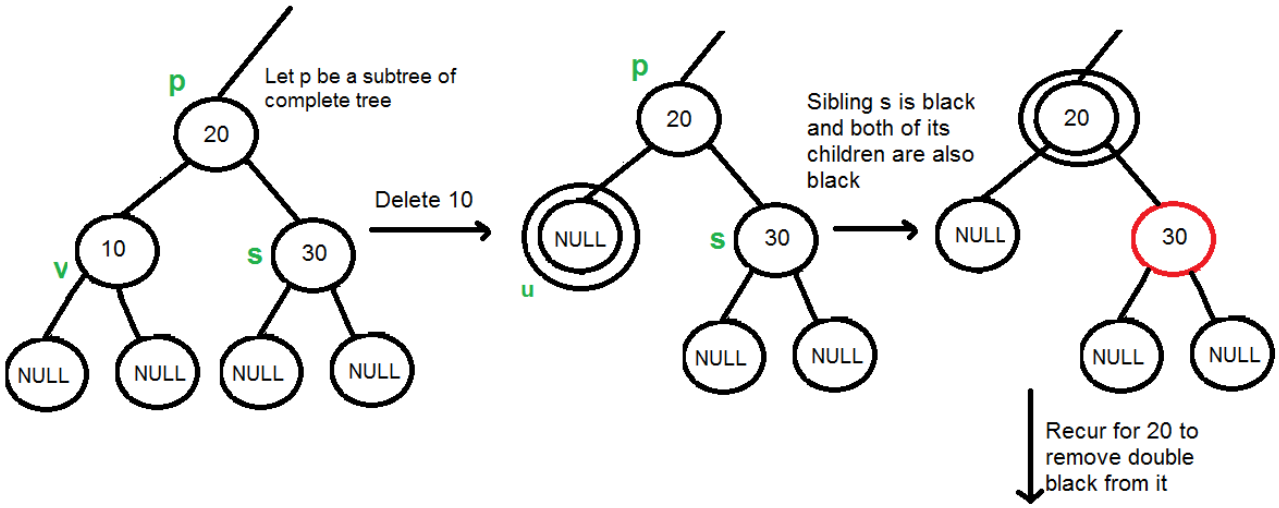

Figure 3-10: Sibling is black and its both children are black.

6. If the sibling is red and it is the left child of its parent.

   (a) Color the parent red.

   (b) Color the sibling black.

   (c) Right rotate at parent.

   (d) It becomes one of the other cases above now. Repeat the appropriate case to resolve this.

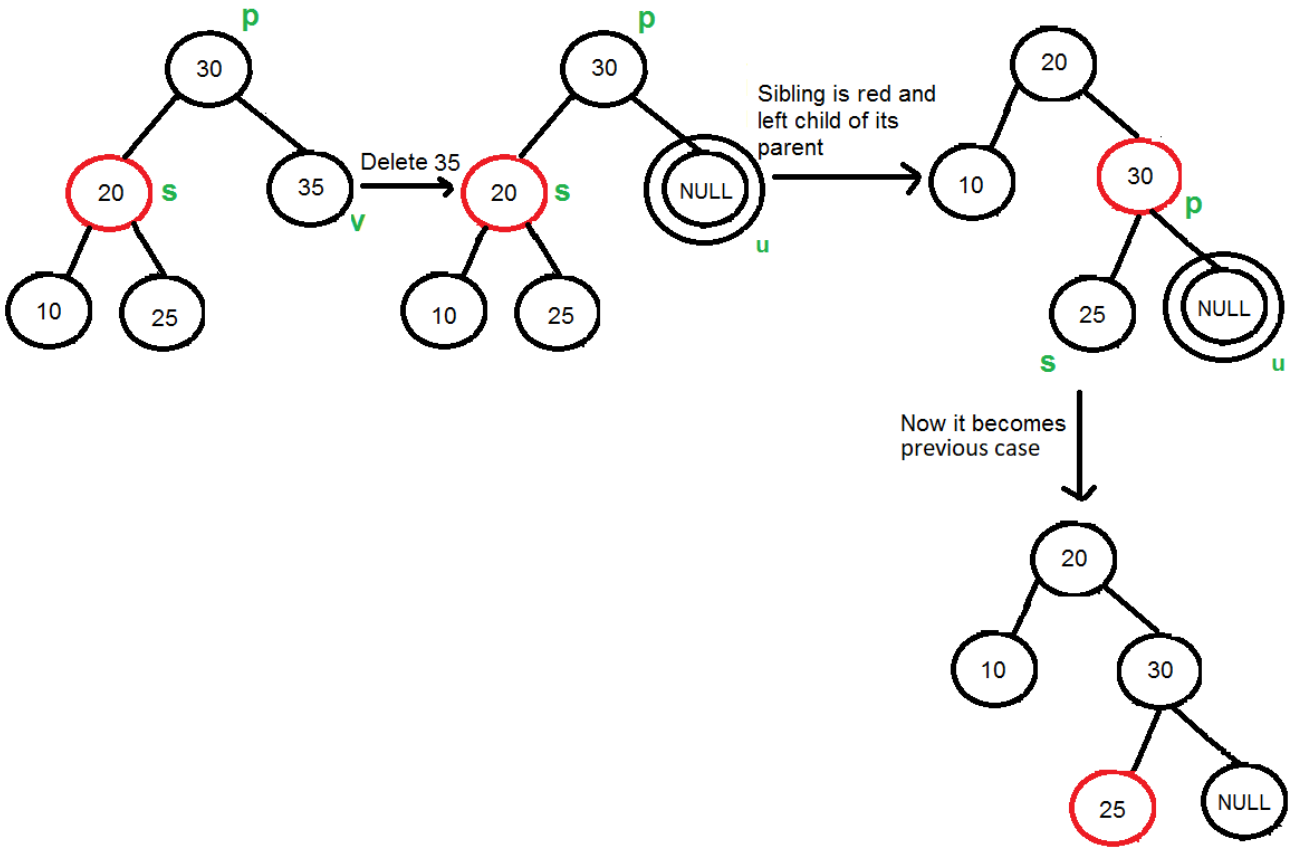

Figure 3-11: Sibling is red and it is the left child of its parent.

7. If the sibling is red and it is the right child of its parent.

   (a) Color the parent red.

   (b) Color the sibling black.

   (c) Left rotate at parent.

   (d) It becomes one of the other cases above now. Repeat the appropriate case to resolve this.

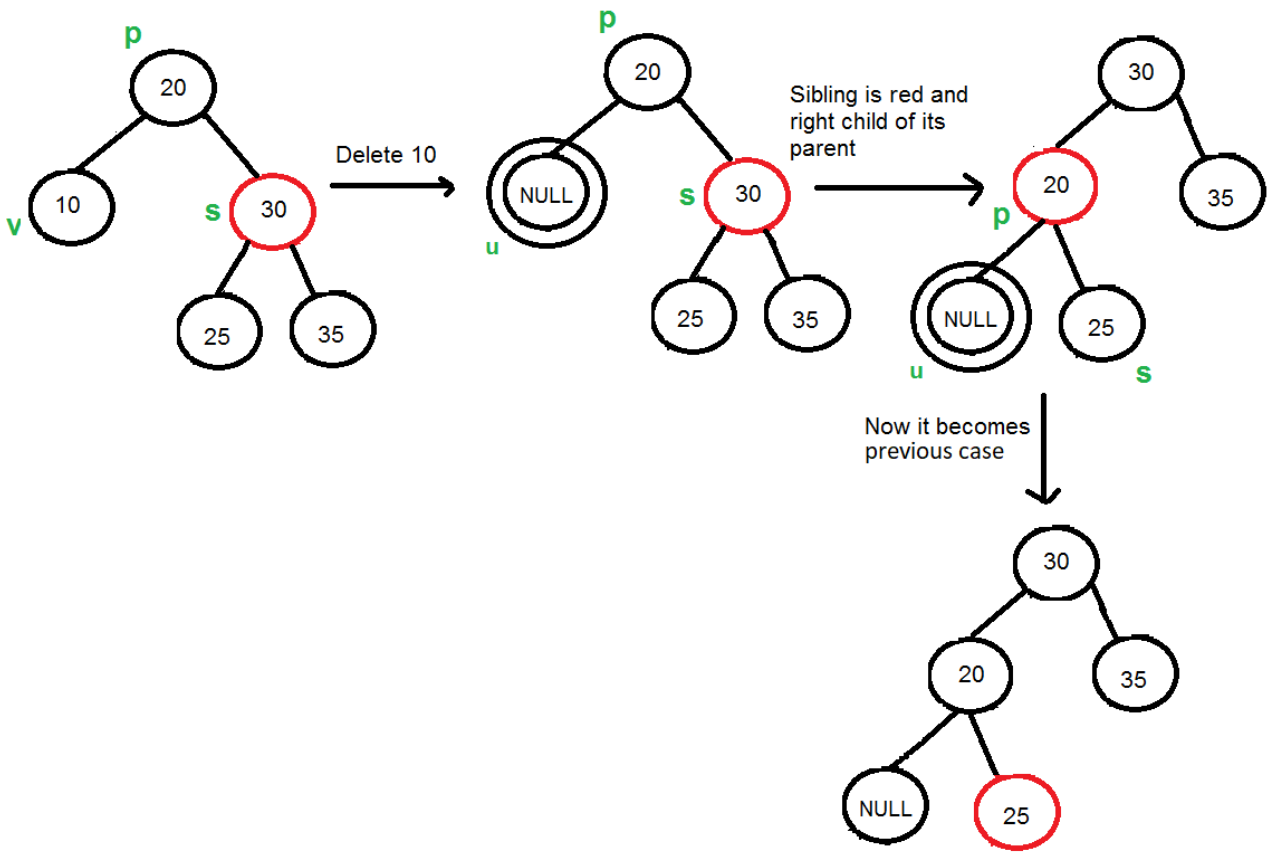

Figure 3-12: Sibling is red and it is the right child of its parent.

Chapter 4

Verifier analysis

Many verifiers, each with their own pros and cons are available. However, there are four tools that officially support the Java programming language and have active communities. They will be discussed in this section. However, firstly, we will look at the specification language that these tools based theirs on.

4.1 Java Modeling Language

Many verifiers use or base their specification language on *Java Modeling Language* (JML) [14]. JML is a behavioral interface specification language that is used to describe the functional behavior of Java classes and methods. It is written between /*@ . . . @*/ or after //@ and shares most of Java's syntax and logical expression. JML also adds its own keywords to support specifying.

A pre-condition or post-condition is written following the keyword "requires" or "ensures". An example is given below:

```
1 class Account {
2    int balance = 0;
3
4    //@ requires balance >= 0 && amount >= 0 && amount <= balance;
5    //@ ensures balance >= 0;
6    void debit(int amount) {
7       balance = amount > 0 && amount <= balance ? balance - amount : balance;
8    }
9 }
```

The debit method receives an integer amount as the input and deducts that amount from the balance. Prior to calling the debit method, both the balance and the input amount have to be larger than 0 and the input amount has to be equal or smaller than the current balance. Right after the method returns, the balance value must be equal or larger than 0. Modular verifiers verify the body of a method by assuming the pre-conditions and asserting the post-conditions at return. They verify the call of a method by asserting the pre-conditions at the call and asserting the post-conditions after the call.

