Verificación de estructuras de datos arborescentes con iteradores en Dafny

Verification of tree-like data structures with iterators in Dafny

Jorge Blázquez Saborido

Máster en Métodos Formales en Ingeniería Informática
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Trabajo de fin de máster

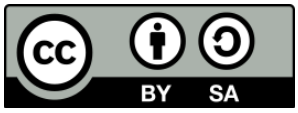
9 de septiembre de 2022

Directores:
Clara María Segura Díaz
Manuel Montenegro Montes

Calificación: 10 (Matrícula de Honor)

El sueño de la razón produce monstruos

Francisco de Goya (1746–1828)



Contents

Resumen

Las estructuras de datos arborescentes son una forma común de implementar tipos de datos importantes que se usan de forma generalizada en muchos lenguajes de programación. Concretamente, los árboles de búsqueda autoequilibrados pueden garantizar coste logarítmico para las operaciones principales, haciéndolos la implementación elegida de muchas bibliotecas de contenedores de datos.

En este trabajo damos una especificación de conjuntos, mapas y multiconjuntos en el lenguaje de programación Dafny. También implementamos y verificamos en Dafny los árboles rojinegros inclinados a la izquierda, un tipo de árbol autoequilibrado, siguiendo la metodología que desarrollamos en trabajo previo. Finalmente, avanzamos en la implementación y verificación de los iteradores para árboles binarios de búsqueda.

Palabras clave: verificación formal, estructuras de datos, árboles autoequilibrados, Dafny

Abstract

Tree-like data structures are a common way to implement important data types that are pervasively used in many programming languages. Specifically self-balancing binary search trees can guarantee logarithmic cost for the main operations, making them the implementation choice of many container libraries.

In this work we give a specfication for sets, maps and multisets in the Dafny programming language. We also implement and verify in Dafny left-leaning red-black trees, a kind of self-balancing binary search tree, following the same methodology that we developed in our previous work. Lastly, we advance on the implementation and verification of iterators for binary search trees.

Keywords: program verification, data structures, self-balancing trees, Dafny

Chapter 1

Introduction

Tree-like data structures constitute an important way to implement key-value stores, the backbone of many applications in daily use today like internet routing, file systems and compilers [11]. In this work we will study the specification of such key-value stores in the form of maps, as well as sets and multisets. We will also approach the problem of implementing and verifying tree-like data structures to implement those data types. In particular we will explore binary search trees and a derivation of them, red-black trees, that has seen wide adoption in standard libraries of common programming languages[2].

Our work is set in the context of formal verification, an established method to prove that the behaviour of a computer program matches its specification. There are several approaches to formal verifications that vary in their degree of automation. In this work we use a programming language, Dafny [7], based on a semiautomated theorem prover. This way we enjoy the automatic verification of the theorem prover while still being able to give more detailed proofs for the cases that it cannot prove automatically.

1.1 Goals

This work is focused on the verification of tree-like data structures, namely binary search trees and red-black trees. Since they are usually employed to implement key-value stores, we also set as a goal the specification of this data type. The definition of that data type is usually accompanied by iterators that traverse the container, so we also set as a goal the specification of iterators and the verification of their implementation on binary search trees.

This document begins with Chapter 2 giving an overview on Dafny and on the verification and specification techniques that we developed in our previous work, needed to understand the current work. Since no verification can be done without a specification, we continue with Chapter 3 exploring the specification of sets, multisets and maps. Next we study the verification and implementation of binary search trees in Chapter 4. They provide the foundation of red-black trees, explored in Chapter 5. In Chapter 6 we close by showing our work on the verification of iterators for binary search trees. Chapter 7 concludes this work.

1.2 Work plan

The work plan for the different tasks of this project was as follows:

- February: define the specification of sets, multisets and maps. Start the implementation and verification of binary search trees.
- March: implementation of sets, multisets and maps with linked lists. Continue implementation and verification of binary search trees.
- March to July: implementation and verification of red-black trees.
- April to July: implementation and verification of iterators for binary search trees.

Chapter 2

Preliminaries

2.1 An overview of Dafny

Dafny [6] is the language we have chosen for this work, continuing the code we wrote in previous work [1]. Dafny is an object-oriented programming language with verification features. We assume knowledge of imperative programming, but the verification utilities that Dafny brings need some introduction. This section is devoted to that.

To begin, in Figure 2.1 we have a small example class that implements a counter. It only has a constructor, a method to increment the counter and a method to retrieve its count. This is a similar program to what we would write in a convential OOP language, but Dafny allows us to introduce specifications to our methods that can be statically verified to hold for all inputs, in all executions. In Figure 2.2, we add such specifications. We define a new predicate that discerns between `Counter` objects that are *valid* and those that are not. Note that this predicate will not be compiled, it is a verification-only definition. The definition of validity is simple: the count should never be less than 0. We verify that the constructor produces a valid object by adding a postcondition via an `ensures`-clause. In the increment method we set as a precondition that the object should be valid via a `requires`-clause, and also ensure that the object is valid after the execution of the method. Finally, we require that the object that `Get` receives is valid. We do not add a postcondition to `Get` that ensures that the counter is still valid since the method does not modify it.

You may have noticed that there are some clauses in predicates and methods, the `reads` and `modifies` clauses, that we have not mentioned yet. These are

```
class Counter {
  var x: int;

  constructor()
  {
    x := 0;
  }

  method Inc()
    modifies this
  {
    x := x + 1;
  }

  method Get() returns (r: int)
  {
    r := x;
  }
}
```

Figure 2.1: Counter class

```
class Counter {
  var x: int;

  predicate Valid()
    reads this
  { x ≥ 0 }

  constructor()
    ensures Valid()
  { x := 0; }

  method Inc()
    modifies this
    requires Valid()
    ensures Valid()
  { x := x + 1; }

  method Get() returns (r: int)
    requires Valid()
    ensures r ≥ 0
  { r := x; }
}
```

Figure 2.2: Counter class with validity

```
predicate Sorted(v: array<int>)
  reads v
{
  ∀ i | 0 ≤ i < v.Length-1 • v[i] ≤ v[i+1]
}

method DoNothing(v: array<int>)
  modifies v
{}

method Main()
{
  var v := new int[3];
  v[0] := 0; v[1] := 1; v[2] := 2;
  assert Sorted(v);

  var w := new int[3];
  w[0] := 0; w[1] := 1; w[2] := 2;
  assert Sorted(w);

  DoNothing(w);
  assert Sorted(v);
  // Dafny cannot prove this assertion:
  // assert Sorted(w);
  // It cannot prove this either:
  // assert ¬Sorted(w);
}
```

Figure 2.3: Framing example

important clauses that make up the *framing* feature. Framing is the way in which Dafny delimits the memory locations modified by methods. By adding a `reads` clause to a predicate, Dafny knows which memory locations may be accessed by the predicate. If some memory space is modified by a method (i.e. it is included in its `modifies` clause) and a predicate reads it, Dafny might not be able to verify if the predicate holds after the execution of the method. If, on the contrary, a predicate holds before the execution of the method and the memory that it reads is not modified, the predicate will still hold. For example, in Figure 2.3 we define a predicate for sorted arrays. We also define a method `DoNothing` that may modify an array. In fact, it does not modify it, but methods in Dafny are *opaque*, that is, outside of the method Dafny will not know its body, it will only take into account its specification (in this case, only that it may modify the input array). So once we pass a sorted array into the `DoNothing` method, Dafny cannot verify that it remains sorted, but we are able to verify that other arrays are still sorted since those have certainly not been modified.

There are some special expressions related with framing which are useful to specify the memory behavior of methods:

- `fresh(x)`: the object `x` has been newly allocated in the body of the method.
- `old(e)`: evaluates to the value of the `e` expression before the execution of the method.
- `unchanged(x)`: every field of object `x` is the same as before the execution of the method. For example, if the object has fields `foo` and `bar`, it would be equivalent to the expression `x.foo==old(x.foo) && x.bar==old(x.bar)`. Note that `x==old(x)` is not the same as `unchanged(x)`, since `x` is a reference and in the first case you are only asserting that the variable `x` is pointing to the same object at the beginning of the execution and at the end, but the fields of the object might have been modified.

Apart from predicates, we can also write other definitions for the specification of our methods. Functions, as opposed to methods, are specification-only definitions. In Figure 2.4 we see how we can use a function to write the specification of a method, namely, of an imperative implementation of the Fibonacci numbers. These functions cannot be compiled, but we can make them compilable by writting `function method` instead of just `function`. This way we can invoke them as if they were methods, but with the guarantee that they do not have any side effects. These function methods, of course, can also be used in

```
function Fib(n: nat): nat
  decreases n
{
  if n = 0 then
    0
  else if n = 1 then
    1
  else
    Fib(n-1) + Fib(n-2)
}

method Fibonacci(n: nat) returns (r: nat)
  ensures r = Fib(n)
{
  var i := 0;
  var x := 0;
  var y := 1;
  while i < n
    decreases n - i
    invariant i ≤ n
    invariant x = Fib(i)
    invariant y = Fib(i+1)
  {
    x, y := y, x + y;
    i := i + 1;
  }
  r := x;
}
```

Figure 2.4: Fibonacci sequence implementation

specifications.

In the example of Figure 2.4 we can also see the `decreases`-clause. This clause is used to help Dafny prove the termination of recursive functions/methods and of loops.

In Dafny we can define the interface of a class without providing any implementation. We, however, can provide the implementation for any functions and/or methods that we wish and the classes that implement the interface will not need to provide such implementation. This feature is usually called just *interface* in some languages. In Dafny and other languages it is called a *trait*. In Figure 2.5 we define a trait for animals, and define two classes that implement such trait. This feature will be very useful when we define *abstract datatypes* (ADTs).

Dafny comes prebundled with some datatypes that we use in the specifications throughout our development. We have the `seq` type (e.g. `[1,2,3]`), a type for immutable lists that can be appended (with the + operator), indexed (`v[i]`), sliced (`v[i..j]`) and converted to a multiset (`multiset(v)`); the

```
trait Animal {
  function NumberOfLegs(): nat
}

class Cat extends Animal {
  function NumberOfLegs(): nat
  {
    4
  }
}

class Bonobo extends Animal {
  function NumberOfLegs(): nat
  {
    2
  }
}
```

Figure 2.5: Animal taxonomy expressed as a trait and classes

`set` type (e.g. `{1, 2, 3}`), a type for immutable sets that has the union operation (+ operator) and difference operation (− operator) and can also be defined in terms of other sets with set comprehensions (`set x | x in s :: x % 2==0`); the `multiset` type (e.g. `multiset{1, 1, 2}`), similar to sets but allowing repeated elements; and the `map` type (e.g. `{"hello" := "world"}`), a key-value store with similar operations as sets. We can get the size of all of those collections by the use of the `|c|` operator.

2.2 Verification methodology

Our methodology is based on the verification of abstract datatypes (ADTs). In our previous work we developed a methodology that evolves from the work of other authors [7, 6]. For brevity, here we will only discuss our methodology. For a comparison with other ones we refer to [1].

To specify and implement ADTs we need three definitions:

- **Representation (footprint)**: the set of objects that each instance owns and uses to implement the interface. Mutators may modify it. It is expressed as a set of objects.
- **Model**: the formal interpretation we give to the ADT. It should be a value type because it is used for verification purposes.

```
trait List {
  function ReprDepth(): nat
    reads this
    ensures ReprDepth() > 0

  function ReprFamily(n: nat): set<object>
    decreases n
    requires n ≤ ReprDepth()
    ensures n > 0 ⟹ ReprFamily(n) ≥ ReprFamily(n-1)
    reads this, if n = 0 then {} else ReprFamily(n-1)

  function Repr(): set<object>
    reads this, ReprFamily(ReprDepth()-1)
  {
    ReprFamily(ReprDepth())
  }

  predicate Valid()
    reads this, Repr()

  function Model(): seq<int>
    reads this, Repr()
    requires Valid()
}
```

Figure 2.6: Example ADT

- **Representation invariant**: it is the property that delimits which instances denote a value of the model. In Dafny it is represented with a predicate that reads the representation.

In Figure 2.6 we show an extract of the interface of a list ADT[1]. It specifies all the definitions that should be fulfilled by the implementation: the representation `Repr`, defined as a function that returns a set of objects; the `Model`, defined as a function that returns a sequence; and the representation invariant, defined in the `Valid` predicate. Note that the representation is defined by a representation family. This way of defining the representation is called *representation stratification*, since it allows us to use ADTs to implement other ADTs, in a way that reminds us of strata or levels. We will explore further how to define this stratification at the end of this section.

