Technical Report, number 04/18
Departamento de Sistemas Informáticos y Computación

An Assertional Proof of Red-Black Trees using Dafny

Ricardo Peña
Universidad Complutense de Madrid

November 20th, 2018

Abstract Red-black trees are convenient data structures for inserting, searching, and deleting keys with logarithmic costs. However, keeping them balanced requires careful programming, and sometimes to deal with a high number of cases. In this paper, we present a functional version of a red-black tree variant called *Left-Leaning*, due to R. Sedgewick, which reduces the number of cases to be dealt with to a few ones. The code is rather concise, but reasoning about its correctness requires a rather large effort.

We provide formal preconditions and postconditions for all the functions, prove their termination, and that the code satisfies its specifications. The proof is assertional, and consists of interspersing enough assertions among the code in order to help the verification tool to discharge the proof obligations. We have used the Dafny verification platform, which provides the programming language, the assertion language, and the verifier.

To our knowledge, this is the first assertional proof of this data structure, and also one of the few ones including deletion.

Keywords Data Structures · Balanced Trees · Verification Platforms

CR Subject Classification Software and its engineering-Software verification · Software and its engineering-Specification languages

1 Introduction

Red-Black trees were invented by Rudolf Bayer in 1972 [4] under the name *Symmetric Binary B-trees*, and were presented as a specialisation of B-trees, invented in turn by himself and Edward McCreight that same year [5]. Its current name, *Red-Black Trees* (in what follows, RB), is due to Leonidas Guibas and Robert Sedgewick [11], who studied in depth their properties. After that, many other variants have been proposed.

The main idea is to have binary search trees with a balance property weaker than that of AVL trees [1], which historically was the first balanced search tree proposal. While AVLs require the difference in height between two sibling trees to be at most one unit, in RB this difference can grow up to one height doubling the other. This is because a distinction is made between red nodes and black nodes, and the invariant requires the number of black nodes to be the same in every path from the root to the leaves, and no two consecutive red nodes are allowed in any path. In spite of that, the costs of searching, inserting, or deleting a key are still logarithmic in the tree size.

The work is supported by the Spanish Ministry of Economy, Industry, and Competitiveness under Grant TIC2017-8621-R, CAVI-ART-2, *Computer Assisted Validation by Analysis, Transformation and Proofs*

Ricardo Peña
Computer Science School, Universidad Complutense de Madrid
C/ Jose Garcia Santesmases 9, 28040 Madrid, Spain
Tel.: +34-91-3947627
Fax: +34-91-3947529
E-mail: ricardo@sip.ucm.es

There is a correspondence between some RB variants and other balanced trees such as 2-3 trees and 2-3-4 trees, which are also specializations of B-trees. One problem of these balanced trees is that they need to deal with a high number of different cases during insertion and deletion [21]. This complexity seems to have been inherited by RBs, which also require careful programming not to forget any of the possible cases.

For instance, the implementation presented by Cormen *et al* [9] requires a pointer to the parent in each node, and 6 cases for repairing the invariant during the insertion. Deletion needs its own re-balancing, and it introduces 8 cases more. The code only for deletion amounts to 80 lines.

Anderson proposed a simpler version of RB [2], known as AA-trees, by forbidding that a node may have a red left child. This makes AA-trees isomorphic to 2-3 trees. Only two basic transformations, respectively called *skew* and *split*, are needed for repairing the invariant during insertion or deletion. However, up to five of them may be used in a row. Insertion, deletion and balancing can be programmed in about 100 lines of Java code, as it is presented by Weiss [24].

Chris Okasaki proposed a functional version [19] and reduced re-balancing to four simple cases. His RBs are the less restricted variant in which any child of a node may be red or black. Nevertheless, he declines to present a delete algorithm and admits that it would have a higher number of cases to deal with.

Regarding formal verification, Jean-Christophe Filliâtre and Pierre Letouzey built [10] a proof of correctness of RBs with the proof assistant Coq [6]. The Coq scripts[1] amount to 2150 lines. They include deletion, and the proof script of this function extends along 240 lines, which gives an idea of the task complexity. Later, in 2011, Andrew Appel and Pierre Letouzey built another proof in Coq[2] much along the same lines as that of 2004. In the Why3 [8, 7] repository, it appears a functional version of RBs with a high number of auxiliary lemmas proving its correctness. Its author is also Filliâtre, and it does not include a delete method.

Here, we present a functional version of the so called *Left-Leaning Red-Black Trees* (in what follows, LLRB), introduced by Robert Sedgewick in 2008 [3][3], and having a more restricted version in 2011 [22]. In the 2008 version, only nodes with a red right child and a black left one are forbidden, the other combinations being legal. This makes LLRB isomorphic to 2-3-4 trees. In the 2011 one, they introduce the additional restriction of forbidding two red children. These latter trees, as AA-trees above, are isomorphic to 2-3 trees.

The reason for choosing the left-leaning version of red-black trees is that the number of cases to be dealt with seems to be minimal with respect to the other versions. In LLRBs, there are three basic transformations, which will be explained in detail in Sec. 2, and the number of cases to be considered —three in insertion and five more in deletion— seem to be low enough to attempt a direct assertional proof of the code. This kind of proofs [17] are done by providing a formal precondition and postcondition to each method, and by introducing intermediate assertions in the code in order to help the verification tool to complete the proof. Assertional proofs are better suited for programmers because they force them to document the code with the properties expected at critical text locations. This way of thinking is closer to the way programmers use to reason about programs.

As a consequence of this simplicity, the code is rather concise. Our functional code has 113 non-empty lines of code, and it includes insertion, deletion, membership, and minimum. The deletion code is 26 lines long, and by including the intermediate assertions of the proof, it extends to about 80 lines.

We have used the Dafny platform [15, 13, 16] for proving LLRBs correct. This system provides a programming language with imperative constructions such as destructive assignments and **while** loops, but a subset of it is purely functional. It also allows algebraic datatypes and pattern matching on values of these types. There are specific syntactic constructs for specifying preconditions, postconditions, invariants and intermediate assertions. The distinctive feature of Dafny is that a verification condition generator and a logical verifier are running in the background while the programmer is introducing code, so providing an immediate feedback on which parts of the text are incorrect, or cannot be proved, with respect to the specification. The verifier is Z3 [18], an up-to-date SMT (*Satisfiability Modulo Theories*) solver which can directly deal with linear arithmetic formulas, arrays, algebraic datatypes and uninterpreted functions. It can also decide the validity of first-order formulas, provided the appropriate hints are given to the solver.

Our contribution here is the proof of correctness of a functional implementation of LLRB with insertion and deletion. To our knowledge, this is the first assertional proof of this data structure, and one of the few proofs

[1] Available at `https://www.lri.fr/~filliatr/fsets/`.

[2] Available at `https://coq.inria.fr/library/Coq.MSets.MSetRBT.html`, with an accompanying technical report at `http://www.cs.princeton.edu/~appel/papers/redblack.pdf`.

[3] Java code available in the slides at `https://www.cs.princeton.edu/~rs/talks/LLRB/08Dagstuhl/RedBlack.pdf`.

including deletion. In addition to the proof itself, we believe that the preconditions and postconditions proposed in the text clarify the exact assumptions each function does about its environment, and which properties it guarantees. This much contributes to understand why the code works and why it is correct. The explanations given in the 2008 paper [3] are informal and leave the reader with the unsatisfactory feeling that something is missing, and that perhaps not all possibilities have been covered by the code. In fact, we believe to have found a bug in that code, if it were intended for the less restricted version presented in 2008, in which a black node may have two red children. That code only works for the more restricted version of 2011 [22]. Our code is correct for the less restricted LLRBs.

Having a verified version of RB trees is a valuable knowledge by itself, since these trees constitute the basis for implementing sets, multisets and maps in the public libraries of some popular programming languages. With this code and proof we also expect to contribute to spreading the use of tools like Dafny in teaching program design and verification. This example may be useful for introducing complex data structures to students, with both rigor and clarity. The code snippets presented in the paper reproduce the fundamentals of the proof. A complete file with all the code and assertions accompanies the paper[4] and can be run in any Dafny system.

Paper outline: In Sec. 2 we informally explain the LLRB invariant and the three basic transformations. In Sec. 3 we introduce the Dafny features that will be used in the rest of the paper. Sec. 4 presents the Dafny definitions of the predicates and auxiliary functions that will occur in the assertions. Sec. 5 explains in detail the *insert* function with all its invariants and transformations, while Sec. 6 does the same with the *delete* function. In Sec. 7 we report on our experience using Dafny to verify this data structure. Finally, we draw some conclusions.

2 Left-Leaning Red-Black Trees

The variant of red-black trees proposed in [3] has the following formal definition:

Definition 1 An LLRB is a binary tree with unique keys in the internal nodes, and with the edges joining two nodes having either a black or a red color. An LLRB satisfies the following invariant:

1. It is a Binary Search Tree, i.e. its inorder traversal yields a sorted list (we will refer to this property as BST).
2. All the paths from the root to the leaves have the same number of black edges (we will refer to this property as the tree being *balanced*).
3. In any of these paths there cannot be two consecutive red edges.
4. There cannot be a node with a red edge to its right child and a black edge to its left one (we will refer to this property as the tree being *left-leaning*).