The "invariant" keyword specifies a condition that must be true when entering or exiting any method. For example, "the balance must always be equal or larger than 0" is annotated as:

```
1  class Account {
2     int balance = 0;
3
4     //@ invariant balance >= 0;
5
6     //@ requires amount >= 0 && amount <= balance;
7     void debit(int amount) {
8        balance = amount > 0 && amount <= balance ? balance - amount : balance;
9     }
10 }
```

Likewise, the "loop_invariant" keyword specifies a condition that is true at the beginning and the end of any iteration of a loop. It is an auxiliary keyword to help proving the method specification. Beside keywords, JML also extends Java with a few operators, namely "\old", "\result", "\forall", "\exist". The operators "\old" and "\result" can only be used in a post-condition. "\old" refers to the value of a variable before entering the method and "\result" refers to the returning value of the method. The operators "\forall" and "\exist" behave like universal quantifications in predicate logic. The syntax is (\forall <var-declaration>; <var-boundary>; <expression>). It can be described as "for all of the variables in <var-declaration> that get the value within <var-boundary>, the <expression> is true". The same syntax is applied for "\exist". For example, to say the returning value is the largest number in the array, we annotate as:

```
1  class Math {
2     int[] a = {10,20,30,40,50,60,71,80,90,91};
```

```
3
4     //@ ensures (\forall int i; i >= 0 && i < 10; a[i] <= \result);
5     //@ ensures (\exist int i; i >= 0 && i < 10; a[i] = \result);
6     int max() {
7        int max = a[0];
8
9        //@ invariant i >= 0;
10       //@ invariant i <= 10;
11       //@ invariant (\forall int j; j >= 0 && j < i; a[j] <= max);
12       //@ invariant (\exist int j; j >= 0 && j < i; a[j] = max);
13       for (int i = 0; i < 10; i++) {
14          if (a[i] > max) {
15             max = a[i];
16          }
17       }
18
19       return max;
20    }
21 }
```

4.2 OpenJML

4.2.1 Technology

OpenJML [4] is a set of tools that can be used to type-check, static check and run-time check Java source code of sequential software. The static checking is fully automated. From the annotations, the static checker generates verification conditions and then sends them to an underlying first-order logic prover. OpenJML supports major Satisfiability Modulo Theories (SMT) solvers such as Z3, CVC4 and Yices. The success of the proofs will depend on the capability of the SMT solver and the complexity of the code and specifications. OpenJML can be run both from command line and as an Eclipse plugin. To do that, the source code must be annotated with JML statements recording the design and implementation decisions.

4.2.2 Documentation

OpenJML is documented very well. In its website, there are links to a user guide and a reference manual, which present the syntaxes and meanings of all supported keywords. It also has a page which lists papers that are related to OpenJML and JML. Example verifications can be found and run online. However, only basic samples are available.

4.3 KeY

4.3.1 Technology

KeY [10] is a deductive verifier for sequential Java and JavaCard applications. It can be run from Eclipse or as a standalone Graphical User Interface (GUI) application. It allows to prove the correctness of programs with respect to a given specification. KeY relies on a first-order theorem prover based on sequent calculus to close the proof. Sequent calculus takes the form $A_1, ..., A_n \vdash B_1, ..., B_k$ and is a generalization of natural deduction judgment. KeY also use JML to annotate the specification like OpenJML. Although they both agree on using JML, there are major differences between them as confirmed by Jan Boerman, Marieke Huisman and Sebastiaan Joosten in an Integrated Formal Methods paper [9]. While OpenJML is fully automated, KeY is an interactive program verifier. That means that the responsibility for finding a proof lies with the user. The core of KeY is based on dynamic logic, which can be viewed as a further development of Hoare Logic to a modal logic. A Hoare triple $\{\phi\}p\{\psi\}$ can be expressed as dynamic logic formula $\phi \rightarrow [p]\psi$. However, a set of dynamic logic formulas is closed under the usual logical operators [8].

4.3.2 Documentation

KeY's documentation lies mostly in the published papers and books. Unfortunately, there is not a central place where most information about KeY can be found. However, there are a lot of example verifications in the KeY download package, especially examples about JavaCard applications.

4.4 VeriFast

4.4.1 Technology

VeriFast [15] is a program verifier based on separation logic for verification of C and Java programs. Separation logic [18] is an extension of Hoare logic. It describes "states" comprising a store and a heap, which are comparable to the state of local variables and dynamic objects in C and Java. The current VeriFast implementation uses the Z3 SMT solver as the underlying technology. The paper "VeriFast: Imperative Programs as Proofs" [16] claims that both automated and interactive program verifiers have their weaknesses. So, VeriFast takes the hybrid approach by including proof steps in the program text, and using a proof formalism where proofs look exactly like imperative programs. Although VeriFast does not have integration with Eclipse, it has its own graphical IDE besides the command line tool. Unlike OpenJML and KeY, which only support sequential software, VeriFast supports both single-threaded and multi-threaded programs. If "0 errors found" is reported by the tool, it means that the implementation does not violate the pre-conditions, post-conditions, invariants and other specification annotations. Furthermore, it also checks for null pointer dereferences, out of bounds array indexes, arithmetic overflow, divisions by zero and data races.

In VeriFast, the method's body is verified by symbolic execution, starting from the pre-condition, checking for permission, changing the symbolic state according to the method's body and ending with checking the final state against the post-condition. Symbolic execution is an ordinary execution with concrete values replaced by symbolic values instead. When it meets with an "if statement", the symbolic execution forks into two branches of the "if statement". The use of symbols ensures that all possible cases are covered at the cost of possible path explosion, especially in large programs.

VeriFast uses its own annotations. However, they are very similar to JML. In fact, the syntax of pre-conditions and post-conditions is almost the same. The only difference is their position. In JML, the contracts are written above the method declaration, but VeriFast writes them below it.

```
1 class Account {
2   int balance;
3
4   void debit(int amount)
5   //@ requires balance >= 0 &*& amount >= 0 &*& amount <= balance;
```

```
6     //@ ensures balance >= 0;
7     {
8        balance = amount > 0 && amount <= balance ? balance - amount : balance;
9     }
10 }
```

Note that the "&*&" operator is the separating conjunction operator in separation logic [18] and is used to combine pre-conditions or post-conditions together. It says that the heap can be divided into two non-overlapping sets where left-hand argument hold in one set and right-hand argument hold in the other set. VeriFast does not have class invariants. The "invariant" keyword is used for loop invariants instead. Likewise, the loop invariant is written below the loop declaration in VeriFast.

VeriFast tackles the concurrency problem by using separation logic's permissions [20]. A method needs permission to access a memory location. Having the permission to access field f of object o and o.f is having value v is denoted as following: "o.f |-> v". Sometimes, there is a question mark before the variable v (?v). This indicates that there is no restriction on the value of o.f, but it is bound to symbol v. It can be said that o.f is having an unknown value v. VeriFast also supports predicates. A predicate is an assertion with a name and parameters. An example is given below:

```
1 //@ predicate account(Account a, int i) = a.balance |-> i &*& i >= 0;
```

In this example, the assertion "account(a1, amount)" is equivalent to the assertion "a1.balance |-> amount &*& amount >= 0".

4.4.2 Documentation

Like OpenJML, VeriFast handles their documentation very well. There are tutorials for both the C and Java language. The tutorials are very detail with clear explanations. The example verifications that come with the download package are abundant and have many interesting cases.

4.5 VerCors

4.5.1 Technology

VerCors [6] is the automated program verifier developed at the University of Twente. It is inspired by VeriFast, but is specialized for reasoning about concurrent and parallel software. At the core, it is realized by a series of model transformation. The first input is Java with JML-like specifications and separation logic access permission. Then, the input is transformed into the intermediate verification language Silver, which is designed and used by Viper (Verification Infrastructure for Permission-based Reasoning) toolset. Finally, this intermediate language is fed as inputs to Viper, which is powered by the Z3 theorem prover. Viper is the verification infrastructure that provide an architecture for quickly developing verification tools and prototypes. It includes the Silver language and automatic verifiers for it. The Viper toolset can be used to implement verification techniques for front-end programming languages via translations into the Viper language. It allows reasoning about programs with persistent mutable state and supports reasoning about the program state in separation logic style permission [17].

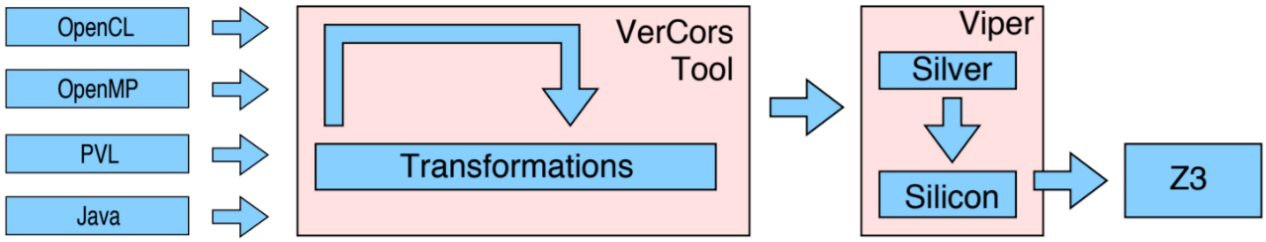

Figure 4-1: The workflow of the VerCors toolset.

VerCors supports JML-like annotations in Java. The keywords "requires", "ensures", "invariant", "loop_invariant" and "\old", "\result" have the same syntaxes and behave pretty much identical to OpenJML. In addition to those, VerCors also introduces new keywords. For example, "context" is the syntactic sugar for a combination of pre-condition and post-condition.

```
1 class Account {
2   int balance = 0;
3
4   //@ requires balance >= 0;
5   //@ ensures balance >= 0;
6   int getBalance() {
7     return balance;
8   }
```

```
}
```

is equivalent to

```
class Account {
   int balance = 0;

   //@ context balance >= 0;
   int getBalance() {
      return balance;
   }
}
```

The concurrency problem and the memory access are handled by separation logic written in JML style specification, which has the syntax "Perm(object, frac);". It accepts two parameters. The first parameter is the object that permission would be granted to. The second parameter is a fraction between 0 and 1 inclusive. If the fraction is 1, it means that write and read permission would be granted to the object. Any fraction strictly smaller than 1 and larger than 0 means that only a read permission would be granted. For an object, at any point, the sum of all active permissions cannot be larger than 1. With this, the write permission is ensured to be unique during the execution, while the read permission could be granted multiple times. The permission of the same object can be combined as followings:

```
//@ context Perm(a, 0.3) ** Perm(a, 0.6);
```

is equivalent to

```
//@ context Perm(a, 0.3 + 0.6);
```

Note that "**" operator is VerCors' syntax of separating conjunction operator in separation logic. It is used to combine multiple pre-conditions or post-conditions like VeriFast's "&*&" operator. It means that the heap can be divided into two non-overlapping sets where left-hand argument hold in one set and right-hand argument hold in the other set.