Now that we know the interface of our ADT, let us focus on how to implement and verify such an ADT. In Figure 2.7 we show the implementation of a list ADT. First we define the nodes of the list and then the list that points to the first node

[1]The code of that figure and the rest of this section can be found in the `src/linear/` directory of the associated code.

```
class Node<A> {
  var data: A;
  var next: Node?<A>;

  constructor(data: A, next: Node?<A>)
    ensures this.data = data
    ensures this.next = next
  {
    this.data := data;
    this.next := next;
  }
}
```

```
class List<A> {
  var head: Node?<A>;
  ghost var spine: seq<Node<A>>;

  function Repr(): set<object>
    reads this
  { set x | x ∈ spine }

  predicate Valid()
    reads this, Repr()
  {
    ∧ (∀ i | 0 ≤ i < |spine|-1 •
         spine[i].next = spine[i+1])
    ∧ (if head = null then
         spine = []
       else
         ∧ spine ≠ []
         ∧ spine[0] = head
         ∧ spine[|spine|-1].next = null)
  }
}
```

Figure 2.7: Definition of `List` class

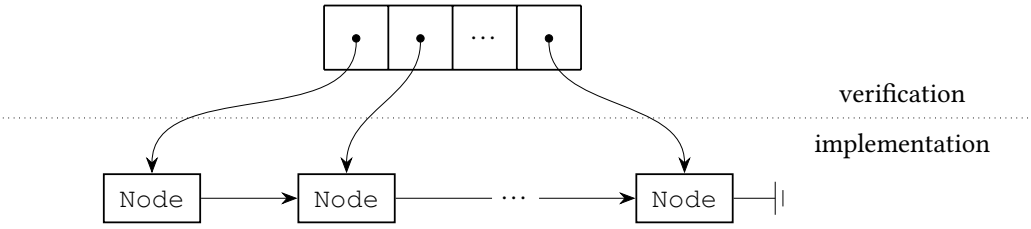


Figure 2.8: Spine and nodes

(called `head`). Note that object types in Dafny are not nullable, so we add an interrogation mark `?` to `Node` to make it nullable.

The `List` class also has a field `spine`. We mark it as *ghost* to make it a verification-only field, that is, it will not be present in memory when the program is executed and the code that modifies it will not be compiled. The `spine` field is a sequence of nodes in the same order as they are linked. We define the representation of our list in terms of the nodes stored in it. Imagine, for example, a list of three or more nodes. In Figure 2.8 we can see at the bottom how those nodes would be present in memory and how, on the top, the spine stores them in order. The `Valid` predicate ensures that the nodes in memory are linked in the same order as in the spine. The `Repr` function, then, just has to return the set of

```
static function ModelAux(xs: seq<Node<A>>): seq<A>
  reads set x | x ∈ xs • x`data
{
  if xs = [] then
    []
  else
    [xs[0].data] + ModelAux(xs[1..])
}

function Model(): seq<A>
  reads this, spine
  requires Valid()
{
  ModelAux(spine)
}
```

Figure 2.9: Definition of `Model` in the `List` class

nodes stored in the spine. This way of implementing the representation is called *structured representation*, as opposed to unstructured representation (having a field with the set of objects). With this methodology we can verify more easily the ADT, since we have easy access to the nodes within the spine. For example, if we want to verify something related to the previous node of a given node in the `i`-th position we can do it with the spine by simple indexing (`spine[i-1]`). If we did not have the spine we would have to traverse the list to access the node and reason about it. While this is not a performance problem (it is verification code that will not be compiled nor executed), it makes the proofs longer and more involved.

To define the model of the list ADT we use a technique called *calculated model*. In Figure 2.9 we can see how the model is computed by traversing the spine each time the `Model` function is applied. We call this technique calculated since other methodologies store the model in a ghost field instead of computing it on-demand.

Until now we have described how to implement and verify our ADT from scratch, but we usually want to reuse ADTs to implement other ones. This is possible in our methodology thanks to the *representation stratification* that we mentioned at the beginning of this section. With this technique the representation is defined as a family of representations built level by level. The first level of the family, 0, is the current object. The family grows from there, level by level, until it reaches the deepest level of nested objects. For example, in Figure 2.10 we illustrate the representation levels of a linked list implementation, along with the corresponding code in Figure 2.11. The ADT has a reference

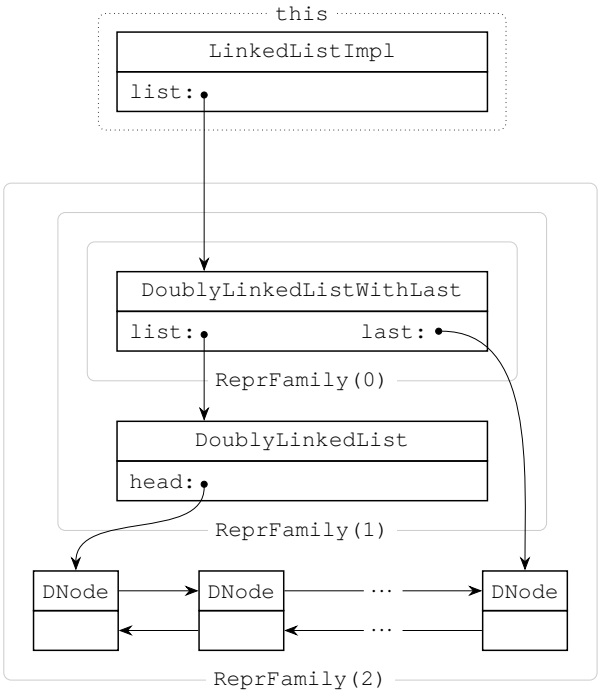


Figure 2.10: Stratified representation of `LinkedList`

to `DoublyLinkedListWithLast`, so we start level 0 with that reference. Then we add the reference to the `DoublyLinkedList` it is built on, building level 1. Finally, we add all the nodes of the `DoublyLinkedList` (contained in `list.list.Repr()` to build level 2.

In the discussion of the representation of `LinkedListImpl` we have omitted the field `iter`, even though it is in the representation. This ghost field stores all the iterators pointing to a node of the linked list. We will now give a short introduction to the iterators that we developed in previous work [1].

We save the iterators of the list in a ghost field to be able to specify the operations on them and to specify their invalidation policy, that is, which iterators are no longer valid after a list operation. In Figure 2.13 we show the definition of iterators for linked lists. It is similar to ADTs in the sense that it has a representation invariant (`Valid`) that ensures that the node is always pointing to the parent list to which it belongs, a model (`Index`) that specifies the position the iterator is pointing to, and a representation (the same as the representation of

```
class LinkedListImpl extends LinkedList {
  var list: DoublyLinkedListWithLast;
  ghost var iters: set<LinkedListIteratorImpl>;

  function ReprDepth(): nat
    ensures ReprDepth() > 0
  { 2 }

  function ReprFamily(n: nat): set<object>
    decreases n
    requires n ≤ ReprDepth()
    ensures n > 0 ⟹ ReprFamily(n) ≥ ReprFamily(n-1)
    reads this, if n = 0 then {} else ReprFamily(n-1)
  {
    if n = 0 then
      {list} + iters
    else if n = 1 then
      ReprFamily(0) + {list.list}
    else
      ReprFamily(1) + list.list.Repr()
  }
}
```

Figure 2.11: Code of stratified representation of `LinkedList`

the parent, `Parent().Repr()`). We also have a function method, `HasNext`, that determines whether the iterator can be advanced, and `Next`, the method that consumes the element of the list in position `Index()` and increases that number by one while keeping the rest of the iterators intact.

In Figure 2.12 we show some of the definitions of the `LinkedListImpl` class related to iterators. First, we define the user-facing function that returns the iterators of a list. That function is needed so that users of the ADT do not need to know that a field exists inside the class. We also add the definition of `Begin`, a method that returns a new iterator pointing to the first position, 0, while keeping the rest of the iterators and the model of the list intact. It is important that we specify that the model and the iterators remain the same because the method specification says that it modifies the representation. You may think that we do not need that `modifies`-clause because we are not really modifying the list, we are only constructing a new iterator, but we are in fact modifying ghost state: the `iters` field. Dafny cannot distinguish between ghost state and heap state in this regard, so we need to add the `modifies`-clause while we ensure that everything stays valid and the model does not change. This is why you will see in the rest of this work that we add a `modifies`-clause to methods that are apparently observers (they keep the model intact): we want to allow the implementors of

```
class LinkedListImpl extends LinkedList {
  function Iterators(): set<ListIterator>
  { iters }

  method Begin() returns (it: ListIterator)
    modifies this, Repr()
    requires Valid()
    ensures Valid()
    ensures Model() = old(Model())
    ensures it.Index() = 0
    ensures Iterators() = {it} + old(Iterators())
    ensures ∀ it | it ∈ old(Iterators()) ∧ old(it.Valid()) •
      ∧ it.Valid() ∧ old(it.Parent()) = it.Parent()
      ∧ old(it.Index()) = it.Index()
  {
    it := new LinkedListIteratorImpl(this);
    iters := {it} + iters;
  }

  method PopFront() returns (x: int)
    modifies this, Repr()
    ensures [x] + Model() = old(Model())
    ensures ∀ it | it ∈ Iterators() ∧ old(it.Valid()) ∧ old(it.Index()) ≠ 0 •
      it.Valid() ∧ it.Index() + 1 = old(it.Index())
}
```

Figure 2.12: `Iterators`, `Begin` and part of the specification of `PopFront`

these methods to modify internal ghost state.

In Figure 2.12 we also see an example method, `PopFront`, that specifies its invalidation policy: all the iterators that were **not** pointing to the first element of the list will still be valid and will have their index decreased by one. The behavior of the rest of the iterators is not specified, but since the user will not be able to prove that they are valid, they are invalid in practice. This specification is added to each of the mutator methods so that the user can verify which iterators they are using are still valid after operations on lists.

We will finish this section discussing the usual boilerplate code that Dafny needs to verify our methods but that we omit in this document. In Figure 2.14 we show the preconditions and postconditions present in all mutator methods. With them we ensure that the objects added to the representation are newly allocated (fresh). We also assert that all of the objects of the representation are allocated, that is, they are present in memory. This assertion is true for every object in Dafny, but we write it explicitly because Dafny sometimes cannot infer it by itself. We give more details on this matter in our previous work [1]. We will also omit the preconditions and postconditions that are not relevant to the

```
class LinkedListIteratorImpl extends ListIterator {
  ghost var parent: LinkedListImpl
  var node: DNode?

  predicate Valid()
    reads this, parent, parent.Repr()
  {
    ∧ parent.Valid()
    ∧ (node ≠ null ⟹ node ∈ parent.list.list.spine)
  }

  function Parent(): List
    reads this
  { parent }

  function Index(): nat
    reads this, parent, parent.Repr()
    requires Valid()
    requires Parent().Valid()
    ensures Index() ≤ |Parent().Model()|

  {
    if node ≠ null then
      parent.list.list.GetIndex(node)
    else
      |parent.list.list.spine|
  }

  function method HasNext(): bool
    reads this, Parent(), Parent().Repr()
    ensures HasNext() ⟺ Index() < |Parent().Model()|
  {
    parent.list.list.ModelRelationWithSpine();
    node ≠ null
  }

  method Next() returns (x: int)
    modifies this
    requires HasNext()
    ensures Index() = 1 + old(Index())
    ensures ∀ it | it ∈ Parent().Iterators() ∧ old(it.Valid()) •
      it.Valid() ∧ (it ≠ this ⟹ it.Index() = old(it.Index()))
  {
    x := node.data;
    node := node.next;
  }
}
```

Figure 2.13: Definition of iterators for linked lists

```
ensures fresh(Repr() - old(Repr()))

requires ∀ x | x ∈ Repr() • allocated(x)
ensures ∀ x | x ∈ Repr() • allocated(x)
```

Figure 2.14: Boilerplate code

subject being discussed and can be inferred from the context or be studied in the source code, for example that a method requires the ADT to be valid and ensures that it is valid after its execution.

2.3 Related work

The verification of abstract data types is well-studied in the automated verification literature. Different approaches have been developed to address the three building blocks needed to define an ADT, as outlined previously in this chapter: the model, the representation invariant and the representation.

Regarding the model, one of the approaches is based on *model fields* [3], that is, ghost fields that are automatically updated as the state is changed. This approach suffers from soundness issues, so Leino and Müller [5, 4] develop an alternative methodology called the *Boogie methodology*. Our *calculated model* is more similar to the approach of [3].

Some verification systems support class invariants that are automatically set as preconditions and postconditions of methods. Sometimes, however, we want to define auxiliary methods that do not fully comply with the invariant. In those systems the invariant can be lifted by defining a ghost attribute that determines whether the invariant must hold or not [8, 4]. In the system that we use we have to explicitly write as preconditions and postcondition that the invariant must hold, so defining a method that does not hold the invariant can be done by removing such pre/postconditions or by weakening them.

Verifying abstract data types sometimes is involved in the specification of how differente objects are related. Our *stratified representation* deals with ownership with a layered approach based on levels. This is similar to Müller's work [8], where objects are classified in contexts of different ownership level. To access an object from a different context there must be an ownership relation.