This definition presents two novelties with respect to the other RBs we have surveyed in the Introduction. First, what are red or black are the edges, not the nodes. This allows us to present the transformations taking place in the tree during insertion and deletion in a more intuitive way: the two basic rotations are applied to the edges, rather than to the whole trees. Second, an isomorphism can be established between LLRB trees and 2-3-4 trees:

- A node with two black edges to its children corresponds to a 2-node in a 2-3-4 tree, i.e. to a node having a key and two children.
- A node joined to its left child with a red link and to its right one with a black link, corresponds to a 3-node in a 2-3-4 tree, i.e. to a node having two keys and three children.
- A node joined to its children with two red links corresponds to a 4-node in a 2-3-4 tree, i.e. to a node having three keys and four children.

Given a 2-3-4 tree, there is a unique LLRB representing it, and the other way around. In Fig. 1 we show the correspondence between nodes in an LLRB and nodes in a 2-3-4 tree, and an example of two isomorphic trees. The authors take advantage of this isomorphism by applying to LLRBs the same strategies used in 2-3-4 trees for insertion and deletion.

[4] Available at `https//:dalila.sip.ucm.es/~ricardo/LLRB.dfy`.

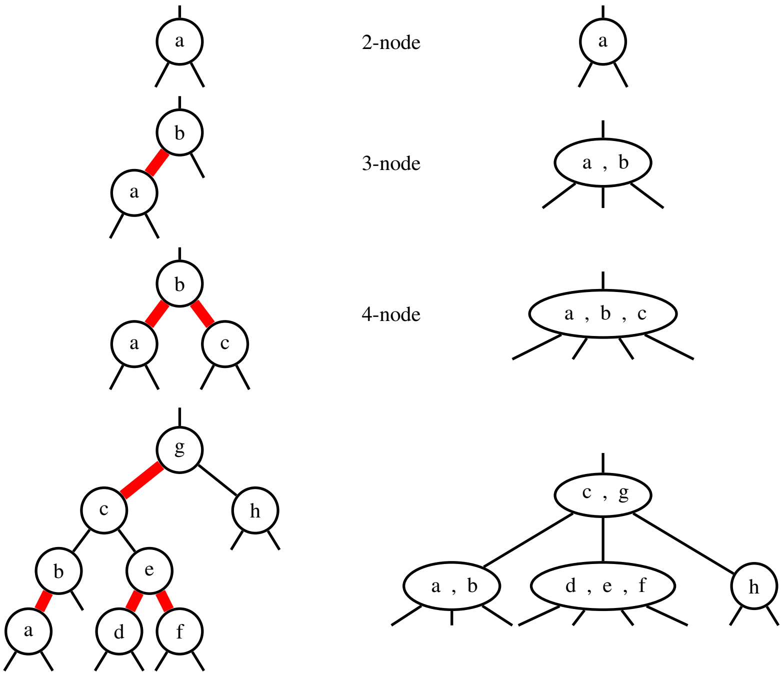

Fig. 1 Isomorphism between LLRB and 2-3-4 trees.

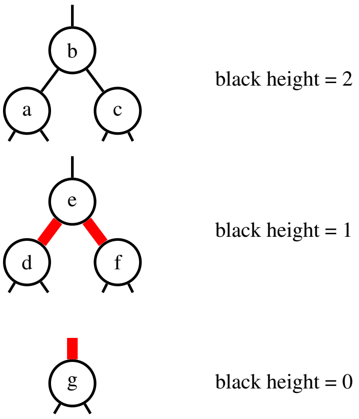

Fig. 2 The *black height* of several trees.

In what follows, and by abuse of the language, we will call in LLRBs 2-nodes, 3-nodes and 4-nodes, to the respective LLRB representations of 2-, 3- and 4-nodes of a 2-3-4 tree. In the case of a 3-node, which is composed of two binary nodes joined by a red link, we will sometimes call 3-node also to the top one of them. Similarly for 4-nodes. When the links of a 2-, 3- or 4-node, other than the red ones, correspond to empty children, we will respectively call to these nodes a 2-leaf, 3-leaf or 4-leaf.

For economy, we will also talk about red or black nodes as abbreviations of binary nodes joined by respectively a red or a black link to their parent nodes. A red node will be part of a 3- or of a 4-node. By convenience, we will assume that there is a link joining a binary node to an empty tree and that its color is black. A *black leaf* is a black binary node with two empty children. Similarly, a *red leaf* is a red binary node with two empty children. A red leaf will always be part of a 3- or of a 4-leaf.

We define the *black height* of an LLRB as the number of black links we find when traversing the tree from the root to any of its leaves. Being more precise, the black height of an empty tree is zero, and the black height of a 2-, 3- or a 4-leaf is one. That is, we do not count the links to empty trees as real black links, but we count as a black link the virtual one going upwards from the root node. This in turns implies that the black

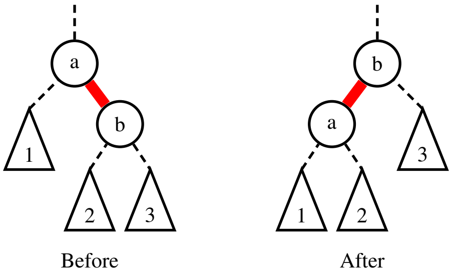

Fig. 3 The rotation-to-the-left transformation.

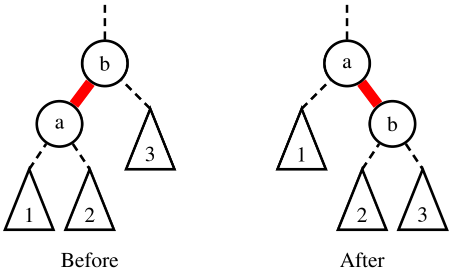

Fig. 4 The rotation-to-the-right transformation.

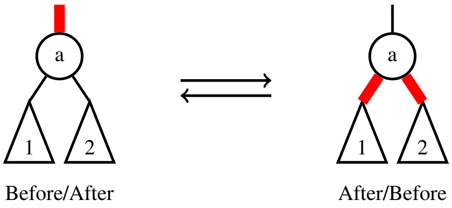

Fig. 5 The color-flip transformation.

height of a black binary leaf is one, but the black height of a red binary leaf is zero. In fact, a red leaf is not a properly formed LLRB, and we should not talk about its height; nevertheless, and by convenience, we will. A consequence of these conventions is that the root of an LLRB is always black. In Fig. 2 we show several (complete and incomplete) LLRB trees with their respective black heights.

We finally present with drawings three important transformations that will be heavily used during insertion and deletion in an LLRB. The first one is *rotation to the left* of a red link. It rotates the link counterclockwise. In Fig. 3 we show the tree before and after the transformation. As it can be applied in many different contexts, we show with dashed lines the links other than the one being rotated, which is required to be red. By this we mean that those links may have any color. However, the color of these links will not be changed by the transformation. In particular, the root will keep its original color. For those familiar with AVL trees [1], this transformation is the one called RR there, because it repairs an unbalance produced by the right child of the right child of the tree.

The second one is *rotation to the right* of a red link. It rotates the link clockwise. In Fig. 4 we show the tree before and after the transformation. This transformation is the one called LL in AVLs, because it repairs an unbalance produced by the left child of the left child of the tree. These transformations preserve at least two properties of the LLRB invariant: the BST property 1(1) and the black height property 1(2), as it can be visually confirmed in figures 3 and 4 .

The last one is *color flip*. It requires the links from the root to its children to have the same color, and this color to be different from the root color. As its name indicates, the colors of these three links are flipped. In Fig 5 we show the two possible trees before and after the transformation. The BST property is trivially preserved, since the tree structure is not modified. So is the black height property, since the number of black links found when the tree is traversed from the top to the bottom is not modified by the transformation.

The importance of these three transformations is that they can be safely applied anywhere in an LLRB, since they do not invalidate the invariant properties 1(1) and 1(2) in any other part of the tree. That is, they are *local* transformations. In order to repair the invariant after the transformation, only care should be taken about properties 1(3) and 1(4).

3 Dafny

Dafny [15] is both a programming language and a verification platform. As a language, it gives support to a variety of programming styles. It has imperative statements such as assignment, conditional, and **while** loops, but it also supports functional programming with algebraic type definitions, pattern matching, and **let** constructs. However, with the exception of a predefined *map* expression, it does not support higher-order. Finally, it also allows object-oriented programming by providing a **class** construct, a notion of dynamically created objects, and a heap. It also supports a limited form of inheritance. The language is strongly typed and admits parametric polymorphism. It provides polymorphic arrays as a predefined type. It can be compiled to .NET code and executed in the .NET platform.

In imperative programming, the programming unit is the **method**, whose body is a sequence of imperative statements. A method receives a number of arguments and produces a tuple of results. Pure functions can be also used by writing the header **function method**. In this case, it produces a single result and its body must be an expression.