Like VeriFast, VerCors also supports predicates. Predicates in VerCors are side-effect-free functions that have resource return type. "resource" is a primitive type exclusive to VerCors that is accepted as a condition. Because predicates are functions, recursion is possible, which greatly increases the flexibility of the feature. An example of recursion in predicate is given

below:

```
1 //@ requires n != null ** Perm(n, 1);
2 /*@ public resource llist(Node n) = Perm(n.val, 1) ** n.next != null ==>
    Perm(n.next, 1) ** llist(n.next); @*/
```

Along with the predicates are the "fold" and "unfold" keywords. Technically, "unfold" consumes a predicate and gives permissions, assertions and other predicates, which are specified inside it. "fold" does the opposite. It checks whether everything required by a predicate are available. If they are, it consumes those things and give back the predicate. Note that VerCors does not know what is inside a predicate until it is unfolded.

Beside "frac" and "resource", VerCors also introduces other primitive types. For example, "seq<E>" represents a sequence of type E. Likewise, "set<E>" represents a set of type E and "bag<E>" represents a multiset or bag of type E. However, these types are only available in the specifications and do not exist in the actual code.

Currently, VerCors is called from the command line and supports programs that are written in C, Java, OpenCL and OpenMP. However, in its current state, it works best when being used along with its own prototype language PVL. PVL stand for Prototypal Verification Language. It is a Java-like language that accepts specifications as valid code. So, the specifications do not have to be written in the comments. That allows for deeper integration between the specifications and the program.

4.5.2 Documentation

VerCors' documentation consists of a user guide, published papers and examples. The user guide provides a nice starting point, but it does not cover all functionalities. Fortunately, the papers give a lot of insight into them. VerCors also has a respectable amount of examples in its GitHub repository. There are simple verifications as well as the more complex ones.

4.6 Summary

Table 4.1 below summarizes the differences between four tools.

	OpenJML	**KeY**	**VeriFast**	**VerCors**
Support language	Java	Java	Java, C	Java, C, OpenCL, OpenMP and PVL
Target program	Sequential	Sequential	Concurrent	Concurrent
Specification language	JML	JML	Custom built	Custom built
Verification type	Automated	Interactive	Hybrid	Automated
Logic	First-order	Dynamic	Separation	Separation
Underlying solver	Z3, CVC4, Yices 2	Custom built	Z3	Z3
Packaging	Command line application and Eclipse plugin	GUI application and Eclipse plugin	Command line application and dedicated IDE	Command line application
User guide	Fully documented	Partially documented	Fully documented	Partially documented
Example	Simple	Simple	Both simple and complex	Both simple and complex

Table 4.1: Verifier comparison.

All in all, each tool has its own strength and weakness and serves different purposes. For the basis of this thesis, VerCors is the most appropriate. It utilizes separation logic. According to Bart Jacobs in "VeriFast: Imperative Programs as Proofs" [16], first-order logic and dynamic logic approaches are subjected to mediocre performance and predictability. The culprit is the huge amount of universally quantified premises put in the verification conditions to handle the framing of heap effects. SMT solvers' performance is unstable in these cases. If rich properties are needed, the performance suffers even more due to not having inductive datatypes or fixpoint functions. For example, quantifications are required over indices of an array or the elements of a set. VeriFast and VerCors are comparable in this aspect. However, it is easier to get support from the VerCors development team because of being a project at the University of Twente.

Chapter 5

Binary search tree specification

5.1 Overview

In this chapter, we talk about the verification process of a binary search tree. A binary search tree is a binary tree with one property, which is: "every node content in the left side is equal or smaller than the current node content and every node content in the right side is equal or larger than the current node content". However, because Vercors deals with concurrent software, we require the permissions to be able to access the memory. So, we want to have correct permissions before dealing with the binary search tree property.

Because we are dealing with a tree structure, it would be reasonable to expect the permissions to have the same structure. Specifically, we want the permissions of the children to be embedded into the permissions of the parent. There are many ways to implement this. In this project, we chose to do this recursively because it resembles how the binary search tree is implemented.

The biggest issue in our permission is that we do not always want full tree permission. For example, the getMin method returns the smallest node in a binary search tree. When the method returns, we expect write permission on both the returned node and the original tree. However, since the returned node is a part of the original tree, we cannot have a separate write permission of the same object at the same time. We could solve this issue by returning a copy of the node instead of the node itself. Because the returning node is not the same node as the one in the tree, it can have its own permission. This approach was dropped because we thought it was a band-aid rather than an actual solution. A typical implementation returns the required object instead of a copy. If we change the implementation to return a copy, it would not a standard implementation of binary search tree anymore. Instead, we decided to take away the

permission of the result that is embedded inside the main tree. To make that possible, we have to give our tree permission the ability to exclude a part of a tree. We called this the permission of a "tree with a hole". Because the permission of the returned result has been excluded from the tree permission, we can give the result of getMin method that permission separately as an independent tree.

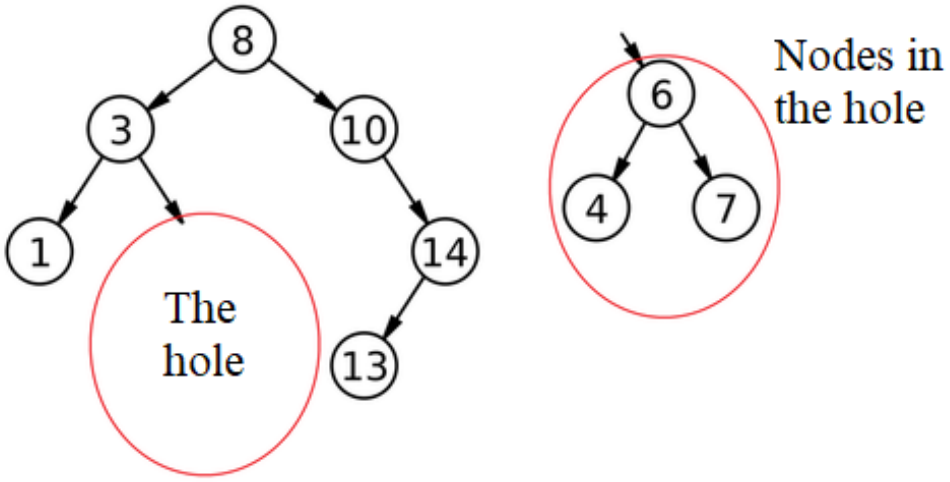

Figure 5-1: Tree with a hole and the hole's content.

To express the permission of a "tree with a hole", we have the "tree_perm_except" predicate. It accepts 2 parameters. The first one is the root of a tree and the second one is the node that we want to exclude. The predicate gives all permissions of the current node and calls itself for the left and right branches. It will do nothing and return when it reaches the end of a branch or it meets the excluded node. Excluding null means excluding nothing. To improve the readability of the code, we have the "tree_perm" predicate. It calls the "tree_perm_except" predicate with null as the second parameter.

```
1  final class Tree {
2     Node root;
3
4     /*@
5        public resource tree_perm_except(Node current, Node node) =
6           current != null ==>
7              (current != node ==>
8                 Perm(current.key, 1) ** Perm(current.left, 1) ** Perm(current.right, 1)
9                    ** tree_perm_except(current.left, node)
10                      ** tree_perm_except(current.right, node));
11    @*/
```

```
12
13    //@ public inline resource tree_perm(Node current) = tree_perm_except(current,
          null);
14 }
```

The "inline" keyword can be seen as doing a text replace before compiling. By utilizing it, we can have predicates with different names, but they are considered as the same predicate by the verifier.

5.2 Node specification

Specifying the Node class is straightforward and self-explanatory. Basically, we have a key containing data, a pointer to the left branch and a pointer to the right branch. The constructor returns the full permission of everything in that node and initializes all pointers to null.

```
1  final class Node {
2     int key;
3     Node left;
4     Node right;
5
6     //@ ensures Perm(key, 1) ** Perm(left, 1) ** Perm(right, 1);
7     //@ ensures key == item ** left == null ** right == null;
8     Node(int item) {
9        key = item;
10       left = null;
11       right = null;
12    }
13 }
```

5.3 Tree properties

Apart from having the structure of a tree, there is only one property that is enforced in a binary search tree: *for every node in the tree, all nodes in the left branch are equal or smaller than the current node and all nodes in the right branch are equal or larger than the current node.* To verify this, we must know the content of "all nodes". Going into each node to check

the contents is rather tedious. So, we wrote a helper that takes the first node of a tree along with the permission and return the contents of all nodes in that tree in a bag. A bag is a data structure of VerCors. It is a container that represents a multiset, which is similar to a set but allows duplicated values. The helper is a pure function called "to_bag". The "to_bag_except" also exists for the same reason as "tree_perm_except". A pure function is a function that has the same enter and exit state. Pure functions ensure that nothing is changed during their evaluation. By using pure functions to compute a result, we do not modify the current state. So, everything that is true before the call remains true and vice-versa. So, the pre-conditions can always be asserted in the post-condition.

```
1  final class Tree {
2     Node root;
3
4     /*@
5        requires [read]tree_perm_except(current, node);
6        public pure bag<int> to_bag_except(Node current, Node node) =
7           current != null && current != node
8              ? \unfolding [read]tree_perm_except(current, node) \in
9                 (to_bag_except(current.left, node) +
10                   bag<int> { current.key } +
11                      to_bag_except(current.right, node))
12             : bag<int> { };
13    @*/
14
15    //@ requires [read]tree_perm(current);
16    //@ public pure inline bag<int> to_bag(Node current) = to_bag_except(current,
          null);
17 }
```

The "\unfolding .. \in" keyword is equivalent to unfold then fold. [read] is a multiplier. If we put a multiplier before a predicate, that multiplier will apply to everything in the predicate. We do not want to give any method more permission than it requires. By not giving full write permission, we can ensure that the method change nothing. Because they are pure functions, we only need read permission on the tree instead of full permission.

We still have to do the comparison of "all nodes", which is in a bag now. A few other helpers were written for this, comparing bag and integer or comparing two bags. The "smaller"

and "larger" function take a bag and an integer. They check if everything in the bag is equal or smaller / equal or larger than the integer. We only need to implement either "smaller" or "larger" to compare two bags because we can simply swap the parameters to get the other.

```
final class Tree {
   Node root;

   //@ public pure boolean smaller(bag<int> b, int max) = (\forall int i; (i
       \memberof b) != 0; i <= max);

   //@ public pure boolean larger(bag<int> b, int min) = (\forall int i; (i
       \memberof b) != 0; i >= min);
   //@ public pure boolean larger(bag<int> b1, bag<int> b2) = (\forall int i; (i
       \memberof b2) != 0; larger(b1, i));
}
```

"(i \memberof b)" returns 0 if b does not contain i and returns another integer otherwise.