There are other works that have focused on the specification and verification of a container ADTs. One of the most notable are the container library EiffelBase2

developed by Polikarpova [10]. In her work, she implements and verifies maps and sets through hash tables, along other linear ADTs. In contrast, our work focuses on tree data structures to implement those ADTs.

There have been several verifications of red-black trees. The closest to our work is the one from Peña [9], where the author implements and verifies red-black trees as a functional data type in Dafny. By focusing on immutable types, all of the issues that come from framing can be avoided. That work could have been useful to specify how the trees in memory are being modified, but that is not the approach we have taken, since we specify directly the nodes and memory references of the trees without defining the red-black tree operations on immutable trees.

Chapter 3

ADT specification

In this chapter we will describe the specification of the ADTs that we have developed. The first ADT we will explore will be the set, in its two forms: ordered and unordered. We begin with sets because they have the key characteristics that we want to discuss throughout this chapter. Later, when we explore multisets and maps, we will present their differences to sets.

The code of this chapter can be found in the `src/tree/layer1/` directory of the associated code. The implementations at the end of this chapter are located in the `src/tree/layer3/` directory.

3.1 Sets

The difference between ordered and unordered sets lies in their iterators. While the iterators of ordered sets return the elements in ascending order, the iterators of unordered sets do not specify any order in particular. This is the reason why the `OrderedSet` trait extends the `UnorderedSet` trait. We will begin with the simplest of the two, unordered sets.

3.1.1 Unordered sets

The main trait of the unordered sets is the `UnorderedSet` trait. It is composed, as shown in the previous chapter, of a stratified representation, a validity predicate and a calculated model (of type `set<int>`[1]). In Figure 3.1 we omit those

[1]Note that our ADTs are not generic in the contained type because of the difficulties that Dafny has with generics. For the time being, our ADTs will contain `int`s, but our methodology

```
trait UnorderedSet {
  method Contains(x: int) returns (b: bool)
    modifies this, Repr()
    requires Valid()
    ensures Valid() ∧ Model() = old(Model())
    ensures b = (x ∈ Model())

  method Add(x: int)
    modifies this,Repr()
    requires Valid()
    ensures Valid() ∧ Model() = old(Model()) + {x}

  method Remove(x: int)
    modifies this,Repr()
    requires Valid()
    ensures Valid() ∧ Model() =  old(Model()) - {x}
}
```

Figure 3.1: `UnorderedSet` trait

definitions for brevity but show some basic methods, namely `Contains`, `Add` and `Remove`. These methods allow us to do basic operations on our sets. For example, in Figure 3.2 we implement a method that determines whether there are duplicate elements in a sequence by adding them to a set. Since we are dealing with abstract datatypes we cannot construct a new instance. Therefore, the method receives an empty unordered set as a parameter for internal use.

We can add, remove and check if a set contains an element, but we have not explored yet how to traverse all the elements of a set. To do so we will use iterators, as defined in Figure 3.3. We add to the main trait, `UnorderedSet`, a function that returns the set of iterators related to an instance: `Iterators()`. It is defined as a `function` and not as a `function method` because it is only used for verification purposes. The iterators of this set are of type `UnorderedSetIterator`. This type is a trait similar to the ones we developed in our previous work, but in this occasion we need an extra function: `Traversed`. This function defines the set of elements of the model that have been consumed by

applies to generic ADTs as well.

```
method HasDuplicates(p: seq<int>, s: UnorderedSet) returns (b: bool)
  modifies s, s.Repr()
  requires s.Valid() ∧ s.Empty()
  ensures s.Valid()
  ensures b = ∃ i, j | 0 ≤ i < j < |p| • p[i] = p[j]
{
  var n := 0;
  while n < |p|
    invariant s.Valid()
    invariant n ≤ |p|
    invariant ¬∃ i, j | 0 ≤ i < j < n • p[i] = p[j]
    invariant multiset(s.Model()) = multiset(p[..n])
  {
    var contained := s.Contains(p[n]);
    assert contained ⟺ p[n] ∈ multiset(s.Model());
    if contained {
      return true;
    }
    s.Add(p[n]);
    assert p[..n] + [p[n]] = p[..n+1];
    n := n + 1;
  }
  return false;
}
```

Figure 3.2: `HasDuplicates` method

the iterator. We also have, as usual, the following definitions:

- `Peek` returns the element currently pointed to by the iterator. The only property we know is that this element is in the model and not in the traversed elements, but it could be any of the possible elements.
- `HasNext` returns whether the iteration can proceed, i.e. whether there are any elements left to be traversed. It is equivalent to the `Traversed` set being a strict subset of the model.
- `Next` returns the currently pointed element and then advances the iterator. Its specification also has to ensure that all the other iterators remain valid and the element they are pointing to is the same.

Thanks to those definitions we can now store the elements of a set in a list, as shown in Figure 3.4. Note, however, that we cannot guarantee any order on the elements of the list since we are using an unordered set. We will change this once we explore ordered sets in the next section. Note, also, that we need to add a `modifies` clause to the method because we are altering ghost state (namely,

```
trait UnorderedSet {
  function Iterators(): set<UnorderedSetIterator>
    reads this, Repr()
    requires Valid()
    ensures ∀ it | it ∈ Iterators() • it ∈ Repr() ∧ it.Parent() = this
}

trait UnorderedSetIterator {
  function Parent(): UnorderedSet
    reads this

  predicate Valid()
    reads this, Parent(), Parent().Repr()

  function Traversed():set<int>
    reads this, Parent(), Parent().Repr()
    ensures Traversed() ≤ Parent().Model()

  function method Peek(): int
    reads this, Parent(), Parent().Repr()
    requires HasNext()
    ensures Peek() ∈ Parent().Model() ∧ Peek() ∉ Traversed()

  function method HasNext(): bool
    reads this, Parent(), Parent().Repr()
    ensures HasNext() ⟺ Traversed() < Parent().Model()

  method Next() returns (x: int)
    modifies this, Parent(), Parent().Repr()
    requires HasNext()
    ensures x = old(Peek())
    ensures Traversed() = {old(Peek())} + old(Traversed())

    ensures ∀ it | it ∈ Parent().Iterators() ∧ old(it.Valid()) •
      it.Valid() ∧ (it ≠ this ⟹ it.Traversed() = old(it.Traversed())
                              ∧ (it.HasNext() ⟹ it.Peek() = old(it.Peek())))
}
```

Figure 3.3: `UnorderedSetIterator` trait

```
method ToSeq(s: UnorderedSet) returns (p: seq<int>)
  modifies s, s.Repr()
  requires s.Valid()
  ensures s.Valid()
  ensures s.Model() = old(s.Model())
  ensures multiset(p) = multiset(s.Model())
{
  p := [];
  var it := s.First();
  while it.HasNext()
    decreases s.Model() - it.Traversed()
    invariant s.Valid() ∧ s.Model() = old(s.Model())
    invariant it.Valid() ∧ it.Parent() = s
    invariant multiset(p) = multiset(it.Traversed())
  {
    var x := it.Next();
    p := p + [x];
  }
}
```

Figure 3.4: `ToSeq` example

we are constructing a new iterator). Currently it is not possible to specify in Dafny that only ghost state can be modified.

Apart from traversing all the elements of our set, iterators also give us some more capabilities. In Figure 3.5 we define methods `Find`, `Insert` and `Erase`. These methods are similar to `Contains`, `Add` and `Remove` but they take or return iterators:

- `Find` returns an iterator pointing to the searched element or, if the element is not in the set, an iterator pointing past the end (that is, an iterator whose `HasNext` predicate returns `false`).

- `Insert` takes an iterator as a hint that can be used by the implementation for better performance[2] and returns an iterator pointing to the newly inserted element. The input iterator is only a hint, it can be used or not by the implementation, but there is no iterator that will cause an invalid state on the set. The worst that can happen if an "incorrect" iterator is given is a performance penalty. This behavior is borrowed from the C++ Standard [2].

[2]For example, a binary search tree could expect an iterator pointing to a node in which the new node can be placed maintaining the order of the tree.

```
method Find(x: int) returns (newt: UnorderedSetIterator)
  modifies this, Repr()
  requires Valid()
  ensures Valid() ∧ Model() = old(Model())
  ensures fresh(newt) ∧ newt.Valid() ∧ newt.Parent() = this
  ensures x ∈ old(Model()) ⟹ newt.HasNext() ∧ newt.Peek() = x
  ensures x ∉ old(Model()) ⟹ newt.Traversed() = Model()
  ensures Iterators() = {newt} + old(Iterators())

method Insert(mid: UnorderedSetIterator, x: int) returns (newt: UnorderedSetIterator)
  modifies this, Repr()
  requires Valid() ∧ mid.Valid() ∧ mid.Parent() = this ∧ mid ∈ Iterators()
  ensures Valid() ∧ Model() = old(Model()) + {x}
  ensures fresh(newt) ∧ Iterators() = {newt} + old(Iterators())
  ensures newt.Valid() ∧ newt.Parent() = this ∧ newt.HasNext() ∧ newt.Peek() = x

method Erase(mid: UnorderedSetIterator) returns (next: UnorderedSetIterator)
  modifies this, Repr()
  requires Valid() ∧ mid.Valid()
  requires mid.Parent() = this ∧ mid.HasNext() ∧ mid ∈ Iterators()
  ensures Valid() ∧ Model() = old(Model()) - {old(mid.Peek())}
  ensures fresh(next) ∧ Iterators() = {next} + old(Iterators())
  ensures next.Valid() ∧ next.Parent() = this
  ensures next.Traversed() = old(mid.Traversed())
```

Figure 3.5: `Find`, `Insert` and `Erase` methods

- `Erase` takes an iterator, removes the element it is pointing to and returns an iterator pointing to the next element.

We do not specify an invalidation policy for the iterators as we did for linked lists in our previous work. An invalidation policy would need to be dependent on the data structure that implements the ADT. For trees, that policy would not be trivial, so we leave it for future work.

3.1.2 Ordered sets

Unordered sets are useful in many cases but we sometimes need to traverse the elements in order. There are performant implementations of ordered sets, like the red-black trees that we will explore in Chapter 5, so it is often a good choice in terms of performance too. In this section we present our ordered set interface and specification.

The `OrderedSet` trait inherits from `UnorderedSet`. The main refinement it does over unordered sets is on the `Traversed` function. In Figure 3.6 we show the postconditions added to the ordered set iterators. In particular,

```
trait OrderedSetIterator extends UnorderedSetIterator{
  function Traversed():set<int>
    ensures ∀ x,y | x ∈ Traversed() ∧ y ∈ Parent().Model()-Traversed() • x < y

  function method Peek(): int
    ensures Peek() = elemth(Parent().Model(), |Traversed()|)

  function method Index(): int
    ensures HasNext() ⟹ Index() = |Traversed()|
    ensures ¬HasNext() ⟹ Index() = |Parent().Model()|

  function method HasPrev(): bool
    ...

  method Prev() returns (x: int)
    ...
}
```

Figure 3.6: Ordered set iterators

the new postcondition of `Traversed` ensures that each traversed element is smaller than the rest of the model. `Peek` is also updated to ensure that it returns the correct element from the model, that is, if n is the number of elements in `Traversed`, it returns the n-th element of the ordered `Model`. This behavior is specified with the help of the `elemth` function, omitted in the figure. We also add a new function method that returns the number of traversed elements.

Another feature of the iterators of ordered sets is going back in the traversal. `OrderedSetIterator` has a `Prev` method that traverses the elements in reverse order. Its precondition is `HasPrev`, similar to `HasNext`. We also add a new constructor method for iterators: `Last`. This method returns an iterator pointing to the last element, allowing the user to traverse the set from the end to the beginning.

Lastly, `Find`, `Insert` and `Remove` have new postconditions that specify the position of the returned iterators. For example, in Figure 3.7 we see the postcondition added to `Find` that says that the traversed elements of the new iterator are all the elements smaller than the found element.

3.2 Maps and multisets

Maps and multisets are related to the set ADT since they usually share the underlying data structure for their implementation and also a good part of the specification. In this section we will explore the differences they have with sets.

```
ensures x ∈ Model() ⟹
  ∧ newt.HasNext()
  ∧ newt.Traversed() = smaller(Model(), x)
  ∧ newt.Peek() = x
ensures x ∉ Model() ⟹
  newt.Traversed() = Model()
```

Figure 3.7: `Find` new postcondition

Both maps and multisets come in ordered and unordered variants, giving four traits in total: `UnorderedMap`, `OrderedMap`, `UnorderedMultiset` and `OrderedMultiset`. The most obvious difference is in the model: maps have a model of type `map<int, int>` and multisets of type `multiset<int>`.

Apart from the methods that they share with sets, maps also have the `At` method which, given a key `k`, returns the value associated with it `Model()[k]`. Similarly, multisets have the `Count` method that returns the number of occurrances of an element (the multiplicity).