With values of an algebraic type, the expression **match** is used for doing pattern matching on the different data constructors. Another way of gaining access to the constructor arguments is by using destructors. For instance, in the definition:

```
datatype BinTree = Empty | Node (left: BinTree, value: int, right: BinTree)
```

if `t` is a variable of type `BinTree`, we can use `t.left` to access to the field `left` of a `Node`. To be sure that `t` is not empty, we can use the query `t.Node?` that returns `true` in case the tree is not empty.

As we said, the body of a function method must be an expression. Expressions include arithmetic and logical ones, function and constructor applications, the **match** expression, and the **if-then-else** expression. Also a kind of **let** expression is supported, having the syntax **var** $x := e_1; e_2$. The value of this expression is that of e_2, but variable x, bound to the value of e_1, may occur free in e_2 and it cannot occur free in e_1. This construction may be iterated, but all the variable definitions should have different names, since it makes no sense to mutate the value of an already defined variable while evaluating an expression.

Dafny allows defining *ghost* code, i.e. functions that are used only for specification purposes. They are defined in the same way as function methods, but omitting the word **method** in the header. Dafny uses their text for reasoning, but it does not generate executable code for them. A **predicate** is just a ghost function returning a Boolean value.

Specifications may include a **requires** φ clause, defining a precondition φ, and an **ensures** ψ clause, defining a postcondition ψ. Several consecutive **requires** clauses are equivalent to a single one having the conjunction of all the preconditions. The same applies to consecutive **ensures** clauses. Both φ and ψ are first-order logic formulas. In a function definition **function** $f(x_1,\ldots,x_n) = e$, only the function arguments $x_1,\ldots,x_n$ may occur free in a precondition φ. In a postcondition ψ, it may occur also the function result, expressed as the term $f(x_1,\ldots,x_n)$.

Several intermediate assertions may occur interspersed among the function code. These are introduced with the syntax **assert** φ; where φ is a logical formula having program variables as free variables. A special case of intermediate assertion is the **invariant** clause. These are used in **while** loops in order to define the loop invariant. Also, a **decreases** clause may be needed to introduce a decreasing expression, in order to prove the loop termination, or the termination of a recursive function. Most of the time, Dafny infers the appropriate decreasing expression for us. If it gets into trouble, then it asks the programmer for an expression. In that case, it tries proving that the expression is bounded from below, and that it decreases at each loop iteration, or at each recursive function application.

Dafny also provides some useful types, such as polymorphic sets, multisets and sequences, to be used in specifications. These are endowed with a bunch of appropriate operations such as union, intersection, difference, and so on, that make it easier specifying and reasoning about complex methods and data structures.

```
datatype LLRB = Empty | Node (left: LLRB, color: Color, key: int, right: LLRB)

datatype Color = Red | Black

predicate isLLRB (t: LLRB)
{
   match t
      case Empty        => true
      case Node(l,c,k,r) => isLLRB(l) ∧ isLLRB(r) ∧ bHeight(l) = bHeight(r) ∧
                            BST(t) ∧ goodColor(t)
}
```

Fig. 6 The Dafny definition of LLRB trees.

```
function tset(t: LLRB): set<int>
{
   match t
      case Empty        => {}
      case Node(l,_,x,r) => tset(l)+{x}+tset(r)
}
predicate promising(l: LLRB, x: int, r: LLRB)
{
   (∀ z • z in tset(l) ⟹ z < x) ∧
   (∀ z • z in tset(r) ⟹ x < z)
}
predicate BST(t: LLRB)
{
   match t
      case Empty        => true
      case Node(l,_,x,r) => promising(l,x,r) ∧ BST(l) ∧ BST(r)
}
```

Fig. 7 The BST predicate.

Finally, Dafny allows defining and proving lemmas, and also provides a form of calculational proof, started by the keyword **calc**, that can be used interspersed with the code in order to deduce a property in a particular text location. Lemmas have a precondition and a postcondition, and can be invoked in the code as if they were methods, and also in the middle of a calculational proof. For more details, the reader may consult the abundant Dafny documentation.[5]

4 Predicates and Ghost Functions

A nice feature of the Dafny language is that the same syntax is used both for defining executable and ghost code, the latter being used for formally specifying the former. We use algebraic data types for defining the type `Color` of colors and the type `LLRB` of binary trees with a color field in each node. This field represents the color of the incoming link. The LLRB invariant is defined by means of the inductive predicate `isLLRB`. These definitions are shown in Fig. 6. They use the auxiliary predicates `BST` and `goodColor`, and the ghost function `bHeight`.

The binary search property `BST` is another inductive predicate, which in turn uses a plain predicate `promising` checking the property only at the top level of the tree. Their definitions are shown in Fig. 7. Notice the use of the auxiliary function `tset` giving the set of elements of a tree.

Function `tset` will be heavily used as *model* specifications of LLRB trees. Following [12,14,23,20], one successful technique for specifying abstract data types (ADT) is the one called model-based, where mathematical objects such as sets, bags, sequences, maps, and so on, are used for establishing the preconditions and postconditions of the ADT methods. This allows us to verify a method separatedly from each other. In our case, there is no doubt that an LLRB of elements is a valid implementation of a set of elements, the method *insert* corresponding to adding to a set, and the method *delete* to removing from a set. So, function `tset` is the so-called *model query* of our ADT, and it will be used in the postconditions of these methods in order to establish the set behavior of LLRBs.

[5] See `https://rise4fun.com/Dafny/tutorial/Guide`.

```
predicate goodColor(t: LLRB)
{
    match t
        case Empty         => true
        case Node(l,c,_,r) =>
               match c
                 case Red   => color(l) = Black ∧ color(r) = Black
                 case Black => color(r) = Red ⟹ color(l) = Red
}
```

Fig. 8 Specifying the color properties of LLRB.

```
function bHeight(t: LLRB) : nat
{
  match t
     case Empty         => 0
     case Node(l,c,_,r) => match c
                             case Black => 1 + max (bHeight(l), bHeight(r))
                             case Red   => max (bHeight(l), bHeight(r))
}
```

Fig. 9 The black height of an LLRB.

An LLRB satisfies `goodColor` at a certain level of the tree if there are no two consecutive red links and no isolated red links to the right. For checking the first property, we need to know the link colors of two consecutive levels. This is done in Fig. 8 by comparing the color of the incoming link of a node with the color of the outgoing ones. If the incoming link is red, there is no other possibilities for the outgoing ones that being black. Otherwise, all color combinations are allowed, except the one with a red link to the right and a black one to the left. The function `color` is not shown but, as expected, it returns the color field of a node, or the color `Black` in the case of an empty tree.

Notice that this `goodColor` definition is satisfied by some additional trees other than those of Definition 1. In particular, a red leaf, and in general those trees having a red root node and two black children satisfy the predicate. These are good news because such trees will arise during insertion and deletion, and we will be glad to consider them as proper LLRBs.

The `bHeight` function gives us the highest number of black links along a tree path from the root node to the leaves. Its definition, shown in Fig. 9, follows the conventions established in Sec. 2: it counts the root upwards link, and it does not count the links to empty trees. Its use in the inductive predicate `isLLRB` of Fig. 6 forces the tree to have the same black height in every path.

Finally, some auxiliary functions, such as `height` and `card`, respectively returning the full height of a binary tree —i.e. without distinguishing between red and black links— and its cardinality, will be needed in order to prove termination of some of the algorithms. Their respective codes are not shown.

5 The *insert* Function

The insertion in an LLRB proceeds in much the same way as that of a binary search tree: in the recursive cases it proceeds by inserting the element in the left or in the right subtree, should the element be respectively smaller or greater than the root. This process may end up in one of two basic cases:

- either the element is present in the tree, then a root is found with its key being equal to the element. In this case, the insertion usually finishes without modifying the tree.
- or the insertion runs on an empty tree. Then, a new leaf is created and appended to its parent.

If no special care were taken, then adding a new leaf to the tree would violate property 1(2). To avoid this, the strategy proposed in [3] is to always create a red leaf. The idea is inspired in that of 2-3-4 trees: a 2-leaf at the bottom of the tree may be transformed into a 3-leaf by adding a new key. Similarly, a 3-leaf can be transformed into a 4-leaf. In terms of red-black trees, this idea can be implemented by linking the new leaf to its parent by a red link.

```
function method mayFlipColors(t: LLRB): LLRB
requires isLLRB(t)
ensures  isLLRB(mayFlipColors(t))
ensures  tset(mayFlipColors(t)) = tset(t)
{
  match t
     case Empty          => Empty
     case Node(l,c,x,r) => if  color(l) = Red ∧ color(r) = Red
                           then colorFlip(t)
                           else t
}
```

Fig. 10 Dafny code of function *mayFlipColors*.

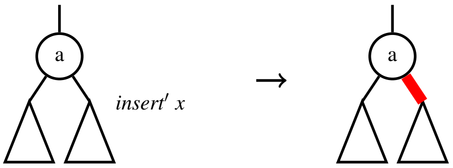

Fig. 11 Creation of an isolated red link to the right.