With those helpers, the binary search tree property is expressed as following:

```
final class Tree {
   Node root;

   /*@
      requires [read]tree_perm_except(current, node);
      public pure boolean sorted_except(Node current, Node node) =
         current != null && current != node
            ? \unfolding [read]tree_perm_except(current, node) \in
               (smaller(to_bag_except(current.left, node), current.key)
                  && larger(to_bag_except(current.right, node), current.key)
                     && sorted_except(current.left, node)
                        && sorted_except(current.right, node))
            : true;
   @*/

   //@ requires [read]tree_perm(current);
   //@ public pure inline boolean sorted(Node current) = sorted_except(current,
       null);
```

```
18 }
```

The property check whether the content of every node in the left is equal or smaller than the content of current node, and the content of every node in the right is equal or larger than the content of current node. Then, it recursively call itself for all nodes in the tree.

5.4 Insert function

For the insert function, we want to ensure that we have full permission of the tree and it is "sorted" both before and after the insertion. This as well as many other functions need full write permission of the whole tree to execute. Giving them the write permission means anything and everything can be changed, especially the content. This is dangerous because everything that is true before the call might not be true anymore afterward. We need a way to control the changes in the tree's content. In this insert function, the content of the tree after the insertion is equal to the content before plus the newly inserted key. We cannot get the old content using "\old" in the post-condition because nodes are just addresses, which stay the same. We need to extract the content of the tree in the pre-condition and store it to compare with the content of the tree in the post-condition. We had the helper function to extract the content to a bag already, and in VerCors, a bag is immutable. So, we only need to store this bag. We used a ghost parameter for this purpose. Ghost parameters are the parameters that can be seen and used by the verifier only. They are declared by the "given" keyword and get their value from the "with" keyword. This ghost parameter B can be seen in all functions that require full permission, even if nothing is changed. It is our way to control the content of a tree in a function.

```
1  final class Tree {
2     Node root;
3
4     //@ context Perm(root, 1) ** tree_perm(root) ** sorted(root);
5     void insert(int key) {
6        root = insertRec(root, key) /*@ with { B = to_bag(root); } @*/;
7     }
8
9     //@ given bag<int> B;
10    //@ requires tree_perm(current) ** sorted(current);
```

```
11    //@ requires B == to_bag(current);
12    //@ ensures \result != null ** tree_perm(\result) ** sorted(\result);
13    //@ ensures to_bag(\result) == B + bag<int> { key };
14    Node insertRec(Node current, int key) {
15       if (current == null) {
16          Node result = new Node(key);
17          //@ fold tree_perm(result.left);
18          //@ fold tree_perm(result.right);
19          //@ fold tree_perm(result);
20          return result;
21       }
22
23       //@ unfold tree_perm(current);
24       if (key <= current.key) {
25          current.left = insertRec(current.left, key) /*@ with { B =
                to_bag(current.left); } @*/;
26       } else {
27          current.right = insertRec(current.right, key) /*@ with { B =
                to_bag(current.right); } @*/;
28       }
29
30       //@ fold tree_perm(current);
31       return current;
32    }
33 }
```

Line 12 says that the result cannot be null, because it must has at least one newly added node. Furthermore, we have the permission of the result and the result is a sorted tree. Line 13 reasons about the content of the tree. The insert function add a new node into the tree. So, the content of the result must be equal to the content of the original tree plus the content of the newly added node. The new node is created at line 16. The constructor of the Node gives us all permissions of that node. However, to match the post-condition, those permissions must be folded in a "tree_perm". Once the right annotation is given, the verified can verify everything on its own.

5.5 Min function

Although we removed the permission of the result from the tree permission, the resulting node is still a part of the tree. Changes in it could violate the validity of the whole binary search tree and the tree would not fully function without all permissions. We have to have the ability to reattach two tree together when needed. To fit a tree into a hole in another tree, obviously, both trees must have the property of a binary search tree. This property applies for all nodes in both trees, except for the node at the joint, because we split the permission of the hole out of the main tree, the main tree cannot access the content of the hole to check for binary search tree property. We need a property to keep track of whether it can fit into the hole and we can combine them into a full, complete tree. By being a tree, a binary search tree has no shared nodes in its structure. Therefore, by looking at the Figure 5-1, we can see that if the hole is in the left branch, then it would not be in the right branch and vice versa. Furthermore, if the hole is in the left branch, its contents must be equal or smaller than the current node; if the hole is in the right branch, its contents must be equal or larger than the current node. This property should be preserved as we go down the tree, until we find the hole. Although we cannot see the contents of the hole, we can still see the address. So, we would know when we are at the hole.

A node is not in a tree when we cannot find it in the tree. The "in_tree" function is a pure function that checks whether a node exists in a tree.

```
1  final class Tree {
2     Node root;
3
4     /*@
5        requires [read]tree_perm_except(current, node);
6        public pure boolean in_tree(Node current, Node node) =
7           current != null
8              ? node != null && current != node
9                 ? \unfolding [read]tree_perm_except(current, node) \in
10                   (in_tree(current.left, node) || in_tree(current.right, node))
11                : true
12             : false;
13    @*/
14 }
```

The code should be self-explanatory. It is just a recursive check. With "in_tree" defined, the pure function "valid" checks for the said property.

```
final class Tree {
   Node root;

   /*@
      requires [read]tree_perm_except(current, node);
      requires [read]tree_perm(node);
      public pure boolean valid(Node current, Node node) =
         current != null
            ? node != null && current != node
               ? \unfolding [read]tree_perm_except(current, node) \in
                  ((valid(current.left, node)
                     && !in_tree(current.right, node)
                        && smaller(to_bag(node), current.key))
                     || (valid(current.right, node)
                        && !in_tree(current.left, node)
                           && larger(to_bag(node), current.key)))
               : true
            : false;
   @*/
}
```

The function "getMin" returns the smallest node in a tree along with its full permission. That permission is gotten from the full tree permission, which now has a "hole" in it.

```
final class Tree {
   Node root;

   //@ given bag<int> B;
   //@ requires tree_perm(current) ** sorted(current);
   //@ requires B == to_bag(current);
   //@ ensures current == null ==> \result == null ** tree_perm(current) **
       sorted(current);
   //@ ensures current != null ==> \result != null ** tree_perm_except(current,
       \result) ** tree_perm(\result) ** valid(current, \result) **
```

```
      sorted_except(current, \result) ** sorted(\result) **
      larger(to_bag_except(current, \result), to_bag(\result)) ** \unfolding
      tree_perm(\result) \in \result.left == null;
  //@ ensures B == to_bag_except(current, \result) + to_bag(\result);
  Node getMin(Node current) {
    if (current == null) {
      return current;
    }

    //@ unfold tree_perm(current);
    if (current.left != null) {
      //@ bag<int> left_before = to_bag(current.left);
      Node result = getMin(current.left) /*@ with { B = left_before; } @*/;
      //@ bag<int> left_after = to_bag_except(current.left, result);
      //@ bag<int> result_content = to_bag(result);
      //@ assert left_before == left_after + result_content;
      //@ assert smaller(left_before, current.key);
      //@ assert subbag(left_after, left_before);
      //@ assert subbag(result_content, left_before);
      //@ assert tree_perm(current.right) ** tree_perm(result);
      //@ prove_not_in_tree(current.right, result, to_bag(current.right));
      //@ fold tree_perm_except(current, result);
      return result;
    }
    //@ fold tree_perm(current);
    //@ fold tree_perm_except(current, current);
    return current;
  }
}
```

In the beginning, we have a full permission of the tree and the tree is sorted. If the tree is null, there is no min and nothing changes. If the tree is not null, there must be a min and the permission is split into two: the permission of the result and the permission of the tree, except for the result. The sorted property is also split accordingly. The result, obviously, is a valid member of the tree. Because the result is the smallest member of the tree, it doesn't have a left branch and the tree with a hole is always equal or larger than the result. Finally, if we

combine the content of the tree with a hole and the content of the result, we should have the full original content.

The verifier needs some help to verify this function. At line 21 and 22, we have the information that "left_before == left_after + result_content" and "smaller(left_before, current.key)". It is obvious that "left_after" and "result_content" are equal or smaller than "current.key" because they are parts of "left_before" and everything in "left_before" is equal or smaller than "current.key". However, the verifier cannot derive that conclusion by itself. Only when it is told to check if the "left_after" and "result_content" are sub-bags of "left_before", it can conclude the above. The pure function "subbag" is defined as a predicate that checks whether every member of the first bag is also a member of the second bag.

```
1 final class Tree {
2    Node root;
3
4    //@ static pure boolean subbag(bag<int> b1, bag<int> b2) = (\forall int i; (i
         \memberof b1) != 0; (i \memberof b2) != 0);
5 }
```

At line 25, we have the full permission of "current.right" and "result" at the same time. That means "result" is not in "current.right" tree and hence, "tree_perm(current.right)" and "tree_perm_except(current.right, result)" are equivalent. We have to write the lemma "prove_not_in_tree" to prove these ourselves. When a statement is too complex for the verifier to prove automatically, we write functions to help proving it without changing the behavior of the main code. Those functions are called lemmas. Lemmas are the functions that are used as the stepping stones to prove more complex statements.

```
1 final class Tree {
2    Node root;
3
4    /*@
5       requires node != null ** tree_perm(current) ** ([1/2]tree_perm(node)) **
            sorted(current) ** sorted(node);
6       requires B == to_bag(current);
7       ensures node != null ** current != node ** tree_perm_except(current, node) **
            !in_tree(current, node) ** ([1/2]tree_perm(node)) ** sorted_except(current,
            node) ** sorted(node);
```

```
8       ensures B == to_bag_except(current, node);
9       public void prove_not_in_tree(Node current, Node node, bag<int> B) {
10         if (current == null) {
11            fold tree_perm_except(current, node);
12            return;
13         }
14
15         unfold tree_perm(current);
16         unfold [1/2]tree_perm(node);
17         assert current != node;
18         fold [1/2]tree_perm(node);
19         prove_not_in_tree(current.left, node, to_bag(current.left));
20         prove_not_in_tree(current.right, node, to_bag(current.right));
21         fold tree_perm_except(current, node);
22      }
23   @*/
24 }
```

The lemma requires two permissions and proves that the node is not a member of the tree and everything stays the same when we change from "tree_perm" to "tree_perm_except". The lemma is proved by unfolding the whole tree to see that the node is not equal to any node in the tree.