Regarding iterators, map iterators give a pair of integers (the key and the value) each time we call `Next`, traversing one by one all the pairs of the map. Apart from that, they behave the same as set iterators. Multisets, however, behave differently. `Next` returns only one integer each time it is called, but it can return the same integer several times. This depends on the multiplicity of the element it is pointing to. For example, suppose that we transform the `ToSeq` method we explored in the last section in Figure 3.4 to accept multisets. For a multiset `multiset{1,1,2,2,2}` the returned sequence would be `[1,1,2,2,2]` if the multiset was ordered (or any permutation of it if it was unordered). This means that the loop body has been executed several times for each element.

3.3 Implementations

3.3.1 Inefficient list-based implementations

Currently we have implemented unordered sets with arrays and with linked lists. To do so, we used the linear ADTs we developed in our previous work [1]. Thanks to the stratified representation we can use ADTs to implement other ADTs without breaking encapsulation. In Figure 3.8 we show the stratification of the linked list implementation. We need to include all of the representation

```
class UnorderedSetImplLinkedList extends UnorderedSetLinkedList {
  var elems: LinkedListImpl
  ghost var iters: set<UnorderedSetIteratorImplLinkedList>

  function ReprDepth(): nat
  { elems.ReprDepth() + 2 }

  function ReprFamily(n: nat): set<object>
  { if n = 0 then {elems} + iters
    else if n = 1 then ReprFamily(0) + (set it | it ∈ iters • it.iter)
    else ReprFamily(1) + elems.ReprFamily(n-2)
  }
}
```

Figure 3.8: Stratification of `UnorderedSetImplLinkedList`

```
method Add(x: int)
{
  var b := Contains(x);
  if ¬b { elems.PushBack(x); }
}
```

Figure 3.9: `Add` method of `UnorderedSetImplLinkedList`

of the `LinkedList`, since the memory footprint of that ADT is included in the footprint of the set. The implementation with arrays is similar.

These implementations are simple but inefficient, since their only objective is to test the specification and the use of other ADTs. For example, the `Add` method of `UnorderedSetImplLinkedList` consists of only two lines of code (and none of them are verification related), as shown in Figure 3.9.

Iterators are implemented by using the underlying iterators of linked lists, as shown in Figure 3.10. The representation invariant of iterators makes sure that no list iterator is shared between set iterators. The rest of the implementation has to implement the correspondance between the model of the list and the external specification of sets.

3.3.2 Implementations with binary search trees

Binary search trees have long been used to implement map-like data structures [11]. In this work our primary objective was to verify the implementation of such trees, more concretely red-black binary search trees. More details will be given on the intricacies of their verification in chapters 4 and 5, but we would also like to add that their encapsulation into a map ADT is being worked on. Currently,

```
class UnorderedSetIteratorImplLinkedList extends UnorderedSetIterator {
  var iter: LinkedListIteratorImpl;
  ghost var parent: UnorderedSetImplLinkedList;

  predicate Valid()
    reads this, Parent(), Parent().Repr()
  {
    ∧ iter ∈ Parent().Repr()
    ∧ iter.Parent() = parent.elems
    ∧ parent.Valid()
    ∧ iter.Valid()
    ∧ iter ∈ parent.elems.Iterators()
    ∧ (∀ it | it ∈ parent.iters ∧ it ≠ this •
        (it as UnorderedSetIteratorImplLinkedList).iter ≠ iter)
  }
}
```

Figure 3.10: `Add` method of `UnorderedSetImplLinkedList`

`OrderedMapImpl` implements the `Contains`, `Add` and `Remove` methods by using binary search trees. An implementation of iterators is also being worked on using a stack, as explained in Chapter 6.

3.4 Examples

We close this chapter by giving some examples of use of iterators for ordered and unordered sets. Notably, we want to show the use of multiple iterators at once.

In Figure 3.11 we show the implementation of a method that returns whether an ordered set is a subset of another one. To do so, it has to receive two sets that have disjoint representations and are valid. The implementation is based on a `while`-loop that traverses the two sets in order until a discrepancy is detected, in that case the method returns `false`. Some of the invariants of the loop have been omitted because they are the same as the preconditions, but others are important to note. Namely, we are guaranteeing that during the execution of the loop, the traversed elements of the first iterator are a subset of the traversed elements of the second. After the `while`-loop, we give a small manual proof on why we can assure that the first set is a subset of the second by checking `!it1.HasNext()` after the execution of the loop.

In Figure 3.12 we show a method that does the same as the previous one but using unordered sets, omitting the boilerplate that has already been shown. This time we cannot rely on the order of the elements consumed by the iterator, so we need to use `Find` method. If after the execution of the method none of the

elements of the first set have not been found in the second set, we return `true`. We return `false` otherwise.

```
method Contained(s1: OrderedSet, s2: OrderedSet) returns (b: bool)
  modifies s1, s1.Repr(), s2, s2.Repr()

  requires ({s1} + s1.Repr()) ∩ ({s2}+s2.Repr()) = {}
  requires s1.Valid() ∧ s2.Valid()

  ensures ({s1} + s1.Repr()) ∩ ({s2}+s2.Repr()) = {}
  ensures s1.Valid() ∧ s2.Valid()
  ensures s1.Model() = old(s1.Model()) ∧ s2.Model() = old(s2.Model())
  ensures s1.Iterators() ≥ old(s1.Iterators())
  ensures s2.Iterators() ≥ old(s2.Iterators())

  ensures b = (s1.Model() ≤ s2.Model())
{
  var it1 := s1.First();
  var it2 := s2.First();

  while it1.HasNext() ∧ it2.HasNext()
    decreases s2.Model() - it2.Traversed()
    invariant it1.Valid() ∧ it2.Valid()
    invariant it1 ∈ s1.Iterators() ∧ it2 ∈ s2.Iterators()
    invariant it1.Parent() = s1 ∧ it2.Parent() = s2

    invariant it1.Traversed() ≤ it2.Traversed()
    invariant it1.HasNext() ⟹
      ∧ (∀ x | x ∈ it2.Traversed() • it1.Peek() > x)
      ∧ it1.Peek() ∉ it2.Traversed()
  {
    if it1.Peek() = it2.Peek() {
      var _ := it1.Next();
      var _ := it2.Next();
    } else if it1.Peek() < it2.Peek() {
      assert it1.Peek() ∉ it2.Traversed();
      assert ∀ z | z ∈ s2.Model() - it2.Traversed() - {it2.Peek()} •
        it1.Peek() < it2.Peek() < z;
      assert it1.Peek() ∉ s2.Model();
      return false;
    } else {
      assert it1.Peek() > it2.Peek();
      var _ := it2.Next();
    }
  }

  if it1.HasNext() {
    assert it1.Peek() ∉ it2.Traversed();
    assert it2.Traversed() = s2.Model() by {
      assert ¬it2.HasNext();
    }
    assert it1.Peek() ∉ s2.Model();
    assert ¬(s1.Model() ≤ s2.Model());
  } else {
    assert it1.Traversed() = s1.Model() ≤ it2.Traversed() ≤ s2.Model();
  }
  return ¬it1.HasNext();
}
```

Figure 3.11: `Contained` example for ordered sets

```
method Contained(s1: UnorderedSet, s2: UnorderedSet) returns (b: bool)
  modifies s1, s1.Repr(), s2, s2.Repr()
  ensures b = (s1.Model() ≤ s2.Model())
{
  var it := s1.First();
  b := true;
  while b ∧ it.HasNext()
    decreases s1.Model() - it.Traversed()
    invariant b = (it.Traversed() ≤ s2.Model())
  {
    var x := it.Next();
    var f := s2.Find(x);
    b := f.HasNext();
  }
}
```

Figure 3.12: `Contained` example for unordered sets

Chapter 4

Binary search trees

Binary trees are a basic data structure used to store information in a hierarchical way. In this work we focus on self-balancing binary search trees, but before we can get to self-balancing trees we need a good foundation on binary search trees. In this chapter we will explore how to implement and verify these trees in Dafny.

The code of this chapter can be found in the `src/tree/Tree.dfy` and `src/tree/SearchTree.dfy` files of the associated code.

4.1 Binary search trees in Dafny

Before we define binary search trees, we define binary trees in Figure 4.1. Each node stores a key and a value, together with its left and right children. The tree only stores a reference to the root. It also saves in a ghost field the *skeleton* of the tree, a concept similar to the spine of linked lists that we explored in our previous work [1]. The skeleton is an immutable tree of nodes, as defined in Figure 4.2, that matches the structure of the real tree in memory. In Figure 4.3 we show a tree with five nodes as it is stored in memory, alongside its corresponding skeleton, represented with dotted lines.

The representation invariant, that is, the `Valid` predicate, ensures that the skeleton matches the real nodes in memory. It also ensures that there are no loops or sharing in the tree. In Figure 4.4 we show its definition[1]. For empty

[1]Note that in that figure the only predicate shown is `ValidRec`. As a convention, we add the `Rec` suffix to the auxiliary definitions of methods. Since they are the more interesting definitions, we will omit the main methods without the suffix that only call the recursive definition with the correct arguments (`root` and/or `skeleton`).

```
type K = int
type V = int

class TNode {
  var key: K;
  var value: V;
  var left: TNode?;
  var right: TNode?;
}
```

```
class Tree {
  var root: TNode?;
  ghost var skeleton: tree<TNode>;

  function Repr(): set<object>
    reads this
  {
    elems(skeleton)
  }
}
```

Figure 4.1: Tree and tree node classes

```
datatype tree<A> = Empty | Node(left: tree<A>, data: A, right: tree<A>)

function method elems<A>(t: tree<A>): set<A>
{
  match t {
    case Empty ⇒ {}
    case Node(l, x, r) ⇒ elems(l) + {x} + elems(r)
  }
}
```

Figure 4.2: Immutable trees

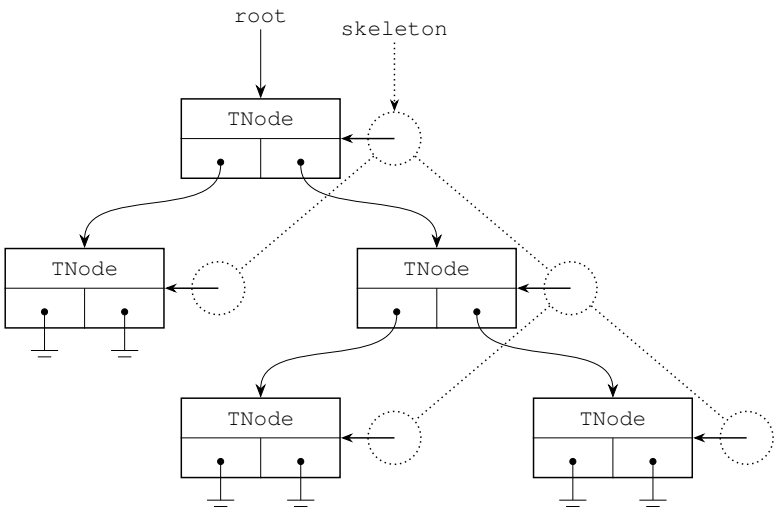


Figure 4.3: Skeleton and nodes

```
static predicate ValidRec(node: TNode?, sk: tree<TNode>)
  reads set x | x ∈ elems(sk) • x`left
  reads set x | x ∈ elems(sk) • x`right
{
  match sk {
    case Empty ⇒ node = null
    case Node(l, x, r) ⇒
      ∧ x = node
      ∧ x ∉ elems(l) ∧ x ∉ elems(r)   // No loops
      ∧ elems(l) ∩ elems(r) = {}      // No sharing
      ∧ ValidRec(node.left, l)
      ∧ ValidRec(node.right, r)
  }
}
```

Figure 4.4: Validity of trees

trees we require that the root is null. For nodes of the tree, we require that the root of the skeleton and of the tree are the same, that the root is not present in any of its children (no loops) and that the children do not share any of their nodes (no sharing).

With the definition of the model in Figure 4.5 we complete our definition of binary trees. The model collects all the key-value pairs in the nodes of the tree and stores them in a map. We chose maps as the model for trees since they are meant to implement sets, multisets and maps, not a general interface for trees, although that is an alternative. This definition is *opaque*, that is, its body is not known to Dafny for verification until the `reveal ModelRec();` statement is issued. The decision to make it opaque was made after discovering that, by separating the proofs about the model from the rest of the proofs, the verification time is reduced significantly[2]. By making the `ModelRec` function opaque we can introduce its definition only in the fragments related to it. This way the rest of the proofs can be verified faster since, as we understand it, Dafny can do a more directed verification by hiding the unrelated assertions.

Now that we have binary trees, we define binary search trees using a predicate in Figure 4.6. Adding a new definition for search trees allows us to verify the validity and the order of the keys separately. `SearchTreeRec` holds for trees such that, for each of its subtrees, the nodes down its left branch have smaller keys than the root, and bigger keys down its right branch.