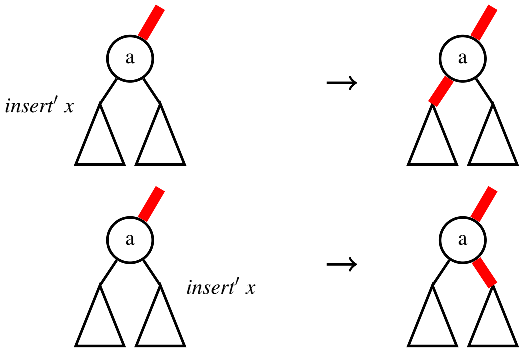

Fig. 12 Creation of two consecutive red links.

Clearly, this process cannot go on forever. At some point, the insertion will face the problem of making a 4-leaf grow. As this is difficult to manage, the strategy of [3] is to maintain, while descending along the tree, the invariant that the current tree root is never a 4-node. If a 4-node were found, it is immediately transformed into two 2-nodes, before proceeding with the insertion. A 4-node has three keys and four children, while two 2-nodes have also four children, but only two keys. So, the middle key is injected upwards into the parent node. This is not a problem because, by the invariant, the parent is never a 4-node.

In an LLRB, this strategy can be very easily accomplished by applying function *colorFlip* to the current tree. As we saw in Sec. 2, this transformation preserves the tree black height. But, at the same time, it propagates a red link upwards in the tree. This joins the current node to its parent with a red link, so making the parent grow. We have implemented this checking and transformation in the function *mayFlipColors*. Its Dafny code and formal specification are shown in Fig. 10. The code of *colorFlip* is not shown.

While the trees given as an argument to the insertion function are always LLRBs —because they are children of LLRBs, and the *colorFlip* transformation when applied to 4-nodes does not modify this property in the children trees—, the parent or grandparent trees may lose their character of being LLRB. This is because the red links left ‘behind’ may create isolated red links to the right, so violating property 1(4), or even worse, they may create two consecutive red links, so violating property 1(3).

As we were not sure that the recursive insert always returned an LLRB, we distinguished between the visible exported *insert* function, which always returns an LLRB, and the recursive hidden one, we call it *insert*′, which may return other kinds of trees. As we will show, this decision has conveyed important benefits to the verification process.

```
function method rightRed(t: LLRB): LLRB
{
  match t case Node(l,c,x,r) =>
           if color(r) = Red ∧ color(l) = Black
              then rotateLeft(t)
              else t
}
function method twoRed(t: LLRB): LLRB
{
  match t case Node(l,c,x,r) =>
           if color(l) = Red ∧ l.Node? ∧ color(l.left) = Red
              then rotateRight(t)
              else t
}
function method checkColors(t: LLRB): LLRB
{
   twoRed(rightRed(t))
}
```

Fig. 13 Dafny code of functions *rightRed*, *twoRed*, and *checkColors*.

```
function method insert'(x: int, t: LLRB): LLRB
{
  match mayFlipColors(t)
    case Empty          => Node(Empty,Red,x,Empty)
    case Node(l,c,y,r) =>
      if x < y then checkColors(Node(insert'(x,l),c,y,r))
               else if x > y then checkColors(Node(l,c,y,insert'(x,r)))
                             else /* x = y */ t
}
```

Fig. 14 Dafny code of function *insert′*.

The strategy followed by [3] to repair the invariant is doing it in the way up in the tree, that is after the recursive calls to the *insert′* function. By studying the different cases, an isolated red link to the right may arise when inserting to the right child of a 2-node (see Fig. 11). Two consecutive red links may arise when inserting to the left, or to the right, of a red node (see Fig. 12). We make note that the red node before the insertion is always a left child of its parent. This is because 4-nodes have been previously transformed into 2-nodes, and so the only red links remaining belong to 3-nodes.

We deal with the first situation by introducing a function *rightRed*. Given a tree, this function rotates to the left an isolated red link to the right, if one exists. This has the additional benefit of converting the two cases of consecutive red links shown in Fig. 12, into a single case: two consecutive red links to the left. This is checked upwards in the tree by another function *twoRed*. The solution to the problem, as explained in [3], is to rotate the top red link to the right. This creates a 4-node, which are allowed in LLRBs. These two actions —*rightRed* and *twoRed*— are called in this order by function *checkColors*. So, the repairing actions in our code consist of calling this latter function after inserting to the right or to the left of an LLRB, and before returning the result. The Dafny code of these three functions is shown in Fig. 13. The code of *insert′* is shown in Fig 14.

The formal specification

An observation about the described behavior is that the LLRB invariant may be temporarily broken during the insertion. As we have explained, sometimes isolated red links to the right may be created. Fortunately, these can be immediately repaired by a *rotateLeft* action before returning the result. So, the involved *insert′* call is able to repair the invariant and to return a properly formed LLRB. The situation is different when two consecutive red links to the left are created: there are two nested calls to *insert′* involved. The external one is able to repair the invariant by using a *rotateRight* action, but the internal one is forced to return an ill-formed LLRB.

We have defined the predicate *weakLLRB* to describe these new trees. They are LLRB in all respects except perhaps in the relationship between the root upward link and the left downwards one. This relationship is not checked, so both could be red and the predicate would still hold. Its formal definition is shown in Fig. 15.

```
predicate weakLLRB (t: LLRB)
{
   match t
      case Empty        => true
      case Node(l,c,k,r) => isLLRB(l) ∧ isLLRB(r) ∧ BST(t) ∧ bHeight(l) = bHeight(r) ∧
                            (color(r) = Red ⟹ color(l) = Red)
}
```

Fig. 15 The *weakLLRB* predicate.

```
function method insert(x: int, t: LLRB): LLRB
requires isLLRB(t)
ensures isLLRB(insert(x,t))
ensures  tset(insert(x,t)) = tset(t) + {x}
{
   blacken(insert'(x, blacken(t)))
}
function method insert'(x: int, t: LLRB): LLRB
requires isLLRB(t)
ensures color(t) = Black ⟹ isLLRB(insert'(x,t))
ensures color(t) = Red   ⟹ (weakLLRB(insert ‘(x,t)) ∧ color(insert'(x,t)) = Red)
ensures bHeight(t) = bHeight(insert'(x,t))
ensures  tset(insert'(x,t)) = tset(t) + {x}
decreases height(t)
```

Fig. 16 The *insert* function and the *insert′* specification.

A first approach to the *insert′* specification is to require its argument to be an LLRB, and to ensure that its result is at least a weak LLRB. But, the verification process led us to a stronger statement: if the argument is black-rooted, we can assert that the output will always be an LLRB; but should the input have a red root, then only *weakLLRB* could be asserted for the result.

Another observation is that the black height of the tree is not increased after *insert′*, as we have anticipated by requiring to always append a red leaf to the tree. But it is important to make note that the black height is not increased either by the splitting of 4-nodes taking place during the way down in the tree, or the repairing actions taking place in the way up. This is because all of them consist of black height preserving actions such as *rotateLeft*, *rotateRight*, and *colorFlip*.

But this contradicts the intuitive idea that, sooner or later, inserting in a tree will make its black height grow. Where is the mystery? The crucial observation is that, given a black-rooted LLRB, function *insert′* will return an LLRB, but its root may be red. If we wished to do a new insertion, we should first paint its root black. Otherwise, we will not be sure to receive an LLRB as a result. Painting black a red root will increase the tree height by one.

The solution we have found has been to have a visible *insert* function that always returns an LLRB, and that is a wrapper for *insert′*: first it ensures a black root, then it calls *insert′*, and finally it paints black the root of the resulting tree. The formal specification of both *insert* and *insert′*, and the code of the first one, are shown in Fig. 16. The code for *blacken* is not shown.

The proof

Once we have presented the specification and the code of *insert′*, we summarize here the main problems encountered during its assertional proof of correctness performed in the Dafny platform.

A first issue was proving termination. Structural decreasing of the input tree t is not guaranteed because it may be transformed into a different one by *mayFlipColors*(t). As a consequence, Dafny is unable to find a properly decreasing function. It is clear for us that flipping colors does not modify the structure or the cardinality of the tree. So, we have added the property `"height(colorFlip(t)) == height(t)"` to the postcondition of *colorFlip*, and instructed Dafny to use `height(t)` as decreasing function of *insert′*. This information is enough for it to prove termination. We recall that `height(t)` returns the full height of an LLRB, i.e. including red links.

```
function method rightRed(t: LLRB): LLRB
requires BST(t)
requires t.Node?
requires bHeight(t.left) = bHeight(t.right)
requires if color(t) = Black ∧ color(t.left) = Red ∧ t.left.Node? ∧ color(t.left.left) = Red
         then weakLLRB(t.left) ∧ color(t.right) = Black ∧ color(t.left.right) = Black
         else isLLRB(t.left)
requires isLLRB(t.right)
requires ¬(color(t) = Red ∧ color(t.left) = Red ∧ color(t.right) = Red)
requires (color(t) = Black ∧ ¬(color(t.right) = Red ∧ color(t.left) = Black) ∧
             ¬(color(t.left) = Red ∧ t.left.Node? ∧ color(t.left.left) = Red))
           ⟹ isLLRB(t)
requires (color(t) = Red ∧ color(t.left) = Red ∧ color(t.right) = Black)
           ⟹ weakLLRB(t)
ensures rightRed(t).Node?
ensures (color(t) = Black ∧ ¬(color(t.left) = Red ∧ t.left.Node? ∧
         color (t.left.left) = Red)) ⟹ isLLRB(rightRed(t))
ensures (color(t) = Black ∧ color(t.left) = Red ∧ t.left.Node? ∧ color (t.left.left) = Red)
           ⟹ (weakLLRB(rightRed(t).left) ∧ isLLRB(rightRed(t).right))
ensures color(t) = Red ⟹ weakLLRB(rightRed(t))
ensures mirrorLLRB(t) ⟹ isLLRB(rightRed(t))
ensures isLLRB(t) ⟹ rightRed(t) = t
ensures tset(rightRed(t)) = tset(t)
```