5.6 Delete function

The specification of the delete function is very similar to that of the insert function, but the verifying process is more complicated. Just as for the insert function, we want to ensure that we have full permission of the tree and it is "sorted" both before and after the deletion. If the delete key is found in the content of the tree, the content of the tree before the deletion is equal to the content after plus the just deleted key. Otherwise, it stays the same after the deletion.

```
1  final class Tree {
2     Node root;
3
4     //@ context Perm(root, 1) ** tree_perm(root) ** sorted(root);
5     void deleteKey(int key) {
```

```
6       root = deleteRec(root, key) /*@ with { B = to_bag(root); } @*/;
7   }
8
9   //@ given bag<int> B;
10  //@ requires tree_perm(current) ** sorted(current);
11  //@ requires B == to_bag(current);
12  //@ ensures tree_perm(\result) ** sorted(\result);
13  //@ ensures (key \memberof B) == 0 ==> B == to_bag(\result);
14  //@ ensures (key \memberof B) != 0 ==> B == to_bag(\result) + bag<int> { key };
15  Node deleteRec(Node current, int key) {
16      if (current == null) {
17          return current;
18      }
19
20      //@ unfold tree_perm(current);
21      if (key < current.key) {
22          //@ bag<int> left_before = to_bag(current.left);
23          current.left = deleteRec(current.left, key) /*@ with { B = left_before; }
                @*/;
24          //@ bag<int> left_after = to_bag(current.left);
25          //@ assert left_before == left_after || left_before == left_after + bag<int>
                { key };
26          //@ assert subbag(left_after, left_before);
27      } else if (key > current.key) {
28          //@ bag<int> right_before = to_bag(current.right);
29          current.right = deleteRec(current.right, key) /*@ with { B =
                to_bag(current.right); } @*/;
30          //@ bag<int> right_after = to_bag(current.right);
31          //@ assert right_before == right_after || right_before == right_after +
                bag<int> { key };
32          //@ assert subbag(right_after, right_before);
33      } else {
34          if (current.left == null) {
35              Node result = current.right;
36              return result;
37          }
```

```
38          if (current.right == null) {
39             Node result = current.left;
40             return result;
41          }
42
43          //@ bag<int> right_1 = to_bag(current.right);
44          Node successor = getMin(current.right) /*@ with { B = right_1; } @*/;
45          //@ unfold tree_perm(successor);
46          //@ bag<int> successor_key = bag<int> { successor.key };
47          current.key = successor.key;
48          //@ fold tree_perm(successor);
49          //@ assert tree_perm_except(current.right, successor) **
                tree_perm(successor) ** valid(current.right, successor) **
                sorted_except(current.right, successor) ** sorted(successor);
50          //@ combine(current.right, successor, to_bag_except(current.right,
                successor), to_bag(successor));
51          current.right = deleteRec(current.right, current.key) /*@ with { B =
                right_1; } @*/;
52          //@ bag<int> right_2 = to_bag(current.right);
53          //@ assert right_1 == right_2 + successor_key;
54          //@ assert subbag(right_2, right_1);
55       }
56
57       //@ fold tree_perm(current);
58       return current;
59    }
60 }
```

Like the getMin functions, we have to explicitly ask the verifier to check the sub-bag at line 25, 31 and 53. Furthermore, at line 49, we have this information: the permission of "tree with a hole", the permission of nodes in that hole, the node is a valid member of the tree, "tree with a hole" is sorted and the node is sorted. Those are the results of the "getMin" function. We need a way to combine them back to the full tree without any hole. I wrote the "combine" lemma for that purpose.

```
1 final class Tree {
```

```
2   Node root;
3
4   /*@
5      requires node != null ** tree_perm_except(current, node) ** tree_perm(node) **
           valid(current, node) ** sorted_except(current, node) ** sorted(node);
6      requires B1 == to_bag_except(current, node) ** B2 == to_bag(node);
7      ensures node != null ** tree_perm(current) ** sorted(current);
8      ensures B1 + B2 == to_bag(current);
9      public void combine(Node current, Node node, bag<int> B1, bag<int> B2) {
10        if (current == null) {
11           fold tree_perm(current);
12           return;
13        }
14
15        if (current == node) {
16           return;
17        }
18
19        unfold tree_perm_except(current, node);
20        assert (valid(current.left, node) && !in_tree(current.right, node) &&
              smaller(to_bag(node), current.key)) || (valid(current.right, node) &&
              !in_tree(current.left, node) && larger(to_bag(node), current.key));
21        if (valid(current.left, node) && !in_tree(current.right, node) &&
              smaller(to_bag(node), current.key)) {
22           combine(current.left, node, to_bag_except(current.left, node),
                 to_bag(node));
23           assert tree_perm_except(current.right, node) ** !in_tree(current.right,
                 node);
24           prove_safe_disregard(current.right, node, to_bag_except(current.right,
                 node));
25        } else {
26           combine(current.right, node, to_bag_except(current.right, node),
                 to_bag(node));
27           prove_safe_disregard(current.left, node, to_bag_except(current.left,
                 node));
28        }
```

```
29
30          fold tree_perm(current);
31       }
32    @*/
33 }
```

The lemma requires those 5 pieces of information. It proves that we could get the full tree permission back, the tree is sorted and the content of the full tree is equal to the content of "tree with a hole" plus the content of "the hole". It does that by going through all possible cases and fold and unfold or recur for the branch at each case to get the required permission. At line 25, we have the permission of a tree except for some nodes. However, we also know that those nodes are not in the tree. So, technically, we have the full permission of the tree, but the verifier could not realize that itself. Hence, we have another lemma for that. The lemma is called "prove_safe_disregard". This lemma can be reused at line 29.

```
1  final class Tree {
2     Node root;
3
4     /*@
5        requires node != null ** tree_perm_except(current, node) ** !in_tree(current,
             node) ** sorted_except(current, node);
6        requires B == to_bag_except(current, node);
7        ensures node != null ** tree_perm(current) ** sorted(current);
8        ensures B == to_bag(current);
9        public void prove_safe_disregard(Node current, Node node, bag<int> B) {
10          if (current == null) {
11             fold tree_perm(current);
12             return;
13          }
14
15          unfold tree_perm_except(current, node);
16          prove_safe_disregard(current.left, node, to_bag_except(current.left, node));
17          prove_safe_disregard(current.right, node, to_bag_except(current.right,
                node));
18          fold tree_perm(current);
19       }
```

```
20    @*/
21 }
```

The lemma requires the permission of a "tree with a hole" and a proposition saying that the hole is actually not in the tree. It proves that the tree really does not have any hole and we can have the full permission of that tree. It also says that the tree stays the same when we change from "tree_perm_except" to "tree_perm". The lemma is proved by unfolding the whole tree to see that the hole is not there.

5.7 Search function

The search function is the min function with additional steps. Instead of searching the left branch only, we now search either branch. Because they are pretty much alike, I only note the new techniques that are used here.

```
1  final class Tree {
2     Node root;
3
4     //@ given bag<int> B;
5     //@ requires tree_perm(current) ** sorted(current);
6     //@ requires B == to_bag(current);
7     //@ ensures current == null ==> \result == null;
8     //@ ensures \result != null ==> tree_perm_except(current, \result) **
          tree_perm(\result) ** valid(current, \result) ** sorted_except(current,
          \result) ** sorted(\result) ** B == to_bag_except(current, \result) +
          to_bag(\result);
9     Node search(Node current, int key) {
10       if (current == null) {
11          return current;
12       }
13
14       //@ unfold tree_perm(current);
15       if (key < current.key) {
16          //@ bag<int> left_before = to_bag(current.left);
17          Node result = search(current.left, key) /*@ with { B = left_before; } @*/;
18          /*@
```

```
19            if (result != null) {
20               bag<int> left_after = to_bag_except(current.left, result);
21               bag<int> result_content = to_bag(result);
22               assert subbag(left_after, left_before);
23               assert subbag(result_content, left_before);
24               prove_not_in_tree(current.right, result, to_bag(current.right));
25               unfold tree_perm(result);
26               assert current != result;
27               fold tree_perm(result);
28               fold tree_perm_except(current, result);
29            }
30         @*/
31         return result;
32      }
33      if (key > current.key) {
34         //@ bag<int> right_before = to_bag(current.right);
35         Node result = search(current.right, key) /*@ with { B = right_before; } @*/;
36         /*@
37            if (result != null) {
38               bag<int> right_after = to_bag_except(current.right, result);
39               bag<int> result_content = to_bag(result);
40               assert subbag(right_after, right_before);
41               assert subbag(result_content, right_before);
42               prove_not_in_tree(current.left, result, to_bag(current.left));
43               unfold tree_perm(result);
44               assert current != result;
45               fold tree_perm(result);
46               fold tree_perm_except(current, result);
47            }
48         @*/
49         return result;
50      }
51
52      //@ fold tree_perm(current);
53      //@ fold tree_perm_except(current, current);
54      return current;
```

```
55      }
56  }
```

At line 25, we are inside the "tree_perm(current)" predicate, which is unfolded at line 14. At the same time, we also have the "tree_perm(result)" predicate. So, "current" and "result" must be different. The verifier does not know this naturally, but if we unfold the "tree_perm(result)" and assert the difference there, it would remember. The same thing happens again at line 43.

Chapter 6

Red-black tree specification

6.1 Overview

In this chapter, we talk about the verification process of a red-black tree. After successfully verifying the binary search tree, the red-black tree is implemented on top of it. To implement a red-black tree from binary search tree, we need to add to each node the information about color and parent node. Another implementation method use a double black flag instead of a parent node. In this thesis, I used double black flag because handling two-way references could be very hard in a concurrent program. Because they are new information and red-black tree does not change the existing properties of binary search tree, existing specifications should verify.

Another difference from the binary search tree is the rotations. A red-black tree utilizes rotation methods for its algorithm. There are two rotations in total: left and right. They are illustrated by the Figure 6-1.

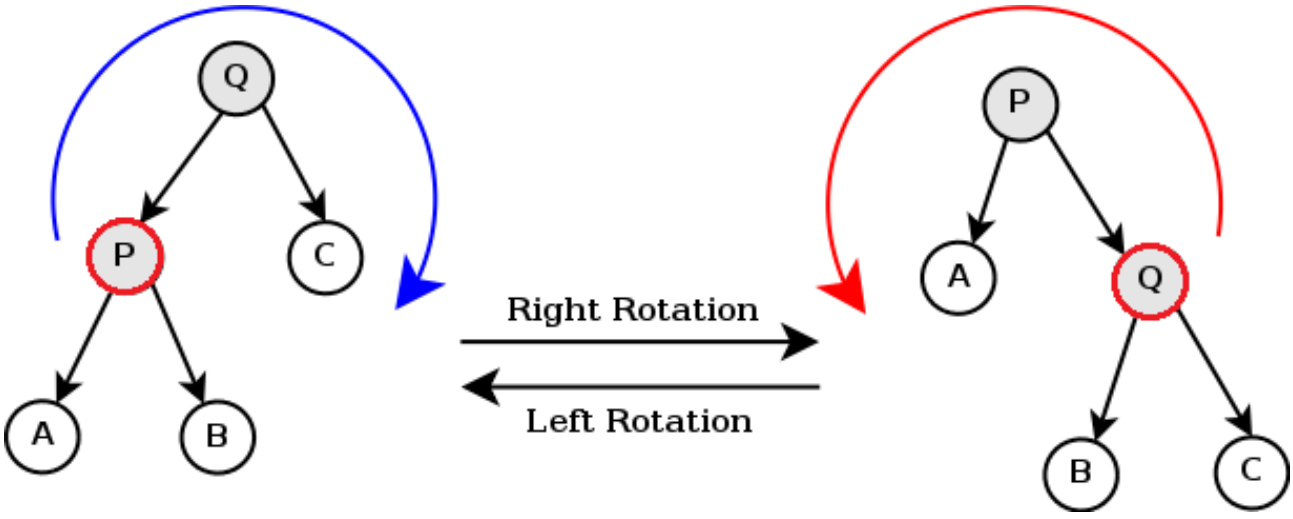

Figure 6-1: Tree rotation.