[2]We believe that this is due to the `map` automatic reasoning of Dafny not being as performant as the automation for sets, since we have not had this problem with the other predicates of this work.

```
static function {:opaque} ModelRec(sk: tree<TNode>): map<K, V>
  reads set x | x ∈ elems(sk) • x`key
  reads set x | x ∈ elems(sk) • x`value
{
  match sk {
    case Empty() ⇒ map[]
    case Node(l, n, r) ⇒ ModelRec(l) + ModelRec(r) + map[n.key := n.value]
  }
}
```

Figure 4.5: Model of trees

```
static predicate SearchTreeRec(sk: tree<TNode>)
  reads set x | x ∈ elems(sk) • x`key
{
  match sk {
    case Empty() ⇒ true
    case Node(l, n, r) ⇒
      ∧ (∀ m | m ∈ elems(l) • m.key < n.key)
      ∧ (∀ m | m ∈ elems(r) • n.key < m.key)
      ∧ SearchTreeRec(l)
      ∧ SearchTreeRec(r)
  }
}
```

Figure 4.6: Definition of binary search tree

```
static lemma ModelLemmas(node: TNode?, sk: tree<TNode>)
  requires ValidRec(node, sk)
  requires SearchTreeRec(sk)
  ensures ModelRec(sk).Keys = TreeKeys(sk)
  ensures ModelRec(sk) = map k | k ∈ TreeKeys(sk) • FindNode(k, node, sk).value
  ensures ∀ n | n ∈ elems(sk) • n.key ∈ ModelRec(sk) ∧
n.value = ModelRec(sk)[n.key]
  ensures sk.Node? ⟹
    ∧ sk.data.key ∉ ModelRec(sk.left)
    ∧ sk.data.key ∉ ModelRec(sk.right)
    ∧ (∀ k | k ∈ ModelRec(sk.left) • k ∉ ModelRec(sk.right))
    ∧ (∀ k | k ∈ ModelRec(sk.right) • k ∉ ModelRec(sk.left))
    ∧ (∀ k | sk.data.key ≤ k • k ∉ ModelRec(sk.left))
    ∧ (∀ k | k ≤ sk.data.key • k ∉ ModelRec(sk.right))
{
  reveal ModelRec();
  match sk {
    case Empty() ⇒ {}
    case Node(l, n, r) ⇒ {
      ModelLemmas(node.left, sk.left);
      ModelLemmas(node.right, sk.right);
    }
  }
}
```

Figure 4.7: Definition of `ModelLemmas`

Before we continue with the methods that operate on binary search trees, we need to discuss the `ModelLemmas` definition in Figure 4.7. These lemmas relate the model with the skeleton, for example, proving that every node of the tree has its key in the model, that its value is the corresponding from the model, that the key of the root is not in the model of its children, etc. The methodology we follow to verify the methods on binary search trees is based on verifying first properties about the nodes of the tree and then using that knowledge to prove the postconditions about the model. The reason we do not prove directly the properties about the model is that the properties on `Model` are harder to verify. `ModelLemmas` helps with the verification of such properties.

4.2 Search

We begin defining methods that operate on binary search trees with the `FindRec` method in Figure 4.8. This method searches a node with the given key. If the node is not found, it returns null. Otherwise, it returns the correct node. This is specified with the two last two postconditions. The first one ensures that the

```
static method FindRec(node: TNode?, ghost sk: tree<TNode>, k: K)
  returns (found: TNode?)

  requires ValidRec(node, sk)
  requires SearchTreeRec(sk)

  ensures ValidRec(node, sk)
  ensures SearchTreeRec(sk)
  ensures found = null ⟺ k ∉ ModelRec(sk)
  ensures found ≠ null ⟹ found.key = k
                       ∧ found.value = ModelRec(sk)[k]
                       ∧ found ∈ elems(sk)
{
  if node = null {
    found := null;
  } else {
    assert elems(sk) = elems(sk.left) + {sk.data} + elems(sk.right);
    if k = node.key       { found := node; }
    else if node.key < k { found := FindRec(node.right, sk.right, k); }
    else                  { found := FindRec(node.left, sk.left, k); }
  }
  assert ... by {
    reveal ModelRec();
    ModelLemmas(node, sk);
  }
}
```

Figure 4.8: `FindRec` method

returned node is null if and only if the key is present in the model. The second one ensures that if the node is found, its key and value are correct and that it is in the skeleton. This is implemented with a binary search along the tree and proved by invoking `ModelLemmas`.

The `FindRec` method is used to implement several user-facing methods shown in Figure 4.9. They implement simple operations on trees and are verified trivially.

4.3 Insertion

Next we will discuss the `InsertRec` method of Figure 4.10. This method inserts a new node in the tree while preserving the order of the keys. If there is already a node with the input key, it returns it; otherwise it returns the newly inserted node. The implementation of this method performs binary search until a node with the input key is found or, if none exists, it constructs a new node in the correct place. The verification adds some concepts that will be explored when

```
method Find(k: K) returns (found: TNode?)
  requires Valid()
  requires SearchTree()
  ensures found = null ⟺ k ∉ Model()
  ensures found ≠ null ⟹
    ∧ found.key = k
    ∧ found.value = Model()[k]
    ∧ found ∈ elems(tree.skeleton)
{
  found := FindRec(tree.root, tree.skeleton, k);
}
```

```
method Get(k: K) returns (v: V)
  requires Valid()
  requires SearchTree()
  requires k ∈ Model()
  ensures Model()[k] = v
{
  var found := Find(k);
  return found.value;
}

method Search(k: K) returns (b: bool)
  requires Valid()
  requires SearchTree()
  ensures b = (k ∈ Model())
{
  var found := Find(k);
  return found ≠ null;
}
```

Figure 4.9: Methods derived from `FindRec`

we discuss deletion.

4.4 Deletion

The last method we want to discuss is the `RemoveRec` method, but first we need to define `RemoveMinRec`. This method, shown in Figure 4.11, removes and returns the node with the smallest key of a non-empty tree. It is implemented by descending through the left branch until it is empty. This method mutates the tree that it receives, so we need to specify and verify that mutation. Concretely, the new model is the previous method without the removed key, formalized in the postcondition marked with a ★ symbol. Dafny cannot prove this automatically so we need to prove it manually. We only show one of the proofs in Figure 4.12 since they follow the same strategy; namely, they prove a series of equalities that depend on properties of maps.

Finally, we can implement the `RemoveRec` method of Figure 4.13. The implementation is as we would expect: binary search until the node is found, once it is found remove the minimum node of the right branch and replace the node we want to remove with the minimum node. The verification of this method has more cases but the proof techniques have already been presented, so we omit the proofs in the figure and add annotations to inform of the number of lines omitted.

```
static method InsertRec(node: TNode?, ghost sk: tree<TNode>, k: K, v: V)
    returns (newNode: TNode, ghost newSk: tree<TNode>, ghost insertedNode: TNode)
  modifies elems(sk)
  requires ValidRec(node, sk) ∧ SearchTreeRec(sk)
  ensures ValidRec(newNode, newSk) ∧ SearchTreeRec(newSk)
  ensures ModelRec(newSk) = old(ModelRec(sk))[k := v]

  ensures elems(newSk) = elems(sk) + {insertedNode}
  ensures insertedNode.key = k ∧ insertedNode.value = v
  ensures ∀ n | n ∈ elems(sk) ∧ old(n.key) ≠ k •
    n.key = old(n.key) ∧ n.value = old(n.value)
{
  if node = null {
    newNode := new TNode(null, k, v, null);
    newSk := Node(Empty, newNode, Empty);
    insertedNode := newNode;
  } else {
    newNode := node;
    if k = node.key {
      node.value := v;
      newSk := sk;
      insertedNode := node;
    } else if node.key < k {
      ghost var newSkRight;
      node.right, newSkRight, insertedNode := InsertRec(node.right, sk.right, k, v);
      newSk := Node(sk.left, node, newSkRight);
    } else {
      ghost var newSkLeft;
      node.left, newSkLeft, insertedNode := InsertRec(node.left, sk.left, k, v);
      newSk := Node(newSkLeft, node, sk.right);
      assert ModelRec(newSk) = old(ModelRec(sk))[k := v] by {
        /* 10 omitted lines of proof */
      }
    }
  }
}
```

Figure 4.10: `InsertRec` method

```
static method RemoveMinRec(node: TNode, ghost sk: tree<TNode>)
    returns (newNode: TNode?, ghost newSk: tree<TNode>, removedNode: TNode)
  modifies elems(sk)
  requires ValidRec(node, sk)
  requires SearchTreeRec(sk)
  ensures ValidRec(newNode, newSk)
  ensures SearchTreeRec(newSk)

  ensures removedNode.key ∈ old(ModelRec(sk))
  ensures old(ModelRec(sk))[removedNode.key] = removedNode.value
  ensures ModelRec(newSk) = old(ModelRec(sk)) - {removedNode.key}   // ★

  ensures removedNode ∈ elems(sk) ∧ removedNode ∉ elems(newSk)
  ensures elems(newSk) = elems(sk) - {removedNode}
  ensures ∀ n | n ∈ elems(sk) • n.key = old(n.key) ∧ n.value = old(n.value)
  ensures ∀ n | n ∈ elems(newSk) • removedNode.key < n.key
  ensures fresh(elems(newSk)-old(elems(sk)))
{
  if node.left = null {
    newNode := node.right;
    newSk := sk.right;
    removedNode := node;
    assert ModelRec(newSk) = old(ModelRec(sk)) - {removedNode.key} ∧ ... by {
      // See figure 4.12
    }
  } else {
    newNode := node;
    newSk := sk;
    ghost var newSkLeft;
    newNode.left, newSkLeft, removedNode := RemoveMinRec(newNode.left, newSk.left);
    newSk := Node(newSkLeft, newNode, newSk.right);
    assert ... by { ... }
  }
}
```

Figure 4.11: `RemoveMinRec` method

```
reveal ModelRec();
var k := removedNode.key; var v := removedNode.value;
calc = {
  ModelRec(newSk);
  old(ModelRec(sk.right));
  old(ModelRec(sk.right)) + (map[k := v] - {k});
  { assert k ∉ old(ModelRec(sk.right)) by { ... } }
  (old(ModelRec(sk.right)) + map[k := v]) - {k};
  { assert old(ModelRec(sk.left)) = map[]; }
  (old(ModelRec(sk.left)) + old(ModelRec(sk.right)) + map[k := v]) - {k};
  old(ModelRec(sk)) - {k};
}
```

Figure 4.12: Proof of the mutation of the model in `RemoveMinRec`

```
static method RemoveRec(node: TNode?, ghost sk: tree<TNode>, k: K)
    returns (newNode: TNode?, ghost newSk: tree<TNode>, ghost removedNode: TNode?)
  modifies elems(sk)
  requires ValidRec(node, sk) ∧ SearchTreeRec(sk)
  ensures ValidRec(newNode, newSk) ∧ SearchTreeRec(newSk)

  ensures ∀ n | n ∈ elems(sk) • n.key = old(n.key) ∧ n.value = old(n.value)
  ensures elems(newSk) = elems(sk) - {removedNode}
  ensures removedNode ≠ null ⟹
    ∧ removedNode ∈ elems(sk)
    ∧ removedNode ∉ elems(newSk)
    ∧ removedNode.key = k
    ∧ removedNode.key ∈ old(ModelRec(sk))
    ∧ old(ModelRec(sk))[removedNode.key] = removedNode.value

  ensures removedNode = null ⟺ k ∉ old(ModelRec(sk))
  ensures ModelRec(newSk) = old(ModelRec(sk)) - {k}
{
  newNode := node;
  newSk := sk;
  if newNode = null {
    removedNode := null;
    assert ... by { ... }
  } else {
    if node.key > k {
      assert k ∉ ModelRec(sk.right) by { ... }
      ghost var newSkLeft;
      newNode.left, newSkLeft, removedNode :=
        RemoveRec(newNode.left, newSk.left, k);
      newSk := Node(newSkLeft, newNode, newSk.right);
      assert ... by { /* 20 omitted lines of proof */ }
    } else {
      if k = newNode.key ∧ newNode.right = null {
        assert node.key ∉ ModelRec(sk.left) by { ... }
        removedNode := newNode;
        newNode := newNode.left;
        newSk := newSk.left;
        assert ... by { /* 11 omitted lines of proof */ }
        return;
      }

      if k ≠ newNode.key {
        ghost var newSkRight;
        newNode.right, newSkRight, removedNode :=
          RemoveRec(newNode.right, newSk.right, k);
        newSk := Node(newSk.left, newNode, newSkRight);
        assert ... by { /* 15 omitted lines of code */ }
      } else {
        assert k ∉ ModelRec(sk.left) ∧ k ∉ ModelRec(sk.right) by { ... }
        removedNode := newNode;
        ghost var newSkRight;
        var minNode;
        newNode.right, newSkRight, minNode :=
          RemoveMinRec(newNode.right, newSk.right);
        minNode.left := newNode.left;
        minNode.right := newNode.right;
        minNode.color := newNode.color;
        newNode := minNode;
        newSk := Node(newSk.left, newNode, newSkRight);
        assert ... by { /* 15 omitted lines of code */ }
      }
    }
  }
}
```

Figure 4.13: `RemoveRec` method

Chapter 5

Red-black trees

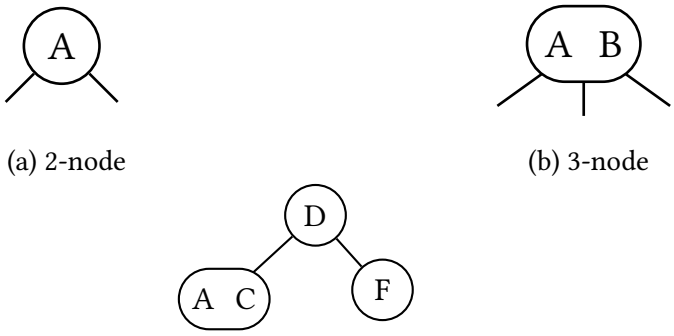
In this chapter we are going to study the implementation and verification of a specific instance of self-balancing trees: left-leaning red-black trees [11]. Self-balancing trees represent an advantage over the binary search trees presented in the last chapter, since they guarantee logarithmic time for all the operations that we show in this work. That performance advantage, however, comes with implementation complexity. In this chapter we will verify that the implementation of self-balancing trees is correct while the performance boosts are retained.