Fig. 17 The *rightRed* specification.

```
function method twoRed(t: LLRB): LLRB
requires BST(t)
requires t.Node?
requires ¬(color(t) = Red ∧ color(t.right) = Red)
requires (color (t.left) = Red ∧  t.left.Node? ∧ color (t.left.left) = Red)
           ⟹ (color(t) = Black ∧ color(t.right) = Black ∧ color(t.left.right) = Black)
requires (color(t) = Black ∧ ¬(color(t.left) = Red ∧ t.left.Node? ∧
         color (t.left.left) = Red)) ⟹ isLLRB(t)
requires (color(t) = Black ∧ color(t.left) = Red ∧ t.left.Node? ∧ color (t.left.left) = Red)
           ⟹ (weakLLRB(t.left) ∧ isLLRB(t.right))
requires color(t) = Red ∧ color(t.left) = Red     ⟹  weakLLRB(t)
requires color(t) = Red ∧¬(color(t.left) = Red) ⟹  isLLRB(t)
requires bHeight(t.left) = bHeight(t.right)
ensures twoRed(t).Node?
ensures if color(t) = Red ∧ color(t.left) = Red ∧ t.left.Node? ∧ color(t.left.left) = Black
        then weakLLRB(twoRed(t)) else isLLRB(twoRed(t))
ensures tset(twoRed(t)) = tset(t)
```

Fig. 18 The *twoRed* specification.

The main difficulty of the assertional proof of *insert*′ has been to find appropriate specifications for the auxiliary functions. In particular, *twoRed* and *rightRed* have rather complex preconditions and postconditions. This is due in part to the strange trees they receive as an argument: they are neither LLRB nor weak LLRB. Additionally, *rightRed* is also used during deletion. This fact multiplies the kind of situations it must take into account in the argument, and the number of different trees it produces as a result. We show their formal specification in figures 17 and 18. Notice that ensuring that the black height of the result is balanced does not occur in the postcondition, but this is implied by the stronger postcondition stating that the result is either an LLRB or a weak LLRB.

In order to verify *insert*′, Dafny needs the help of some intermediate assertions to bridge the distance between the precondition and the postcondition. For instance, for establishing that the red leaf added when t is empty is a proper LLRB, it needs to first prove that a red leaf is a BST. So, an assertion with this property was added to this branch. The complete annotated code of *insert*′ is shown in Fig. 19.

6 The *delete* Function

The deletion in an LLRB proceeds also in the same way as that of a binary search tree: the recursive cases delete the element in the left or in the right subtree, should the element be respectively smaller or greater than the root. This process may end up in one of two basic cases:

```
function method insert'(x: int, t: LLRB): LLRB
{
  match mayFlipColors(t)
     case Empty          => /**/ assert BST(Node(Empty, Red, x, Empty));
                            Node(Empty, Red, x, Empty)
     case Node(l,c,y,r) =>
          if x < y then     /**/ assert color(t) = Black ∧ t.Node? ∧ color(t.right) = Black
                                     ⟹ isLLRB(checkColors(Node(insert'(x, l), c, y, r)));
                            /**/ assert color(t) = Black ∧ t.Node? ∧ color(t.right) = Red
                                     ⟹ c = Red ∧ color(l) = color(r) = Black;
                            /**/ assert c = Red ∧ color(t) = Black
                                     ⟹ (color(t.right) = Red ∧ isLLRB(t.left)
                                          ∧ t.left.Node? ∧ color(t.left.left) = Black);
                            checkColors(Node(insert'(x, l), c, y, r))
                    else if x > y then
                            /**/ assert isLLRB(l);
                            /**/ assert c = Black
                                     ⟹  isLLRB(checkColors(Node(l, c, y, insert'(x, r))));
                            /**/ assert c = Red ∧ color(t) = Black
                                     ⟹ (color(t.right) = Red ∧ isLLRB(t.right)
                                          ∧ t.right.Node? ∧ color(t.right.right) = Black);
                            /**/ assert c = Red ∧ color(t) = Black
                                     ⟹ isLLRB(checkColors(Node(l, c, y, insert'(x, r))));
                                       checkColors(Node(l, c, y, insert'(x, r)))
                                 else  /* x = y */ t
}
```

Fig. 19 The assertional proof of *insert'*.

- either the element is not present in the tree; this happens when the deletion function is given an empty tree as an argument, and then the deletion finishes without modifying it.
- or the element is found in a node, and this key is removed from the tree.

The strategy proposed in [3] when the target is a leaf is to always remove a red one. In this way, the black-height of the path leading to this leaf will not decrease, and property 1(2) will be preserved. When the target is an internal node, its contents is replaced with the contents of the minimum element of the right subtree, and this minimum element is deleted instead. It could equally be chosen to delete the maximum element of the left subtree. In this way, the binary search property is preserved and the code has only to deal with removing a leaf.

As in the case of *insert*, we distinguish between the visible *delete* and the hidden recursive delete function *delete'* doing the actual work, the first being a wrapper for the second.

In order not to remove a black leaf, the strategy is to ensure that *delete'* is never applied to a 2-node. Then, when the recursive call descends to a left or to a right subtree, and its root is a 2-node, some transformations should be done to enforce this invariant. The strategy followed by [3] is inspired in that of 2-3-4-trees: some key must be injected in the child, either from the parent node or from a sibling tree.

In terms of red-black trees, this is equivalent to enforce that, either the subtree *delete'* is applied to has a red root (it is not a 2-node), or one of its children is red (it is not a black leaf). The invariant while descending can be expressed as follows: *delete'* is always applied to a tree whose root has a red link to its parent, or a red link to at least one of its children.

If the recursive call is to be applied to the left subtree, *delete'* checks whether its root is a 2-node. If it is, then the function *ensureRedLeft* takes care of ensuring the invariant to the left: first a *colorFlip* is applied to the parent; this amounts to projecting a red link to both children, because the parent is red (this, in turn, is because it satisfies the invariant, and both of its children are black); then, it checks whether two consecutive red links have been created; this may happen when the left child of the right child is red; then, a *rotateRight* of this red link solves part of the problem; a new check must be done to see whether its sibling link before the rotation was also red; if so, a *rotateLeft* of this link after the *rotateRight* of its brother is performed; a final *rotateLeft* of the parent, and a second *colorFlip*, ensures the desired invariant.

In Fig. 20 we show the process with drawings and in Fig. 21 we show the Dafny code of *ensureRedLeft*, and its formal specification. What it is going on under this transformation is to borrow a key from a sibling, which happens to be a 3- or a 4-node. The final result is descending to a tree with a 3-node at its root.

The formal specification carefully establishes the shape of the output tree in the two possible cases: when a single color flip suffices, and when the whole sequence of rotations and a second color flip are needed. In both cases, the resulting tree is an LLRB one, the black height is preserved, and also does the set of keys. The last