The implementation of right rotation is simple. We move the left node to the top, the top node to the right, and the right child of the left node is turned to the left child of the right node. Then, after moving the nodes, we swap the color of the top node and the right node. The left rotation is a mirrored of right rotation.

```
final class Tree {
   Node root;

   Node rotateRight(Node node) {
      Node tempNode = node.left;
      boolean tempColor = node.color;

      node.left = tempNode.right;
      tempNode.right = node;

      node.color = tempNode.color;
      tempNode.color = tempColor;
      return tempNode;
   }
}
```

A red-black tree is a binary search tree with more restrictions. To maintain those restrictions, we use rotation and color swap. The red-black tree source code should stay almost the same as the binary search tree, with some rotations and color swaps occasionally added to maintain the restrictions. The color swaps should not affect the binary search tree properties at all because we do not have colors in binary search tree. So, to prove that the binary search tree properties persist in red-black tree, we only have to prove that the rotations do not change the existing "sorted" property.

A red-black tree enforces two additional properties, which are "no double red" and "no double black". The first property says that, "there cannot be two consecutive nodes with the same red color". The violation of this property only occurs during the insertion process. The newly inserted node is marked as red. If the new node is inserted after a red node, we have a "double red" problem. This problem is solved by rotating the tree and swapping the color. The second property says that, "the number of black nodes from the top of the tree to any leaf node, called black height, is equal". The violation of this property only occurs during the deletion process. In this process, we only delete nodes with at most one child and replace the deleted

node with its only child or null. If we delete a black node and the node replacing it is also black or null, the count at that branch is decreased without an immediate way to increase it. So, we count that node as double black to temporarily prevent the imbalance. Then, we proceed to remove the "double black" in that node without causing the imbalance. This utilizes color swap and rotation as with the "double red" problem, but with different patterns. Because the rotations to fix these problems have very specific patterns, if we specify those patterns as the pre-conditions of the rotations, we can easily prove that those rotations really fix the problems by specifying the post-conditions. However, we also want to ensure that none of the rotation causes more problems as they fix. So, we have to specify that the rotations that fix the "double red" problem do not cause the "double black" problem and vice versa.

The binary search tree properties in red-black tree and the "no double red" property are specified and successfully verified. However, the "no double black" property is not specified. There is not much difference with the "no double red" property in term of implementation. In this thesis, I only give the base ideas and the speculations of "no double black" property. The concrete specification is a future work. Furthermore, because of the extensiveness of the specification and most of the techniques have been explained in the binary search tree verification, only excerpts of the code will be presented. The complete source code can be downloaded at `https://fmt.ewi.utwente.nl/media/rbt.zip`.

6.2 Node specification

Specifying the Node class is straightforward and self-explanatory. It is the same as the binary search tree node with added color and double black flag.

```
1 final class Node {
2   int key;
3   Node left;
4   Node right;
5   boolean color;
6   boolean dblack;
7
8   //@ ensures Perm(key, 1) ** Perm(left, 1) ** Perm(right, 1) ** Perm(color, 1) **
        Perm(dblack, 1);
9   //@ ensures key == item ** left == null ** right == null ** color == true **
        dblack == false;
```

```
10      Node(int item) {
11          key = item;
12          left = null;
13          right = null;
14          color = true;
15          dblack = false;
16      }
17  }
```

From here, we have to deal with the color of the nodes. In our view, null is considered black. Constantly checking if the node is null can be quite bothersome. So, I wrote a function "get_color" for that. It checks whether the node is null. If it is, the color is considered black. If it is not, the color of that node is returned. To access the color value, we need to have the permission. Read permission of a single node would be enough. However, our permissions are packaged into a predicate that contains the permissions of a node and all of its children nodes. Furthermore, those permissions can have a hole in it. The hole does not affect our result, except for when the hole is as big as the tree itself. Then, we have no permission.

```
1   final class Tree {
2       Node root;
3
4       /*@
5           requires [read]tree_perm_except(node, exclusion);
6           public pure boolean get_color(Node node, Node exclusion) =
7               node != null && node != exclusion
8                   ? \unfolding [read]tree_perm_except(node, exclusion) \in (node.color)
9                   : false;
10      @*/
11
12      //@ requires [read]tree_perm(node);
13      //@ public pure inline boolean get_color(Node node) = get_color(node, null);
14  }
```

In this implementation, if the hole is equal to the tree, we considered it a black node. It represents a clean cut of a tree when we split the permission.

6.3 Binary search tree properties in red-black tree

The objective of this part is to transform the existing binary search tree specification to work with a red-black tree implementation. Because of the two newly added elements in Node, the permission predicate has to include them as well.

```
1  final class Tree {
2     Node root;
3
4     /*@
5        public resource tree_perm_except(Node current, Node node) =
6           current != null ==>
7              (current != node ==>
8                 Perm(current.key, 1) ** Perm(current.color, 1) ** Perm(current.dblack,
                      1)
9                    ** Perm(current.left, 1) ** Perm(current.right, 1)
10                     ** tree_perm_except(current.left, node)
11                        ** tree_perm_except(current.right, node));
12    @*/
13
14 }
```

As stated earlier, we have to prove that the rotations do not affect the "sorted" property. In other words, the tree should remain "sorted" before and after the the rotation. Because "rotateLeft" and "rotateRight" are mirrored functions, only one function is presented here. The rotation of a tree is illustrated by the Figure 6-1.

```
1  final class Tree {
2     Node root;
3
4     //@ given bag<int> B;
5     //@ requires node != null ** tree_perm(node) ** sorted(node);
6     //@ requires \unfolding tree_perm(node) \in (node.right != null);
7     //@ requires B == to_bag(node);
8     //@ ensures \result != null ** tree_perm(\result) ** sorted(\result);
9     //@ ensures \unfolding tree_perm(\result) \in (\result.left != null);
10    //@ ensures B == to_bag(\result);
```

```
11    Node rotateLeft(Node node) {
12       //@ unfold tree_perm(node);
13       Node tempNode = node.right;
14       boolean tempColor = node.color;
15
16       //@ bag<int> temp_content = to_bag(tempNode);
17       //@ unfold tree_perm(tempNode);
18       //@ bag<int> temp_left_content = to_bag(tempNode.left);
19       //@ assert temp_content == temp_left_content + bag<int> { tempNode.key } +
             to_bag(tempNode.right);
20       //@ assert subbag(temp_left_content, temp_content);
21       //@ assert (tempNode.key \memberof temp_content) != 0;
22       node.right = tempNode.left;
23       tempNode.left = node;
24
25       node.color = tempNode.color;
26       tempNode.color = tempColor;
27       //@ fold tree_perm(node);
28       //@ fold tree_perm(tempNode);
29       return tempNode;
30    }
31 }
```

The ghost bag B is explained in the binary search tree chapter. It is there to control the content changes. We would expect it in all of our functions, even when we do not expect any content of the tree to be changed. Without it, we cannot assert if the content has any change or not, and what the changes is. The permission and "sorted" property of the tree stay the same before and after the rotation. To be able to left rotate, there must be something on the right to begin with. Hence, the right node is not null. After the rotation, the left node must not be null because the top node has moved there. It is worth to notice that the left node after the rotation has the color of the right node before the rotation. Although we do not need it to prove the binary search tree properties, it is needed when we have to verify the properties that are affected by color. Everything is mirrored for the right rotation. As we can see at line 19, 20 and 21, VerCors still struggles with "+" operator among bags and needs help showing the relations. This could be a noteworthy improvement of future versions of VerCors. Apart from

that, it handles the specification nicely, and is able to prove the "sorted" property without any further help.

After specifying two rotation function, all of the properties of binary search tree can be verified in the red-black tree context.

6.4 No double red property

The "no_db_red" property can be specified as a recursive function. For every node in the tree, if it is red, its children must be black. Specifying this property is not hard. However, it is not always true. In the process of the insertion, we violate this property by adding a red node to an existing red node, creating a double red. We fix this when we return from the recursion to the parent node. If it is the root node, it does not have a higher node to return to. In that case, we can just set it to black, eliminating the double red. If we look at the cases in Chapter 3.3.2, we can see that the double red problem is always fixed or move up at the parent. That means during the insertion, double red can only occurred at the current node. Because the double red would definitely be resolved at one level higher, we need the property "db_red_at_top_except" saying that the top node has one and only one double red, and it is the only occurrence.

```
1  final class Tree {
2     Node root;
3
4     /*@
5        requires [read]tree_perm_except(current, node);
6        public pure boolean no_db_red_except(Node current, Node node) =
7           current != null && current != node
8              ? \unfolding [read]tree_perm_except(current, node) \in
9                 ((!get_color(current, node)
10                   || (!get_color(current.left, node)
11                      && !get_color(current.right, node)))
12                && no_db_red_except(current.left, node)
13                   && no_db_red_except(current.right, node))
14             : true;
15    @*/
16
```

```
17    /*@
18       requires [read]tree_perm_except(current, node);
19       public pure boolean db_red_at_top_except(Node current, Node node) =
20          current != null && current != node
21             ? \unfolding [read]tree_perm_except(current, node) \in
22                (get_color(current, node)
23                   && ((get_color(current.left, node)
24                      && !get_color(current.right, node))
25                   || (!get_color(current.left, node)
26                      && get_color(current.right, node)))
27                   && no_db_red_except(current.left, node)
28                      && no_db_red_except(current.right, node))
29             : true;
30    @*/
31
32    //@ requires [read]tree_perm(current);
33    //@ public pure inline boolean no_db_red(Node current) =
        no_db_red_except(current, null);
34
35    //@ requires [read]tree_perm(current);
36    //@ public pure inline boolean db_red_at_top(Node current) =
        db_red_at_top_except(current, null);
37 }
```

The "no_db_red" property is preserved across the whole tree like the "sorted" property. So, at every place that we have the "sorted" property, we also add the "no_db_red" property. The only exception is the rotation functions. While rotating does not change the sorted property, it changes the color related property.