To achieve the goals set we will need to explore how red-black trees work in theory, without code. After that we will see the formal definition we give to red-black trees in Dafny, the auxiliary methods that we need and finally the implementation and verification of insertion and deletion in these trees.

The code of this chapter can be found in the `src/tree/Tree.dfy` and `src/tree/RedBlackTree.dfy` files of the associated code.

5.1 The theory of left-leaning red-black trees

Red-black trees come directly from ordered 2-3 trees, the later of which are not binary. Instead, they have nodes with two or three children, as shown in Figure 5.1. These trees can be manipulated to remain perfectly balanced (i.e. the length of every path from the root to a leaf is the same) after insertion and deletion, but they come with a drawback: they are complicated to implement due to the many cases that we have to consider. To solve that problem left-leaning red-black trees were introduced. These trees are binary search trees with red and black links, and have a one to one mapping to 2-3 trees, as shown in Figure 5.2, but their



(a) 2-node

(b) 3-node

(c) Example tree

Figure 5.1: 2-3 trees.

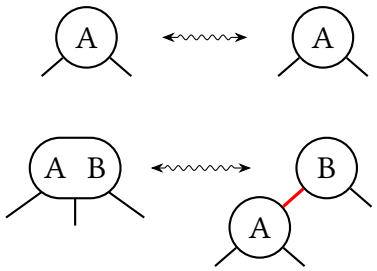


Figure 5.2: 1-to-1 mapping between 2-3 trees and red-black trees.

implementation is easier since we can reuse much of the implementation of the binary trees that we discussed in previous sections. Given that 2-3 nodes and red-black trees are isomorphic we will sometimes write 2-node to mean a node without red children and a 3-node to mean a node with its left child red.

Red-black trees, as already seen in Figure 5.2, have two types of links: red and black. The black links work as the default links, whereas red links bind together two nodes to form a 3-node. In a more formal way, red-black trees must comply with the following rules, as described in [11]:

- Red links lean left.
- No node has two red links connected to it.
- The tree has perfect black balance: every path from the root to a null link has the same number of black links.

```
type K = int
type V = int

datatype Color = Red | Black

class TNode {
  var key: K;
  var value: V;
  var left: TNode?;
  var right: TNode?;
  var color: Color;
}

function method isRed(node: TNode?): bool
{ node ≠ null ∧ node.color.Red? }

function method isBlack(node: TNode?): bool
{ node ≠ null ∧ node.color.Black? }
```

Figure 5.3: `TNode` definition for red-black trees

5.2 Red-black trees in Dafny

In order to represent the color of the link between nodes we add a new field to each node that signifies the color of the link that connects that node to its parent. The final definition is presented in Figure 5.3[1]. In that figure we also add the definitions of `isRed` and `isBlack`. They allow us to succintly express that one node is not null and is of a specific color. If we want to express that, for example, a `node` is either null or black, we write `!isRed(node)`.

In Figure 5.4 we present the formalization of the definition of red-black trees shown in the previous section. We define it as a predicate so that we can verify it independently from other properties, like being a binary search tree. Defining red-black trees this way allows us to reuse all the code the we discussed in the previous chapter, since red-black trees are also binary search trees. For example, `Find`, `Search` and `Get` remain exactly the same, altough we will need to redefine insertion and deletion so that they maintain the invariant of red-black trees.

[1]Note that in our code, binary search trees also use this definition ignoring the `color` field. In a commercial application this could be a bad choice, but our work is more experimental and we find it acceptable and useful.

```
static predicate RedBlackTreeRec(sk: tree<TNode>)
  reads set x | x ∈ elems(sk) • x`key
  reads set x | x ∈ elems(sk) • x`color
{
  match sk {
    case Empty() ⇒ true
    case Node(l, n, r) ⇒
      // Red links lean left:
      ∧ (r.Node? ⟹ r.data.color.Black?)
      // No node has two red links connected to it:
      ∧ (l.Node? ∧ l.data.color.Red? ∧ l.left.Node? ⟹ l.left.data.color.Black?)
      // Perfect black balance:
      ∧ BlackHeight(l) = BlackHeight(r)

      ∧ RedBlackTreeRec(l)
      ∧ RedBlackTreeRec(r)
  }
}
```

Figure 5.4: `RedBlackTreeRec` predicate

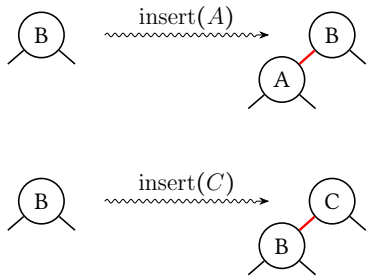
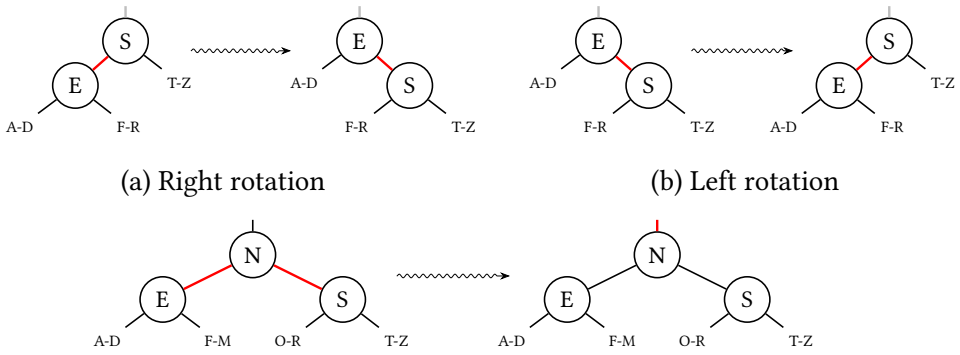


Figure 5.5: Insertion on a 2-node

5.3 Insertion

Now that we have a definition of red-black trees we can start to study how to perform operations on them. We will begin with insertion. To insert a node to a tree composed of a 2-node is easy, as shown in Figure 5.5. If the key is lower than that of the 2-node, insert it on the left; if the key is higher, the new key will be the parent of the old key. In both cases the resulting tree is a 3-node and the perfect black balance is preserved. When we try to insert a node into a 3-node, however, preserving perfect black balance becomes difficult. To do so we first need to study three auxiliary operations on red-black trees. In Figure 5.6 we represent rotation on the left, on the right and flipping colors, where a gray link



(a) Right rotation

(b) Left rotation

(c) Flip colors

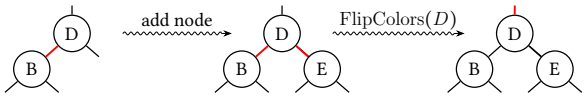
Figure 5.6: Auxiliary operations

means that the color stays the same. All of these operations preserve the perfect black balance, although they can produce invalid red-black trees.

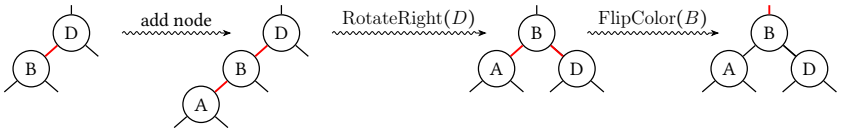
Now we can explore the three cases for insertion in a 3-node in Figure 5.7. In all of them we insert the new key with a red link connected to its parent (preserving perfect black balance) and perform the operations needed to turn it into a valid red-black tree. If the key being inserted is larger than the keys from the 3-node, we add it to the right and then perform a flip-color. The perfect black balance is preserved at the expense of sending a red link upwards. When the key is smaller than the keys of the 3-node we need to perform a right rotation and to flip colors. Finally, when the key is between the other keys we perform a left rotation on the left subtree and then we proceed as in the previous case.

The complete algorithm for the insertion recurses down the tree on the search of a leaf, 2-node or 3-node, in which it can insert a node with the new key. Once it finds it, it performs the operations described and then goes back in the call stack, going up in the tree. These operations will be repeated along the ascent of the tree, probably sending a red link upwards, to produce a valid red-black tree while maintaining perfect black balance, just as we do on the leaves. At the top, if the link to the root is marked as red, we will simply turn it black again, increasing the black height by one but preserving perfect black balance.

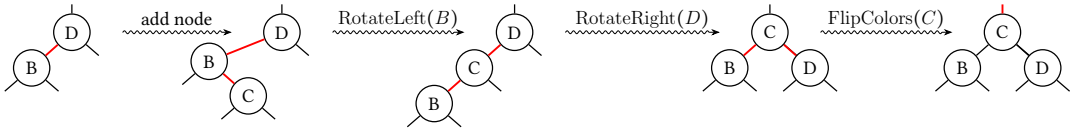
Once we understand the general idea on how trees are being modified to preserve perfect black balance and to remain valid red-black trees, we are going to explore the code for these operations and the complete algorithm. In Figure 5.8 we present the three auxiliary operations on red-black trees. We prove some



(a) insert(E)



(b) insert(A)



(c) insert(C)

Figure 5.7: Insertion on a 3-node

```
static method RotateRight(
  node: TNode,
  ghost sk: tree<TNode>
)
  returns (
    newNode: TNode,
    ghost newSk: tree<TNode>
  )
{
  newNode := node.left;
  node.left := newNode.right;
  newNode.right := node;
  newNode.color := node.color;
  node.color := Red;
  newSk := Node(
    sk.left.left, newNode,
    Node(sk.left.right, node, sk.right)
  );
}
```

(a) Right rotation

```
static method RotateLeft(
  node: TNode,
  ghost sk: tree<TNode>
)
  returns (
    newNode: TNode,
    ghost newSk: tree<TNode>
  )
{
  newNode := node.right;
  node.right := newNode.left;
  newNode.left := node;
  newNode.color := node.color;
  node.color := Red;
  newSk := Node(
    Node(sk.left, node, sk.right.left),
    newNode, sk.right.right
  );
}
```

(b) Left rotation

```
static method FlipColors(node: TNode, ghost sk: tree<TNode>)
{
  node.color := NegColor(node.color);
  node.left.color := NegColor(node.left.color);
  node.right.color := NegColor(node.right.color);
}
```

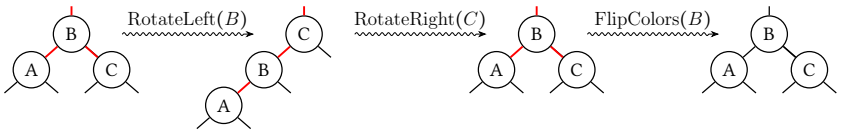
(c) Flip colors

Figure 5.8: Auxiliary operations

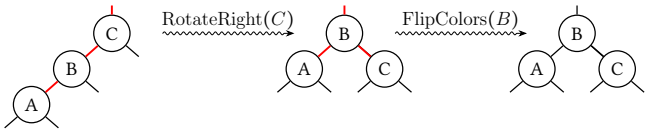
properties about those methods, like the preservation of perfect black balance (`BlackHeight(newSk)==old(BlackHeight(sk))`), that are not shown in that figure.

The operations we described to turn an invalid red-black tree (after a new node is inserted) into a valid red-black tree are encapsulated into the `Restore` method of Figure 5.10. We have omitted the proofs inside the body of the method[2] but the specification gives us an idea on what properties the method relies on to produce a valid red-black tree. Namely, the first three preconditions enumerate the cases that this method cannot receive. Those cases are not produced neither by the insertion algorithm nor by the deletion algorithm. If it received any of those trees, the method would return an invalid red-black tree or it would change the black height. To illustrate this, in Figure 5.9 we show the result of ap-

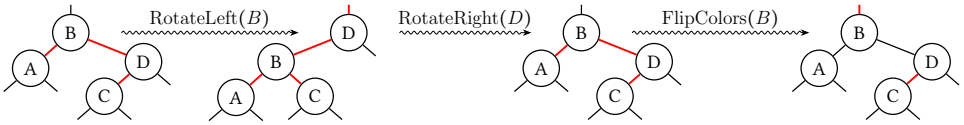
[2]Its body accounts for more than 100 lines of code and proofs.