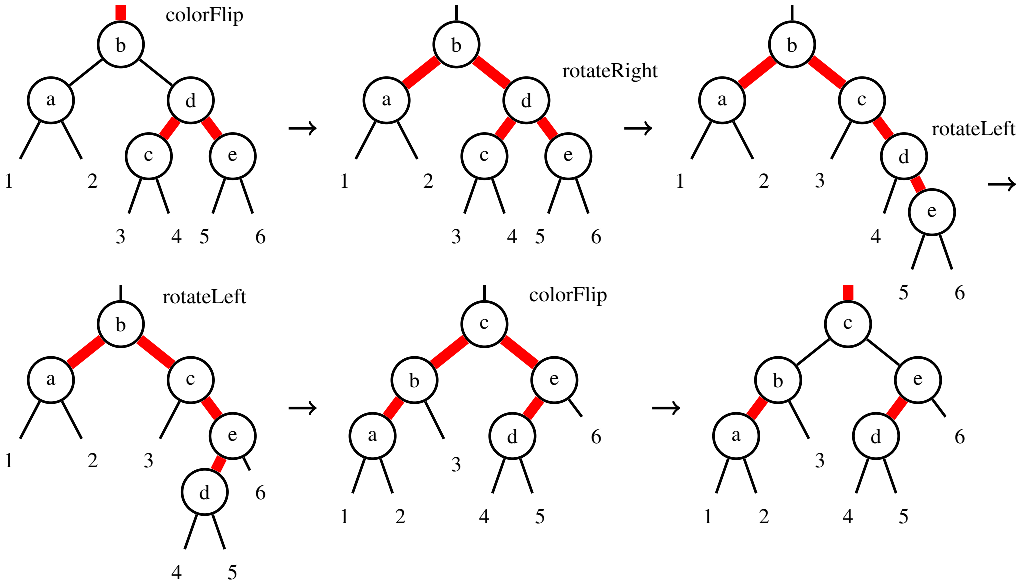

Fig. 20 The worst case of function *ensureRedLeft*.

```
function method ensureRedLeft(t: LLRB): LLRB
requires isLLRB(t) ∧ t.Node? ∧ color(t) = Red
requires t.left.Node? ∧ color(t.left.left) = Black
ensures color(ensureRedLeft(t)) = Black
        ⟹ (ensureRedLeft(t).Node? ∧ color(ensureRedLeft(t).left) = Red)
ensures color(ensureRedLeft(t)) = Red
        ⟹ (ensureRedLeft(t).Node? ∧ color(ensureRedLeft(t).left) = Black ∧
             ensureRedLeft(t).left.Node? ∧ color(ensureRedLeft(t).left.left) = Red ∧
             ensureRedLeft(t).key > t.key ∧ ensureRedLeft(t).left.key = t.key)
ensures isLLRB(ensureRedLeft(t))
ensures bHeight(ensureRedLeft(t)) = bHeight(t)
ensures tset(ensureRedLeft(t)) = tset(t)
ensures card(ensureRedLeft(t).left) < card(t)
{
   var t' := colorFlip(t);
   match t'
     case Node(l,c,y,r) => if color(r.left) = Red
                           then var r'  := rotateRight(r);
                                var r'' := match r' case Node(rl,c'',ry,rr) =>
                                                 if color(rr.right) = Red
                                                 then Node(rl,c'',ry,rotateLeft(rr))
                                                 else r';
                                var t'' := rotateLeft(Node(l, c, y, r''));
                                colorFlip(t'')
                           else t'
}
```

Fig. 21 The Dafny code and specification of *ensureRedLeft*.

clause, `card(ensureRedLeft(t).left) < card(t)`, is needed to ensure the termination of *delete*′, because it will be recursively applied to left child of *ensureRedLeft*(t), and Dafny cannot guess a decreasing function for *delete*′ when several rotations have been applied to its argument.

We would like to remark at this point that the need for the first *rotateLeft* arose during verification, because Dafny was unable to prove that the tree resulting from the whole transformation was indeed an LLRB. This rotation is not present in Sedgewick's code [3] and we believe that this absence is an actual bug of this code, should it be intended for LLRBs. The authors claim that the code is intended for LLRB representing either 2-3-trees or 2-3-4-trees, but in fact it does not work for the latter. Its immediate consequence is that isolated red links to the right may remain in the tree after deletion. This does not invalidate the BST property, but it probably has an effect on the correct balance of the tree, so affecting the efficiency of the subsequent operations.

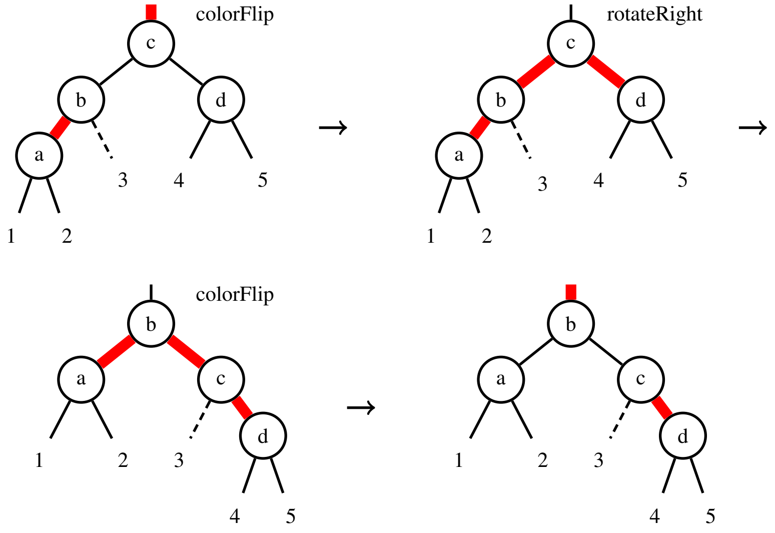

Fig. 22 The worst case of function *ensureRedRight*.

```
function method ensureRedRight(t: LLRB): LLRB
requires isLLRB(t) ∧ color(t) = Red
requires t.Node? ∧  t.right.Node? ∧ color(t.right.left) = Black
ensures color(ensureRedRight(t)) = Black
        ⟹ (isLLRB(ensureRedRight(t)) ∧ ensureRedRight(t).Node? ∧ color(ensureRedRight(t).right) = Red)
ensures color(ensureRedRight(t)) = Red
        ⟹(ensureRedRight(t).Node? ∧ isLLRB(ensureRedRight(t).left) ∧
           (isLLRB(ensureRedRight(t).right) ∨ mirrorLLRB(ensureRedRight(t).right)) ∧
            ensureRedRight(t).right.Node? ∧ color(ensureRedRight(t).right.right) = Red)
ensures bHeight(ensureRedRight(t)) = bHeight(t)
ensures tset(ensureRedRight(t)) = tset(t)
ensures card(ensureRedRight(t).right) < card(t)
{
   var t' := colorFlip(t);
   if color(t'.left.left) = Red
      then colorFlip(rotateRight(t'))
      else t'
}
```

Fig. 23 The Dafny code and specification of *ensureRedRight*.

The cases in which the element to be deleted is greater than, or equal, to the root key must be dealt with together. This is because should the root be deleted, and should it be an internal node, the deletion of the minimum element would be applied to the right subtree. This —to delete an element in the right subtree— is the same action that would be taken should the element be greater than the root key.

The transformation in these two cases is as follows: first, it is checked whether there is an isolated red link to the left; if so, the tree is rotated to the right. This action has two benefits: it ensures that the root of the right child is red, so the invariant holds to the right; and also, it avoids deleting an internal node with an empty right child and a non-empty left one. If no rotation is done, then it is checked whether the right child is a 2-node. If it is, the *ensureRedRight* function takes care of the problem: it applies *colorFlip* to the parent; due to the invariant, and also to the fact that there is no red link to the left, the parent must be red; so, the *colorFlip* projects a red link to both children; a second check is done to ensure that no two consecutive red links have been created; if so, a *rotateRight*, followed by a second *colorFlip*, solves the problem.

In Fig. 22 we show the *ensureRedRight* process with drawings, and in Fig. 23 we show the Dafny code and its formal specification. What it is going on under this transformation is borrowing one or two keys from a sibling, which happens to be a 3- or a 4-node. The final result is applying *delete*$'$ to a tree with a 3- or a 4-node in its root. If it were a 3-node, it would be a strange one, as it will have a black left child and a red right one, something that is forbidden in LLRB trees. We will call these trees *mirrorLLRBs*.

```
function method delete'(x: int, t: LLRB): LLRB
{
  match t
   case Empty          => t
   case Node(l,c,y,r) =>
        if x < y
           then var t' :=  if color(l) = Black ∧ l.Node? ∧ color(l.left) = Black
                           then ensureRedLeft(t)
                           else t;
                match t'
                  case Node(ll,lc,ly,lr) => rightRed(Node(delete'(x, ll), lc, ly, lr))
           else /* x >= y */
                var t'  := if color(l) = Red ∧ color(r) = Black
                           then rotateRight(t)
                           else if color(r) = Black ∧ r.Node? ∧ color(r.left) = Black
                                then ensureRedRight(t)
                                else t;
                match t'
                  case Node(rl,rc,ry,rr) =>
                        if x > ry then rightRed(Node(rl, rc, ry, delete'(x, rr)))
                                  else match rr        /* x = y */
                                         case Empty          => Empty
                                         case Node(_,_,_,_) =>
                                                var z   := minimum(rr);
                                                var rr' := delete'(z, rr);
                                                rightRed(Node(rl, rc, z, rr'))
}
```

Fig. 24 The Dafny code of *delete′*.

```
predicate mirrorLLRB (t: LLRB)
{
   match t
      case Empty          => true
      case Node(l,c,k,r) => isLLRB(l) ∧ isLLRB(r) ∧ bHeight(l) = bHeight(r) ∧ BST(t) ∧
                             c = Black ∧ color(r) = Red ∧ color(l) = Black
}
```

Fig. 25 The predicate *mirrorLLRB*.

The formal specification establishes the shape of the output tree in the two possible cases: when a single color flip suffices, and when a rotation and a second color flip are needed. Also, that the black height property and the set of keys are preserved. The final clause, as in *ensureRedLeft*, is needed to ensure the termination of *delete′*.

After applying *delete′* either to the left or to the right subtree, some repairing actions must be taken regarding properties 1(3) and 1(4). We will reason below that only 1(4) may be violated. So, a call to *rightRed* after deletion, a repairing function also used in the *insert′* code, will take care of the problem. The complete Dafny code of *delete′* is shown in Fig 24. The code for *minimum* is not shown.