Because the rotation change the color pattern, before specifying the insert and delete functions, the rotation functions need to specify the color rotations. The rotation are meant to fix the double red and double black problem. So, their input are very specific. We can use that to our advantage. There are eight color patterns in total. However, the left rotation and right rotation are mirrored. That left us with four unique patterns. The first two are used by the insert function, and the remaining two are used by the delete function.

1. The first case is used when we want to move the double red into position. If we have

double red at top, we can rotate the double red to another branch.

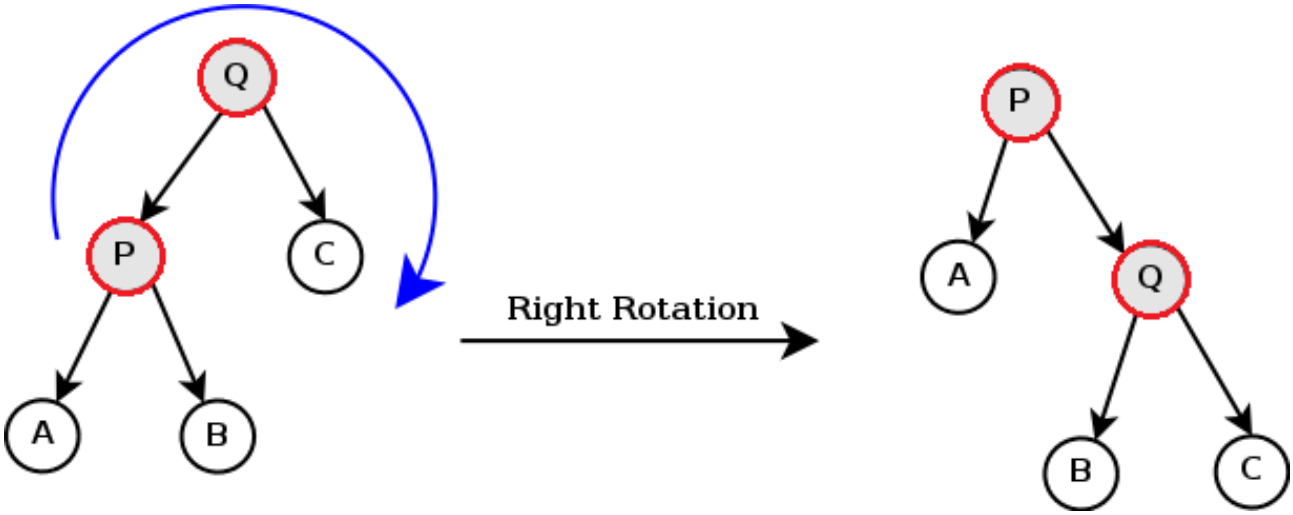

Figure 6-2: Rotation case 1.

2. The second case is used when fixing the double red at parent level. The input is the exact color pattern and the output is a tree without double red.

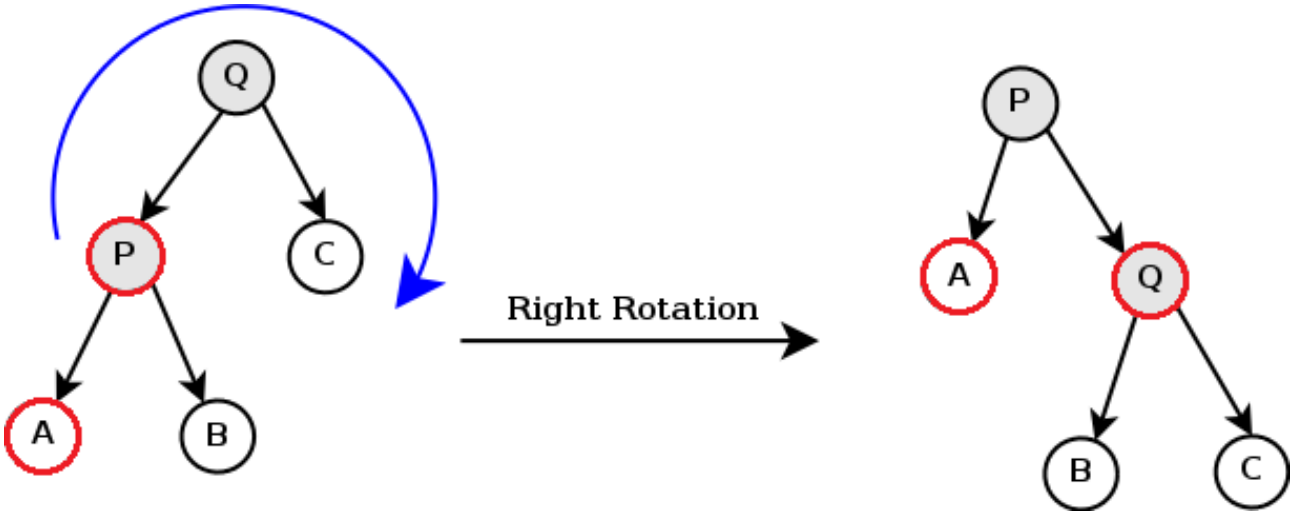

Figure 6-3: Rotation case 2.

3. The third case is used by the delete function. So, it must have no double red in the beginning. If the top node and the node to the right are black, then the right rotation will not create double red. It is mirrored for left rotation.

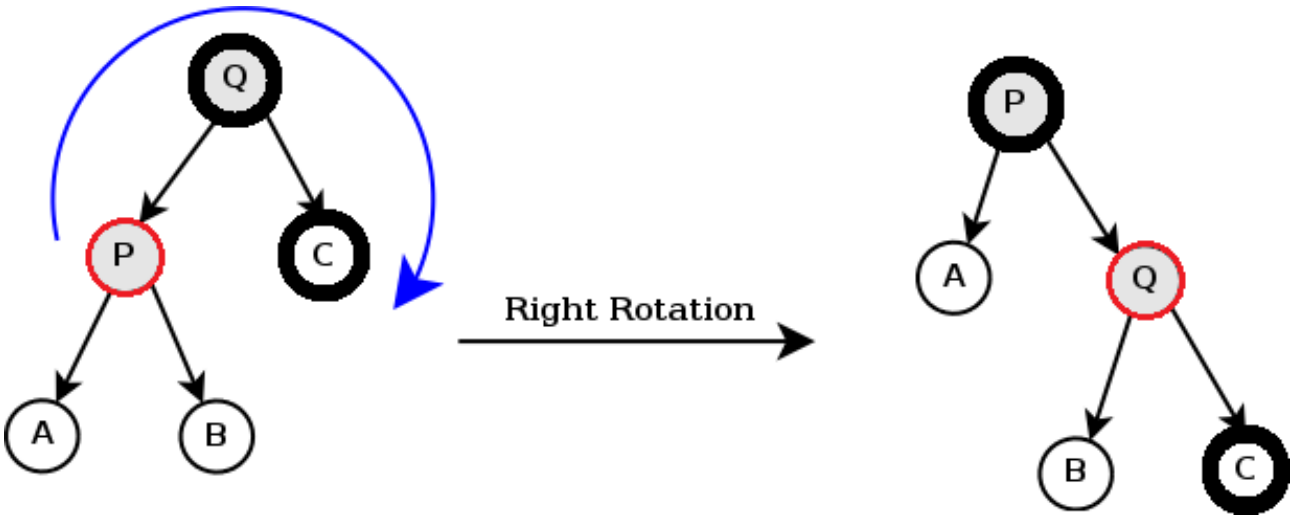

Figure 6-4: Rotation case 3.

4. Like the third case, there is no double red in the beginning of the fourth case. If both the node to the left and its left node are black, right rotation will not create double red. It is mirrored for left rotation.

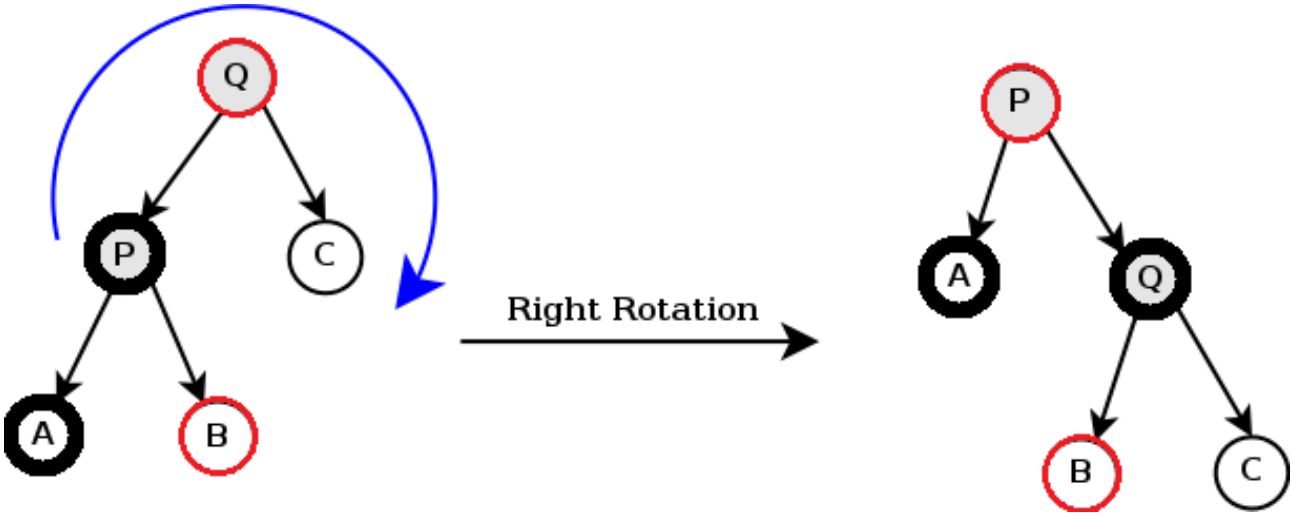

Figure 6-5: Rotation case 4.

Beside those cases, we also have to specify the change of color and link in the rotation itself. More specifically, the color of the top node stay the same, the color of the right node move to the left node after the left rotation and the color of the left node move to the right node after the right rotation. The right left node become the left right node after the left rotation and vice versa.