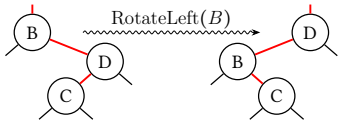
(a) `isRed(node) && isRed(node.left) && isRed(node.right)` (forbidden by precondition 2)



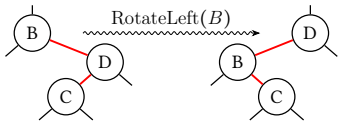
(b) `isRed(node) && isRed(node.left) && isRed(node.left.left)` (forbidden by precondition 3)



(c) `isBlack(node) && isRed(node.left) && isRed(node.right) && isRed(node.right.left)` (accepted)



(d) `isRed(node) && isBlack(node.left) && isRed(node.right) && isRed(node.right.left)` (forbidden by precondition 1)



(e) `isBlack(node) && isBlack(node.left) && isRed(node.right) && isRed(node.right.left)` (forbidden by precondition 1)

Figure 5.9: `Restore` executions on different cases

```
static method Restore(node: TNode, ghost sk: tree<TNode>)
    returns (newNode: TNode, ghost newSk: tree<TNode>)
  modifies elems(sk)

① requires isRed(node.right) ∧ isRed(node.right.left) ⟹
    isBlack(node) ∧ isRed(node.left)
② requires ¬(isRed(node) ∧ isRed(node.left) ∧ isRed(node.right))
③ requires ¬(isRed(node) ∧ isRed(node.left) ∧ isRed(node.left.left))

  requires BlackHeight(sk.left) = BlackHeight(sk.right)
  requires RedBlackTreeRec(sk.left)
  requires RedBlackTreeRec(sk.right)

  ensures BlackHeight(newSk) = old(BlackHeight(sk))
  ensures RedBlackTreeRec(newSk)
  ensures old(isBlack(node)) ∧ isRed(newNode) ⟹ ¬isRed(newNode.left)
{
  newNode := node;
  newSk := sk;

  if isRed(newNode.right) {
    newNode, newSk := GRotateLeft(newNode, newSk);
  }

  if isRed(newNode.left) ∧ isRed(newNode.left.left) {
    newNode, newSk := GRotateRight(newNode, newSk);
  }

  if isRed(newNode.left) ∧ isRed(newNode.right) {
    GFlipColors(newNode, newSk);
  }
}
```

Figure 5.10: `Restore` helper method

plying the `Restore` method to some example trees. In Figures 5.9a and 5.9b the resulting tree after executing `Restore` is a valid red-black tree but the black height has been altered, making it unsuitable for our purposes (it could produce an imbalance in the rest of the tree). Figure 5.9c illustrates the execution of `Restore` on a tree that complies with the preconditions, so the final result is a valid red-black tree with the same black height as the received tree. Figures 5.9d and 5.9e, nonetheless, illustrate two cases that produce invalid red-black trees. In Figure 5.9 we have shown all the trees that `Restore` cannot fix (5.9a, 5.9b, 5.9d, 5.9e). The proof that all the other cases produce a valid red-black tree is omitted, but Dafny can prove it with some help.

In Figure 5.11 we show the complete algorithm for insertion on red-black trees. Its verification is based on the last precondition and the first postcondition.

Those two lines are needed to prove the preconditions to the `Restore` method.

The precondition can be thought of as an alternative definition of red-black trees. Nodes cannot have two red links connected to them, but that property can be expressed in two ways: the way we used previously in this chapter (`!(isRed(node.left) && isRed(node.left.left))`, as in Figure 5.4[3]) or the one we add as a precondition to `InsertRec`. Both are equivalent since the definition of red-black trees is recursive and requires that all the nodes comply with it, but we cannot prove the second one by relying only on the first one. To solve this, we add it as a precondition to `InsertRec` and it is proved almost automatically. This precondition is used to prove, among other properties, that if the root of the given tree was red; then, after insertion, only one of its children can be red, ruling out the case of `isRed(node) && isRed(node.left) && isRed(node.right)` (second precondition of `Restore`).

The postcondition says that if the root has been turned from black to red, then its left child is black. It is used to prove the first precondition of `Restore`, asserting that `!(isRed(newNode.right) && isRed(newNode.right.left))`. That property is actually stronger than what `Restore` needs, but Dafny verifies it without problem nonetheless.

To finish, in Figure 5.12 we show the user-facing method for insertion. The only thing we need to do is turn the root black in case it was turned to red. Apart from that, the job has already been done.

5.4 Deletion

The verification and implementation of deletion for red-black trees proceeds similarly to binary search trees: first implement the removal of the minimum and then use it to implement `RemoveRec`. As with insertion for red-black trees, we will use the `Restore` method on the way up to return a valid red-black tree, but this time we also need to perform rotations on the way down.

We will begin our study of deletion on red-black trees with method `RemoveMinRec`, as shown in Figure 5.13. Similar to insertion, it has as a precondition the alternative definition of red-black trees. But this time we add another precondition: the received node or its left child should be red (not both, because of the previous precondition). This precondition ensures that when we arrive at the base case (the left child is null) the received node is red, ensuring that by

[3] In that figure that property is expressed with other syntax because it is describing the skeleton, not the node, but they are logically equivalent.

```
static method InsertRec(node: TNode?, ghost sk: tree<TNode>, k: K, v: V)
    returns (newNode: TNode, ghost newSk: tree<TNode>, ghost insertedNode:TNode)
  modifies elems(sk)
  requires ValidRec(node, sk) ∧ SearchTreeRec(sk) ∧ RedBlackTreeRec(sk)

  requires ¬(isRed(node) ∧ isRed(node.left))
  ensures old(isBlack(node)) ∧ isRed(newNode) ⟹ ¬isRed(newNode.left)

  ensures ValidRec(newNode, newSk) ∧ SearchTreeRec(newSk)
  ensures RedBlackTreeRec(newSk) ∧ BlackHeight(newSk) = old(BlackHeight(sk))
  ensures ModelRec(newSk) = old(ModelRec(sk))[k := v]

  ensures elems(newSk) = elems(sk) + {insertedNode}
  ensures insertedNode.key = k ∧ insertedNode.value = v
  ensures ∀ n | n ∈ elems(sk) ∧ old(n.key) ≠ k •
    n.key = old(n.key) ∧ n.value = old(n.value)
{
  if node = null {
    newNode := new TNode.RedBlack(null, k, v, null, Red);
    newSk := Node(Empty, newNode, Empty);
    insertedNode := newNode;
  } else {
    newNode := node;
    if k = node.key {
      node.value := v;
      newSk := sk;
      insertedNode := node;
    } else if node.key < k {
      ghost var newSkRight;
      node.right, newSkRight, insertedNode := InsertRec(node.right, sk.right, k, v);
      newSk := Node(sk.left, node, newSkRight);
    } else if k < node.key {
      ghost var newSkLeft;
      node.left, newSkLeft, insertedNode := InsertRec(node.left, sk.left, k, v);
      newSk := Node(newSkLeft, node, sk.right);
    } else {
      assert false;
    }
  }

  assert ¬(isRed(newNode.right) ∧ isRed(newNode.right.left));
  assert isRed(newNode.left) ∧ isRed(newNode.right) ⟹
      isBlack(newNode) ∧ ¬isRed(newNode.left.left);
  assert isRed(newNode.left) ∧ isRed(newNode.left.left) ⟹ isBlack(newNode);

  label PreRestore:
  var newNewNode, newNewSk := Restore(newNode, newSk);

  assert old(isBlack(node)) ∧ isRed(newNewNode) ⟹ ¬isRed(newNewNode.left) by {
    assert old@PreRestore(isBlack(node)) ∧ isRed(newNewNode) ⟹
      ¬isRed(newNewNode.left);
    if old(isBlack(node)) {
      assert old@PreRestore(newNode.color) = old(node.color);
      assert old@PreRestore(newNode.color).Black?;
    }
  }
  newNode, newSk := newNewNode, newNewSk;
}
```

Figure 5.11: Recursive algorithm for insertion on red-black trees

```
method Insert(k: K, v: V)
  modifies this, Repr()
  requires Valid() ∧ SearchTree() ∧ RedBlackTree()
  ensures Valid() ∧ SearchTree() ∧ RedBlackTree()
  ensures Model() = old(Model())[k := v]
{
  ghost var z;
  root, skeleton, z := InsertRec(root, skeleton, k, v);
  root.color := Black;
}
```

Figure 5.12: Insertion on red-black trees

returning the right child we preserve black height. Complying with that precondition, nonetheless, is not trivial when we perform the recursive call. To that end we define the `MoveRedLeft` method in Figure 5.14. That method peforms a series of rotations and flip-colors to ensure that either `newNode.left` or `newNode.left.left` is red while preserving perfect black balance and keeping the black height the same. To better understand these operations we show the two possible cases in Figure 5.15. This method is used before the recursive call of `RemoveMinRec` in case the precondition does not hold.

Until now we have worked to comply with the preconditions, but `RemoveMinRec` also has important properties as postconditions. They are not needed to prove that the returned tree is a valid red-black tree, but we will use them to verify the final method `RemoveRec`. For this reason we will postpone their discussion until we arrive at that method.

To end the discussion about method `RemoveMinRec` we want to describe a problem that Dafny had verifying this method. At first Dafny could not verify any property, it would simply get stuck. The problem, we discovered, was that it could not verify the termination of the method. This is reasonable since with the introduction of the call to `MoveRedLeft` we are no longer recursing purely structurally. To solve this issue we add a `decreases`-clause to inform Dafny that it should focus on the size of the tree (defined as the number of nodes of the tree). Then in `MoveRedLeft` we prove that the left child of the node it returns is smaller than the node it received. This way Dafny's termination checker is satisfied and we can verify the properties described.

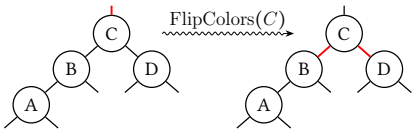
We show the complete implementation of `RemoveRec` in Figure 5.16. This method is the most complicated we have verified in this work. Its implementation follows the same structure as deletion for binary search trees but we need to perform some rotations while descending the tree, the same as we did in

```
static method RemoveMinRec(node: TNode, ghost sk: tree<TNode>)
    returns (newNode: TNode?, ghost newSk: tree<TNode>, removedNode: TNode)
  decreases size(sk)
  modifies elems(sk)

  requires RedBlackTreeRec(sk)
  requires ¬(isRed(node) ∧ isRed(node.left))
  requires isRed(node) ∨ isRed(node.left)

  ensures RedBlackTreeRec(newSk)
  ensures BlackHeight(newSk) = old(BlackHeight(sk))
  ensures isRed(newNode) ⟹ ¬isRed(newNode.left)
  ensures old(¬isRed(node)) ⟹ ¬isRed(newNode)
{
  if node.left = null {
    calc = {
      BlackHeight(sk);
      max(BlackHeight(sk.left), BlackHeight(sk.right));
      BlackHeight(sk.right);
    }
    newNode := node.right;
    newSk := sk.right;
    removedNode := node;
  } else {
    newNode := node;
    newSk := sk;
    if isBlack(newNode.left) ∧ ¬isRed(newNode.left.left) {
      newNode, newSk := MoveRedLeft(newNode, newSk);
    }
    ghost var newSkLeft;
    newNode.left, newSkLeft, removedNode := RemoveMinRec(newNode.left, newSk.left);
    newSk := Node(newSkLeft, newNode, newSk.right);
    newNode, newSk := Restore(newNode, newSk);
  }
}
```

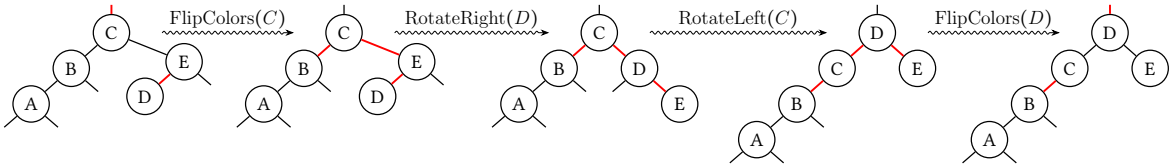
Figure 5.13: Remove the minimum of a red-black tree

```
static method MoveRedLeft(node: TNode, ghost sk: tree<TNode>)
    returns (newNode: TNode, ghost newSk: tree<TNode>)
  modifies elems(sk)

  requires RedBlackTreeRec(sk)
  requires isRed(node) ∧ isBlack(node.left) ∧ isBlack(node.right)
  requires ¬isRed(node.left.left)

  ensures RedBlackTreeRec(newSk.left) ∧ RedBlackTreeRec(newSk.right)
  ensures BlackHeight(newSk.right) = BlackHeight(newSk.left)
  ensures BlackHeight(newSk) = old(BlackHeight(sk))
  ensures newNode.left ≠ null ∧ newNode.right ≠ null
  ensures isRed(newNode.left) ∨ isRed(newNode.left.left)
  ensures
    ∨ (¬isRed(newNode) ∧ isRed(newNode.left) ∧ isRed(newNode.right)
        ∧ ¬isRed(newNode.right.left) ∧ ¬isRed(newNode.left.left))
    ∨ (isRed(newNode) ∧ ¬isRed(newNode.left) ∧ ¬isRed(newNode.right)
        ∧ isRed(newNode.left.left))
  ensures size(newSk) = size(sk)
  ensures size(newSk.left) < size(sk)
{
  FlipColors(node, sk);
  newNode := node;
  newSk := sk;

  if isRed(newNode.right.left) {
    ghost var newSkRight;
    newNode.right, newSkRight := RotateRight(newNode.right, newSk.right);
    newSk := Node(newSk.left, newNode, newSkRight);

    newNode, newSk := RotateLeft(newNode, newSk);

    FlipColors(newNode, newSk);
  }
}
```

Figure 5.14: Move a red link to the left



(a) `!isRed(node.right.left)`



(b) `isRed(node.right.left)`

Figure 5.15: `MoveRedLeft` possible executions

`RemoveMinRec`. The preconditions in this case account for the case when the input node is null but the algorithm follows the same principles to comply with it: when we recur on the left we call `MoveRedLeft`, and when we recur on the right we call the `MoveRedRight` method instead, analogous to `MoveRedLeft`. Note, however, that we call `RotateRight` in the case that the left child of the received node is red. In that case, we comply with the precondition for the right recursive call (and the call to `RemoveMinRec`) without the need to call `MoveRedRight`.