The formal specification

Trying to fix the specification of *delete′*, we made the following observations:

- The tree passed as argument to *delete′* is not always an LLRB. As we have seen, in some cases it may be one with a strange 3-node at the root. We define the predicate *mirrorLLRB* to describe this kind of trees. It is shown in Fig. 25. In the precondition, we then admit a *mirrorLLRB* as a legal input to *delete′*.
- The precondition cannot be guaranteed in the first call to *delete′*. For instance, if the initial tree has a 2-node at its root, there is no red link to start with. We solved this problem by creating a wrapper *delete* function that checks the tree, and before calling *delete′*, it paints red its root in case it is a 2-node. Moreover, it paints black the root of the tree returned by *delete′*, just to prevent the case in which the root still keeps a red color.

Painting the root red decreases the tree black height by one. If the returned tree still has a red root, then painting it black will increase the black height again. This would mean that the whole deletion has not modified the black height. On the contrary, if the result tree has a black root, painting it black will not modify the black height.

```
function method delete(x: int, t: LLRB): LLRB
requires isLLRB(t)
ensures  isLLRB(delete(x,t))
ensures  tset(delete(x,t)) = tset(t) − {x}
{
 match t
   case Empty          => Empty
   case Node(l,_,_,r) => if color(l) = Black ∧ color(r) = Black
                         then blacken(delete'(x, redden(t)))
                         else delete'(x,t)
}
function method delete'(x: int, t: LLRB): LLRB
requires isLLRB(t) ∨ (t.Node? ∧ x ≥ t.key ∧ mirrorLLRB(t))
requires t.Node? ⟹ (color(t) = Red ∨ color(t.left) = Red ∨ color(t.right) = Red)
ensures  color(t) = Black ⟹ color(delete'(x,t)) = Black
ensures  isLLRB(delete'(x,t))
ensures  bHeight(delete'(x,t)) = bHeight(t)
ensures  tset(delete'(x,t)) = tset(t) − {x}
```

Fig. 26 The *delete* function and the *delete*$'$ specification.

This would mean that the whole deletion has decreased the black height by one, as one can expect for a delete function.

This observation is important for establishing the *delete*$'$ postcondition: we can state that it preserves the tree black height. This should not be surprising since all the transformations done in its code (namely, *colorFlip*, *rotateLeft*, and *rotateRight*) are black height preserving. This property is also needed for proving, that after deleting an element in one child of an LLRB, the resulting tree is still balanced with respect to the other sibling.

After recursively applying *delete*$'$ to a child tree, and should this child have a red root, we should expect either a red-rooted tree, or a black-rooted one. In the second case, composing this child with the unmodified sibling may result in a bad-colored LLRB. In particular, an isolated red link may remain to the right. This justify the use of function *rightRed* after recursively deleting in a child.

Differently to the *insert*$'$ function, *delete*$'$ always returns an LLRB tree. We show the complete Dafny specification of both *delete* and *delete*$'$, and the code for *delete*, in Fig. 26. The code for *redden* is not shown.

The proof

We summarize here the main problems encountered during the *delete*$'$ assertional proof of correcteness performed in the Dafny platform. The annotated code is shown in Fig. 27.

As in the *insert*$'$ proof, a first issue was proving termination. Structural decreasing of the input tree t is not guaranteed because t is transformed in different ways before applying *delete*$'$ to one of its children. Dafny complains and asks for a decreasing function. We proved that a *rotateLeft* action decreases the cardinality of the left subtree with respect to that of its parent, and a *rotateRight* action decreases the cardinality of the right one. So, providing a `decreases card(t)` clause to Dafny solves this problem.

A second issue was dealing with the algebraic type `Color` in **if** expressions. If one writes "`if color(t) == Red then ...`", Dafny assumes that "`!(color(t) == Red)`" holds in the **else** branch, but it cannot prove without help that this is equivalent to "`color(t) == Black`". For that reason, a number of auxiliary lemmas were needed to infer properties about colors in certain parts of the text. These lemmas were easily proved using the **match** construction on terms of `Color` type. This construction is aware of the exact number of data constructors the type has.

A third issue was to convince Dafny that the set union and the set difference can be permuted in some cases, i.e. if A, B and C are sets, and B and C are disjoint, then $(A - B) \cup C = (A \cup C) - B$. This justify the intermediate assertions of lines 29-31, 53-55, and 71-74 of Fig. 27.

Apart from that, the proof just consisted of introducing enough intermediate assertions and calculational proofs, in order to assist Dafny in proving the rather complex verification conditions arising from this contrived code. In total, we needed about sixteen intermediate assertions, three calculations, and three auxiliary lemmas. Notice the assertion

```
/**/ assert rl = Empty ∧ rc = Red;
```

```
function method delete'(x: int, t: LLRB): LLRB
decreases card(t)
{
  match t
   case Empty          =>   t
   case Node(l,c,y,r) =>
        if x < y
           then var t' :=   if color(l) = Black ∧ l.Node? ∧ color(l.left) = Black
                           then /**/ assert color(r) = Black ∧ isLLRB(t);
                                calc{
                                       color(l) = Black ∧ l.Node? ∧ color(l.left) = Black;
                                    ⟹ {GoodLeft2(t);}
                                       color(ensureRedLeft(t).left) = Red ∨
                                       color(ensureRedLeft(t).left.left) = Red;
                                    }
                                ensureRedLeft(t)
                           else calc{
                                      ¬(color(l) = Black ∧ l.Node? ∧ color(l.left) = Black);
                                    ⟹ {GoodLeft1(t);}
                                       t.left.Node? ⟹ (color(t.left) = Red ∨
                                       color(t.left.left) = Red);
                                    }
                                t;
                match t'
                  case Node(ll,lc,ly,lr) =>
                                /**/ assert isLLRB(t') ∧ isLLRB(ll) ∧ isLLRB(lr);
                                /**/ assert (color(delete'(x,ll)) = Black ∧ color(lr) = Red)
                                /**/            ⟹ mirrorLLRB(Node(delete'(x, ll), lc, ly, lr));
                                /**/ assert x ≠ ly ∧ ¬(x in tset(lr));
                                /**/ assert ((tset(ll) − {x}) + {ly}) + tset(lr) =
                                            (tset(ll) + {ly} + tset(lr)) − {x};
                                rightRed(Node(delete'(x, ll), lc, ly, lr))
           else /* x >= y */
                var t'  := if color(l) = Red ∧ color(r) = Black
                           then /**/ assert(isLLRB(t))∧ isLLRB(t.left);
                                /**/ assert mirrorLLRB(rotateRight(t));
                                rotateRight(t)
                           else if color(r) = Black ∧ r.Node? ∧ color(r.left) = Black
                                then ensureRedRight(t)
                                else calc{
                                             ¬(color(l) = Red ∧ color(r) = Black) ∧
                                             ¬(color(r) = Black ∧ r.Node? ∧ color(r.left) = Black);
                                          ⟹ {GoodRight(t);}
                                             color(r) = Red ∨ (r.Node? ⟹ color(r.left) = Red);
                                         }
                                     t;
                match t'
                  case Node(rl,rc,ry,rr) =>
                                      /**/ assert ¬rr.Node? ∨ color(rr) = Red ∨
                                                  (rr.Node? ∧ (color(rr.right) = Red ∨
                                                  color(rr.left) = Red));
                       if x > ry then /**/ assert isLLRB(rl);
                                      /**/ assert ¬(x in tset(rl)) ∧ x ≠ ry;
                                      /**/ assert (tset(rr)− {x}) + (tset(rl) + {ry}) =
                                                  (tset(rr)+(tset(rl)+{ry}))− {x} = tset(t') − {x};
                                      rightRed(Node(rl, rc, ry, delete'(x, rr)))
                                 else /* x = ry */
                                      match rr
                                        case Empty          =>
                                                  /**/ assert rl = Empty ∧ rc = Red;
                                                  Empty
                                        case Node(_,_,_,_) =>
                                                  var z   := minimum(rr);
                                                  var rr' := delete'(z, rr);
                                                  /**/ assert z in tset(rr) ∧
                                                              tset(rr) = tset(rr') + {z};
                                                  /**/ assert isLLRB(Node(rl, rc, z, rr')) ∨
                                                  /**/        mirrorLLRB(Node(rl, rc, z, rr'));
                                                  /**/ assert isLLRB(rl) ∧
                                                              (rl.Node? ⟹ isLLRB(rl.left));
                                                  /**/ assert tset(Node(rl, rc, z, rr')) =
                                                              tset(rl) + ({z} + tset(rr')) =
                                                              tset(rl) + tset(rr) =
                                                              tset(t') − {ry} = tset(t') − {x};
                                                  rightRed(Node(rl, rc, z, rr'))
}
```

Fig. 27 The assertional proof of *delete*$'$.

in line 60, where an empty tree is returned. In fact, the assertion is not needed for the proof, but its verified presence there confirms that the algorithm deletes a red leaf (i.e. a node with a red root and two empty children).

7 Experience in using Dafny

The code presented in this paper is rather concise. The functions related only to insertion —*insert, blacken, insert', mayFlipColors, checkColors* and *twoRed*— amount to 28 non-empty lines of code (LOC). The ones related only to deletion —*delete, redden, delete', ensureRedLeft*, and *ensureRedRight*— amount to 51 LOC, and the ones related to balancing, shared by both insertion and deletion, —*rightRed, colorFlip, rotateLeft*, and *rotateRight*— amount to 20 LOC more. If we add the code for *empty, member*, and *minimum*, not shown in the paper, the whole abstract data type totalizes 113 LOC, which is comparable to the Java code of [2] and [22] implementations.