With those in mind, the right rotation is specified as following:

```
1 final class Tree {
2    Node root;
3
```

```
4    //@ given bag<int> B;
5    //@ given boolean node_color;
6    //@ given boolean left_node_color;
7    //@ given Node left_right_node;
8    //@ given int rcase;
9    //@ requires node != null ** tree_perm(node) ** sorted(node);
10   //@ requires \unfolding tree_perm(node) \in (node.left != null);
11   //@ requires B == to_bag(node);
12   //@ requires node_color == get_color(node);
13   //@ requires left_node_color == \unfolding tree_perm(node) \in
         (get_color(node.left));
14   //@ requires left_right_node == \unfolding tree_perm(node) \in (\unfolding
         tree_perm(node.left) \in (node.left.right));
15   //@ requires rcase == 1 ==> node_color && left_node_color && db_red_at_top(node);
16   //@ requires rcase == 2 ==> \unfolding tree_perm(node) \in (!node.color &&
         get_color(node.left) && !get_color(node.right) && db_red_at_top(node.left) &&
         no_db_red(node.right) && \unfolding tree_perm(node.left) \in
         (get_color(node.left.left)));
17   //@ requires rcase == 3 ==> no_db_red(node) && \unfolding tree_perm(node) \in
         (!node.color && !get_color(node.right));
18   //@ requires rcase == 4 ==> no_db_red(node) && \unfolding tree_perm(node) \in
         (!get_color(node.left) && \unfolding tree_perm(node.left) \in
         (!get_color(node.left.left)));
19   //@ ensures \result != null ** tree_perm(\result) ** sorted(\result);
20   //@ ensures \unfolding tree_perm(\result) \in (\result.right != null);
21   //@ ensures B == to_bag(\result);
22   //@ ensures node_color == get_color(\result);
23   //@ ensures left_node_color == \unfolding tree_perm(\result) \in
         (get_color(\result.right));
24   //@ ensures left_right_node == \unfolding tree_perm(\result) \in (\unfolding
         tree_perm(\result.right) \in (\result.right.left));
25   //@ ensures rcase == 1 ==> db_red_at_top(\result);
26   //@ ensures rcase == 2 ==> no_db_red(\result);
27   //@ ensures rcase == 3 ==> no_db_red(\result);
28   //@ ensures rcase == 4 ==> no_db_red(\result);
29   Node rotateRight(Node node) {
```

```
30    }
31 }
```

The left rotation is a mirrored of right rotation. With the "no double red" property defined and both rotations specified, the insert function can be specified as following:

```
1  final class Tree {
2     Node root;
3
4     //@ given bag<int> B;
5     //@ given boolean old_color;
6     //@ requires tree_perm(current) ** sorted(current) ** no_db_red(current);
7     //@ requires B == to_bag(current);
8     //@ requires old_color == get_color(current);
9     //@ ensures \result != null ** tree_perm(\result) ** sorted(\result);
10    //@ ensures no_db_red(\result) || db_red_at_top(\result);
11    //@ ensures B + bag<int> { key } == to_bag(\result);
12    //@ ensures old_color == get_color(\result) || \unfolding tree_perm(\result) \in
          (\result.color && !get_color(\result.left) && !get_color(\result.right));
13    Node insertRec(Node current, int key) {
14    }
15 }
```

Because the double red problem is always fixed or move up at the parent, at the end of each recursion, it has either no double red or double red at the top node only. This property is recorded at line 10. We use "old_color" parameter to control the change of color. The "old_color" parameter is ghost parameter like the bag B. If we remember, in the insertion process, the color of the top node will not change, except for the one case that is shown in Figure 3-1. Line 12 specifies this property using "old_color" parameter.

Up until now in this section, we have only considered a full tree. To verify the delete function, we must have the ability to split and combine a tree permission without losing any information. In the binary search tree, we used "valid" property for this. In the red-black tree, we only have to evolve this property to include the new properties. When we split a tree, both new trees retain the "no_db_red" property. The only unknown factor is at the joint. For the whole tree to have "no_db_red" property, either the top node of the hole or the hole's parent must be black. The "valid" function can be implemented to include this property.

```
final class Tree {
   Node root;

   /*@
      requires [read]tree_perm_except(current, node);
      requires [read]tree_perm(node);
      public pure boolean valid(Node current, Node node) =
         current != null
            ? node != null && current != node
               ? \unfolding [read]tree_perm_except(current, node) \in
                  (((current.left == node ==> !get_color(current, node) ||
                     !get_color(node))
                    && valid(current.left, node) && !in_tree(current.right, node)
                       && smaller(to_bag(node), current.key))
                  || ((current.right == node ==> !get_color(current, node) ||
                     !get_color(node))
                    && valid(current.right, node) && !in_tree(current.left, node)
                       && larger(to_bag(node), current.key)))
               : true
            : false;
   @*/
}
```

After having the valid function adjusted, the specification of the delete function is not so different from the insert function.

```
final class Tree {
   Node root;

   //@ given bag<int> B;
   //@ given boolean old_color;
   //@ requires current != null ** tree_perm(current);
   //@ requires sorted(current) ** no_db_red(current);
   //@ requires B == to_bag(current);
   //@ requires old_color == get_color(current);
   //@ ensures tree_perm(\result);
```

```
11    //@ ensures sorted(\result) ** no_db_red(\result);
12    //@ ensures (key \memberof B) == 0 ==> B == to_bag(\result);
13    //@ ensures (key \memberof B) != 0 ==> B == to_bag(\result) + bag<int> { key };
14    //@ ensures !old_color ==> !get_color(\result);
15    Node deleteRec(Node current, int key) {
16    }
17 }
```

We know that the delete function will not create double red problem under any circumstances. Turning a node from black to red without knowing its parent could potentially create double red problem. So, we would expect that at all recursion cycles, if the top node is black, it would remain black. This is recorded at line 14 by using the "old_color" ghost parameter.

The "no double red" property has been successfully verified across the whole red-black tree. That leaves "no double black" the only unverified property of the red-black tree.

6.5 No double black property

As mentioned in Chapter 6.1, the "no double black" property is not verified. This section is used to give the verification idea of this property. To verify it, we can use the same work-flow and thought process as the previous property.

Although we call it "no double black", this property work with the black height of the tree. The "get_color" function would become recursive "get_height" function. The black height is equal to the black height of either children, then it is increased by 1 if the current node is black.

The "no_db_red" property would become "equal_height" property. It says that, for all nodes in the tree, the black height of the left branch is equal to the black height of the right branch.

Rotations would change the black height in a specific way. However, the rotation patterns that are used by the insert function does not change the black height. This can be confirmed by inspecting the Figure 6-2 and Figure 6-3.

When splitting the permission, the hole gets the black height of its sibling. Hence, the "equal_height" property of the tree with a hole is retained. Another property would be implemented in the "valid" function to support the combination. It says that the black height of the tree in the hole must be equal to the black height of the hole sibling.

These are just rough speculations. Issues are bound to appear at the implementation.

However, with everything which has been discussed in this thesis, I strongly believe that those issues are manageable.

Chapter 7

Related work

It is well-known that the deletion of a red-black tree is much more difficult to implement and verify than the insertion. Although red-black tree has seen several attempt at verifying in the past, most authors leave the deletion out of their papers. For example, the paper Verifying Red-Black Trees [7] proposes verification techniques based on graph rewriting to verify the correctness of the insertion. The deletion is not mentioned anywhere in the paper. The paper Auto-Active Proof of Red-Black Trees in SPARK [11] takes another approach. The authors rely on auto-active verification in SPARK. However, it is not a complete verification of red-black tree. The deletion algorithm is not implemented, and they did not verify that every branch in a red-black tree contains the same number of black nodes. Although this thesis did not fully verify the number of black nodes either, the tactic and guideline is provided.

Constantin Enea, Mihaela Sighireanu and Zhilin Wu have the closest result to our research [12]. By also using separation logic, they were able to verify a C implementation of red-black tree. However, their research is confined in sequential programs. Although our red-black tree is not implemented in a concurrent manner, our verification supports concurrency thanks to the permissions. Having to deal with the permission hugely increase the complexity of the verification, but it also increase the flexibility of it. A concurrent program can be used in a sequential setting, but the opposite is not possible.

Chapter 8

Conclusion and future work

In this thesis, we have shown how VerCors can be used to verify a standard implementation of red-black tree. Although the implementation is far from the one used in the industry, this is a rather successful demonstration of the capability of VerCors.

The techniques presented here can be a significant benefit for verifying a different implementation of the red-black tree. The multi-layer approach, in which the red-black tree is reduced into multiple simpler data structure, can be a good approach for other attempts. The verification also reveals many improvement point that VerCors developer might take interested in, like the ability to automatically identify sub-bags in the "+" operator.

The binary search tree and red-black tree implementation presented in this thesis have been specified. So, it is possible to use them in a library for data structure.

Future work on this topic could include the complete specification of the red-black tree. With the complete specification available, the next step is to be able to verify other implementations of the red-black tree. Ultimately, we want to create a model of the transformation from one implementation to another and apply it to the specification to be able to verify any variations of the red-black tree.

Bibliography

[1] Scala 2.8 collections API – red-black trees. https://www.scala-lang.org/docu/files/collections-api/collections_20.html.

[2] std::map - cppreference.com. https://en.cppreference.com/w/cpp/container/map.

[3] std::set - cppreference.com. https://en.cppreference.com/w/cpp/container/set.

[4] Features | OpenJML, 2015. http://www.openjml.org/documentation/features.shtml.

[5] Preliminary report highway: HWY18MH010, 2018. https://www.ntsb.gov/investigations/AccidentReports/Pages/HWY18MH010-prelim.aspx.

[6] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In *PLPV '12: Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, pages 71–82. ACM, 2012.

[7] Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel, Barbara König, and Vitali Kozioura. Verifying red-black trees. 2012.

[8] Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. Dynamic logic for Java. In *Deductive Software Verification - The KeY Book*, pages 49–106. Springer International Publishing, 2016.

[9] Jan Boerman, Marieke Huisman, and Sebastiaan Joosten. Reasoning about JML: Differences between KeY and OpenJML. In *Integrated Formal Methods*, pages 30–46. Springer, 2018.

[10] Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level verification of algorithms with KeY. *International Journal on Software Tools for Technology Transfer*, 17(6):729–744, 2015.

[11] Claire Dross and Yannick Moy. Auto-active proof of red-black trees in SPARK. In *NASA Formal Methods*, pages 68–83. Springer International Publishing, 2017.

[12] Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. In *Automated Technology for Verification and Analysis*, pages 80–96. Springer International Publishing, 2015.

[13] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[14] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. Formal specification with the Java Modeling Language. In *Deductive Software Verification - The KeY Book*, pages 193–241. Springer International Publishing, 2016.

[15] Bart Jacobs, Jan Smans, Pieter Philippaerts, and Frank Piessens. The VeriFast program verifier: A tutorial for Java Card developers, 2011.

[16] Bart Jacobs, Jan Smans, and Frank Piessens. VeriFast: Imperative programs as proofs. In *VSTTE Workshop on Tools & Experiments*. ACM, 2010.

[17] Sebastiaan Joosten, Wytse Oortwijn, Mohsen Safari, and Marieke Huisman. An exercise in verifying sequential programs with VerCors. In *20th Workshop on Formal Techniques for Java-like Programs Formal techniques*, 2018.

[18] John Reynolds. Separation logic: a logic for shared mutable data structures. *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 17:55–74, 2002.

[19] Jan Smans, Bart Jacobs, and Frank Piessens. Verifast for Java: A tutorial. In *Aliasing in Object-Oriented Programming*, pages 407–442. Springer, 2013.

[20] Jan Smans, Bart Jacobs, Frank Piessens, Willem Penninckx, Frédéric Vogels, and Pieter Philippaerts. Verifying Java programs with VeriFast. In *Aliasing in Object-Oriented Programming*. Springer, 2013.