Assertions are a different matter. A total of 650 lines were needed to write preconditions, postconditions, intermediate assertions, decreasing functions, lemmas, proofs, predicates and ghost functions. This gives a ratio of about 6 lines of text devoted to reasoning per line of actual code. These figures are not surprising in the formal verification area. Depending on the problem at hand, sometimes this ratio is higher.

The experience in using Dafny has been very satisfactory. We used the IDE consisting in a Visual Studio editor with a special plugin for Dafny. As we said in the Introduction, the syntax and type checkers, the verification condition generator, and the verifier, are continuously running in the background. This gives the user an immediate feedback about syntax and type errors, and more importantly, about logical errors. The latter are rather informative. For instance, when a postcondition does not hold in a branch of the program, not only that branch is flagged as erroneous, but also an indication of which parts of the postcondition do not hold are also flagged with colors in the text location where the postcondition is defined. A similar approach is followed when a function is called violating its precondition. This information is of big help to understanding what is going on, and sometimes it gives the user useful hints on how to repair the problem.

The time for verifying the whole file is rather high: about 50 seconds after each modification. This is is due to the high number of proof obligations that must be discharged (more than 2000). But this time can be reduced to a minimum by keeping active only the part of the file the user is really working on. This part is usually small —the function being verified and the few ones used by it—, and the rest of the file can be simply converted into a large comment. This makes verification a reasonably responsive interactive process between the user and the verifier.

Regarding the problem itself, we have contradictory feelings about the difficulty of the proof. On the one hand, we believe that the formal verification of LLRB presented here has been very useful in clarifying some properties that were not evident at first sight. In particular, to understand that the recursive *insert'* and *delete'* do not modify the tree height, and that the actual increasing or decreasing of the tree height takes place when painting the tree root of different colors.

On the other hand, verifying red-black trees has been much more costly for this author than for instance verifying AVL trees. The reason is that, while the AVL recursive functions always receive an AVL as an argument and produce an AVL as a result, this is not the case in LLRBs, as we have already shown. This complicates much the reasoning, because the LLRB invariant is sometimes broken between recursive calls, and it is only repaired at an upper or a lower level in the tree. As a consequence, discovering the preconditions and the postconditions of the functions repairing the invariant —namely *rightRed* and *twoRed*— has been a rather painful trial and error process.

8 Conclusions

Efficient data structures play an important role in programming. In particular, red-black trees are heavily used in most implementations of maps, multimaps, sets and multisets, one can find in the standard libraries of C++ and Java. For that reason, it is critical to have correct implementations of them. A bug in an incorrect implementation may cause a damage in the multiple applications deployed, for instance, in tablets or mobile phones, as it has been recently the case of an implementation of the algorithm *Timsort* for the Android platform, that was shown to be buggy by a formal verification conducted using the KeY verification tool[6]. The bug could cause in some cases an array-out-of-bounds exception.

We have contributed here with a formal verification of a variant of red-black trees, and we claim even to have found a small bug in some published implementations of them. Our proof is assertional, and we believe that these kind of proofs have some advantages: not only they are easier to understand by the average programmer, but also, by studying the preconditions and postconditions of the methods, some hidden properties are brought to the surface and they contribute to clarify why the code works.

Another conclusion is that, in the view of this author, the Dafny platform is mature enough to help to conduct complex assertional proofs. It much alleviates the work of the human verifier, by signalling errors with meaningful messages, by providing useful built-in inductive schema, by automatically proving termination in most situations, and by generating and discharging the verification conditions.

The rich theories supported by modern SMT solvers —arrays, algebraic datatypes, arithmetic, sets, multisets, etc.— makes proving the validity of the generated formulas to be a decidable problem in a majority of situations. Only occasionally one receives a timeout as a response, indicating that the formula has not been either proved or disproved by the solver.

We hope also to have contributed to encourage the use of examples such as the one verified here, and the use of friendly verification platforms such as Dafny, in teaching data structures and algorithms to students both with rigour and clarity.

Acknowledgements The author would like to thank Dr. Francisca Lucio and Dr. Narciso Martí for providing useful remarks to a draft version of this paper.

References

1. Adelson-Velskii, U.M., Landis, E.M.: An algorithm for the organization of information. Soviet Math. Doklady pp. 1259–1263 (1962)
2. Anderson, A.: Balanced search trees made simpler. pp. 61–71. Third Workshop on Algorithms and Data Structures (1993)
3. Arge, L., Sedgewick, R., Seidel, R.: 08081 abstracts collection - data structures. In: L. Arge, R. Sedgewick, R. Seidel (eds.) Data Structures, 17.02. - 22.02.2008, *Dagstuhl Seminar Proceedings*, vol. 08081. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2008). URL http://drops.dagstuhl.de/opus/volltexte/2008/1532/
4. Bayer, R.: Symmetric binary b-trees: Data structure and maintenance algorithms. Acta Inf. **1**, 290–306 (1972). DOI 10.1007/BF00289509. URL https://doi.org/10.1007/BF00289509
5. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Inf. **1**, 173–189 (1972). DOI 10.1007/BF00288683. URL https://doi.org/10.1007/BF00288683
6. Bertot, Y., Casteran, P.: Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
7. Bobot, F., Filliâtre, J.C., Marché, C., Melquiond, G., Paskevich, A.: The Why3 platform. Version 0.86.1, University Paris-Sud, CNRS, Inria (2015)
8. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64. Wroclaw, Poland (2011). URL https://hal.inria.fr/hal-00790310
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press (2001)
10. Filliâtre, J., Letouzey, P.: Functors for proofs and programs. In: D.A. Schmidt (ed.) Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, *Lecture Notes in Computer Science*, vol. 2986, pp. 370–384. Springer (2004). DOI 10.1007/978-3-540-24725-8-26. URL https://doi.org/10.1007/978-3-540-24725-8-26
11. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978, pp. 8–21. IEEE Computer Society (1978). DOI 10.1109/SFCS.1978.3. URL https://doi.org/10.1109/SFCS.1978.3
12. Guttag, J.V., Horning, J.J., Garland, S.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science. Springer (1993). DOI 10.1007/978-1-4612-2704-5. URL https://doi.org/10.1007/978-1-4612-2704-5
13. Herbert, L., Leino, K., Quaresma, J.: Using dafny, an automatic program verifier. In: B. Meyer, M. Nordio (eds.) Tools for Practical Software Verification, *Lecture Notes in Computer Science*, vol. 7682, pp. 156–181. Springer Berlin Heidelberg (2012). DOI 10.1007/978-3-642-35746-6-6. URL http://dx.doi.org/10.1007/978-3-642-35746-6-6
14. Jones, C.B.: Systematic software development using VDM (2. ed.). Prentice Hall International Series in Computer Science. Prentice Hall (1991)
15. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: E.M. Clarke, A. Voronkov (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 6355, pp. 348–370. Springer (2010). DOI 10.1007/978-3-642-17511-4-20. URL https://doi.org/10.1007/978-3-642-17511-4-20

[6] Thorough explanation at: `http://www.envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/`.

16. Leino, K.R.M.: Developing verified programs with Dafny. In: B. Brosgol, J. Boleng, S.T. Taft (eds.) ACM Conf. on High Integrity Language Technology, HILT'12, pp. 9–10. ACM (2012)
17. Leino, K.R.M., Lucio, P.: An assertional proof of the stability and correctness of natural mergesort. ACM Trans. Comput. Log. **17**(1), 6:1–6:22 (2015). DOI 10.1145/2814571. URL http://doi.acm.org/10.1145/2814571
18. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008). DOI 10.1007/978-3-540-78800-3-24. URL http://dx.doi.org/10.1007/978-3-540-78800-3-24
19. Okasaki, C.: Red-black trees in a functional setting. J. Funct. Program. **9**(4), 471–477 (1999). URL http://journals.cambridge.org/action/displayAbstract?aid=44273
20. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: N. Bjørner, F.S. de Boer (eds.) FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9109, pp. 414–434. Springer (2015). DOI 10.1007/978-3-319-19249-9-26. URL https://doi.org/10.1007/978-3-319-19249-9-26
21. Reade, C.M.P.: Balanced trees with removals: an exercise in rewriting and proof. Science of Computer Programming **18**, 181–204 (1992)
22. Sedgewick, R., Wayne, K.: Algorithms, 4th Edition. Addison-Wesley (2011)
23. Stepney, S., Cooper, D., Woodcock, J.: More powerful Z data refinement: Pushing the state of the art in industrial refinement. In: J.P. Bowen, A. Fett, M.G. Hinchey (eds.) ZUM '98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, September 24-26, 1998, Proceedings, *Lecture Notes in Computer Science*, vol. 1493, pp. 284–307. Springer (1998). DOI 10.1007/978-3-540-49676-2-20. URL https://doi.org/10.1007/978-3-540-49676-2-20
24. Weiss, M.A.: Data structures and problem solving using Java. Addison Wesley (1998)