The verification of this method needs some postconditions, marked with numbers, that we did not explore when we discussed `RemoveMinRec`. Both `RemoveRec` and `RemoveMinRec` have to comply with two postconditions to verify that the returned tree of `RemoveRec` is a valid red-black tree. The first is the alternative definition of being a valid red-black tree that we have already discussed. Another way to think about this postcondition is that it is an strengthened version of the postcondition of `InsertRec` by removing one of the premises of the implication (that the received node was black). The postcondition of `InsertRec` works in that method because we use it when we perform recursion on the right, and in that case we know that the node is always black (by the definition of red-black trees the right child is always black). In `RemoveRec`, however, we sometimes turn the right child to red, at least temporarily, to comply with the preconditions that we have described. By strengthening the postcondi-

tion we can prove the preconditions of `Restore`.

Note that the postcondition of `InsertRec` has to stay as it is because the strengthened version we add to `RemoveRec` does not hold in insertion. In fact, to prove the strengthened postcondition we need to add other postcondition: if the received node was not red, then the returned node will not be either. Insertion is built on the principle that we add a new node with a red link when we receive a black node, so this postcondition would not hold. Deletion, however, is concerned with removing nodes. To understand the proof of this postcondition you have to consider these two cases when the received node is black (if it is null the proof is trivial):

- The received node is the node that has to be removed: then we will call `RemoveMinRec` on the right child and the color (black) will be preserved.
- The received node does not have to be removed: then we will perform recursion either on the left or the right. In this case `MoveRedLeft` and `MoveRedRight` will not be executed so the node will remain black until recursion. The recursive call cannot alter the color of the node. And, finally, Dafny has to verify that there is no case in which `Restore` turns the color to red.

To prove the first postcondition we need two assumptions: the two postconditions hold for the tree returned by recursion (the induction hypothesis) and `Restore` complies with its specification[4]. Once we have that, Dafny can verify all the postconditions of the method (given enough time).

To illustrate how hard Dafny has to work to prove this method we add to the source code of Figure 5.16 the number of minutes that Dafny needs to verify each branch. The method as a whole needs more than one hour of verification in the machine of the author. Fortunately, the proofs are simple since most of the heavy-lifting is done by Dafny. However, a great effort had to be made to find the postconditions that Dafny needs.

During this section we have described the verification of methods `RemoveMinRec` and `RemoveRec`. We have focused on the verification of the invariant of red-black trees, but the verification of the modification of the model has not yet been completed. The proof will have to take into account the auxiliary operations (rotations, `MoveRedRight`, etc.), but it should not be too different from the proofs of Chapter 4 since those operations do not change the model.

[4]The specification was expanded to accomodate for the special cases of deletion.

```
static method RemoveRec(node: TNode?, ghost sk: tree<TNode>, k: K)
    returns (newNode: TNode?, ghost newSk: tree<TNode>, ghost removedNode: TNode?)
  decreases size(sk)
  modifies elems(sk)

  requires RedBlackTreeRec(sk)
  requires ¬(isRed(node) ∧ isRed(node.left))
  requires node = null ∨ isRed(node) ∨ isRed(node.left)

  ensures RedBlackTreeRec(newSk)
  ensures BlackHeight(newSk) = old(BlackHeight(sk))
① ensures isRed(newNode) ⟹ ¬isRed(newNode.left)
② ensures old(¬isRed(node)) ⟹ ¬isRed(newNode)
{
  newNode := node;
  newSk := sk;
  if newNode = null {
    removedNode := null;
  } else {
    if node.key > k { // 3'50"
      if isBlack(newNode.left) ∧ ¬isRed(newNode.left.left) {
        newNode, newSk := MoveRedLeft(newNode, newSk);
      }
      ghost var newSkLeft;
      newNode.left, newSkLeft, removedNode :=
        RemoveRec(newNode.left, newSk.left, k);
      newSk := Node(newSkLeft, newNode, newSk.right);
    } else {
      if isRed(newNode.left) {
        newNode, newSk := RotateRight(newNode, newSk);
      }

      if k = newNode.key ∧ newNode.right = null {
        removedNode := newNode;
        newNode := newNode.left;
        newSk := newSk.left;
        return;
      }

      if isBlack(newNode.right) ∧ ¬isRed(newNode.right.left) {
        newNode, newSk := MoveRedRight(newNode, newSk);
      }

      if k ≠ newNode.key {  // 12'20"
        ghost var newSkRight;
        newNode.right, newSkRight, removedNode :=
          RemoveRec(newNode.right, newSk.right, k);
        newSk := Node(newSk.left, newNode, newSkRight);
      } else {  // 25'
        removedNode := newNode;
        ghost var newSkRight;
        var minNode;
        newNode.right, newSkRight, minNode :=
          RemoveMinRec(newNode.right, newSk.right);
        minNode.left := newNode.left;
        minNode.right := newNode.right;
        minNode.color := newNode.color;
        newNode := minNode;
        newSk := Node(newSk.left, newNode, newSkRight);
      }
    }
    newNode, newSk := Restore(newNode, newSk);
  }
}
```

Figure 5.16: Recursive algorithm for deletion on red-black trees

Chapter 6

Iterators in binary search trees

We have already explored the specification of iterators in sets, maps and multisets, but we have not shown yet how we can implement such iterators in tree-like data structures. Although this part of our work has not been fully developed yet, we still want to show the progress we have made on this front. Namely, we have implemented and verified the key operations of iterators over immutable trees. Our work is focused on linked data structures, so this is not our final goal, but it is a step in the path to a fully verified implementation of iterators for binary search trees (and eventually red-black trees). We have also made some progress on the verification and implementation of iterators for the binary search trees that we have shown in previous chapters, but that work will not be shown in this document since it is too experimental. The interested reader can search in the source code provided along this document[1]. We expect to implement the ideas discussed in this chapter to finish the verification of that code.

The main class of our trees is shown in Figure 6.1. Apart from being an immutable tree instead of a linked data structure, this definition is not too different from what we have already seen. The only addition is a ghost field `iterators` that stores all the iterators that point to the tree. That property is ensured in the representation invariant using the `Parent` function of the iterators. We also implement the `First` method by calling the constructor of iterators.

In Figure 6.2 we show the definition of iterators. These iterators traverse the tree in inorder with the help of a stack. The top of the stack represents the currently pointed element. While descending the tree during the traversal, the

[1]The code from this chapter is extracted from the `src/tree/layer3/IterTestImpl.dfy` file, and the experimental implementation of iterators for heap-allocated trees can be found in the `src/tree/layer3/OrderedMapImplIter.dfy` file.

```
datatype Tree = Empty | Node(left: Tree, key: K, value: V, right: Tree)

class UnorderedMap {
  var tree: Tree
  ghost var iterators: set<UnorderedMapIterator>

  function Repr(): set<object>
    reads this
  {
    {this} + this.iterators
  }

  predicate Valid()
    reads this, Repr()
  {
    ∧ SearchTreeRec(tree)
    ∧ ∀ it | it ∈ iterators • it.Parent() = this ∧ it ≠ this
  }

  function Model(): map<K,V>
    reads this, Repr()
    requires Valid()

  method First() returns (it: UnorderedMapIterator)
    modifies this, Repr()
    requires Valid()
    ensures Valid()
    ensures Model() = old(Model())

    ensures Iterators() = {it} + old(Iterators())
    ensures it.Valid()
    ensures it.Parent() = this
    ensures it.Traversed() = {}
  {
    it := new UnorderedMapIterator(this);
    iterators := iterators + { it };
  }
}
```

Figure 6.1: `UnorderedMap` class

stack is filled, and when the traversal cannot proceed downwards (the right child of the top of the stack is null), the stack is used to go back up the tree. This process will be detailed and formalized in the rest of this chapter. The Dafny definition of iterators, apart from the parent and the stack fields, has the following ghost fields: the traversed elements, the traversed keys and the inorder of the parent. The properties of each of these fields are defined in the representation invariant:

- `stack`: the nodes of the stack should not be null.
- `inorderParent`: the `inorderParent` field is actually the inorder traversal of the parent, no key is repeated and every element of the inorder is in the model and viceversa.
- `traversed` and `traversedKeys`: their length is the same, which in turn is less than that of the inorder and `traversedKeys` is the sequence of the left components of `traversed`.
- Relation between them: the inorder of the parent is composed of the traversed elements and all the inorder traversals of the right subtrees of the nodes in the stack.

Some of these properties could be derived from others, for example from the fact that the tree is a binary search tree. We keep them, nonetheless, since they are not difficult to prove and they may give hints to Dafny on how to verify our methods.

Now we want to discuss the `Next` method, but before we show its code we will see what operations this method does through illustrations. In Figure 6.3 we illustrate the state of the tree before the `Next` operation is called. The elements of the stack are marked in blue and the tree is divided in two parts: the left part is composed of the already traversed nodes, and the right is composed of the nodes of the stack and all of their right subtrees (`inorderStack(stack)`). Our objective is to consume the next node, which is always at the top of the stack. In Figure 6.4 we illustrate the state of the tree after we consume that node, that is, we remove it from `stack` and add it to `traversed`. The tree is now divided in three parts: the traversed elements, the right subtree of the (previous) top of the stack (called T in the diagram) and the nodes from the stack along with their right subtrees (`inorderStack(stack)`). To restore the invariant we need to include all of the nodes in T into `inorderStack(stack)`. To do so we descend that tree through the left links until there is no left child, adding every

```
class UnorderedMapIterator {
  var parent: UnorderedMap
  var stack: seq<Tree>
  ghost var traversed: seq<(K, V)>
  ghost var traversedKeys: seq<K>
  ghost var inorderParent: seq<(K, V)>

  function Parent(): UnorderedMap
    reads this
  { parent }

  function inorderStack(st: seq<Tree>): seq<(K, V)>
  {
    if |st| = 0 then
      []
    else
      var top: Tree := st[|st| - 1];
      match top {
        case Empty ⇒ inorderStack(st[0..|st| - 1])
        case Node(_, k, v, r) ⇒ [(k, v)] + inorder(r) + inorderStack(st[0..|st| - 1])
      }
  }

  predicate Valid()
    reads this, Parent(), Parent().Repr()
  {
    ∧ parent.Valid()

    // stack
    ∧ (∀ i | 0 ≤ i < |stack| • stack[i].Node?)

    // inorderParent
    ∧ inorderParent = inorder(parent.tree)
    ∧ (∀ i, j | 0 ≤ i < j < |inorderParent| •
        inorderParent[i].0 ≠ inorderParent[j].0)
    ∧ (∀ p | p ∈ inorderParent • p ∈ parent.Model().Items)
    ∧ (∀ p | p ∈ parent.Model().Items • p ∈ inorderParent)

    // traversed, traversedKeys
    ∧ |traversedKeys| = |traversed| ≤ |inorderParent|
    ∧ (∀ i | 0 ≤ i < |traversedKeys| • traversed[i].0 = traversedKeys[i])

    // Relation between them
    ∧ traversed + inorderStack(stack) = inorderParent
  }
```

Figure 6.2: `RemoveRec` method

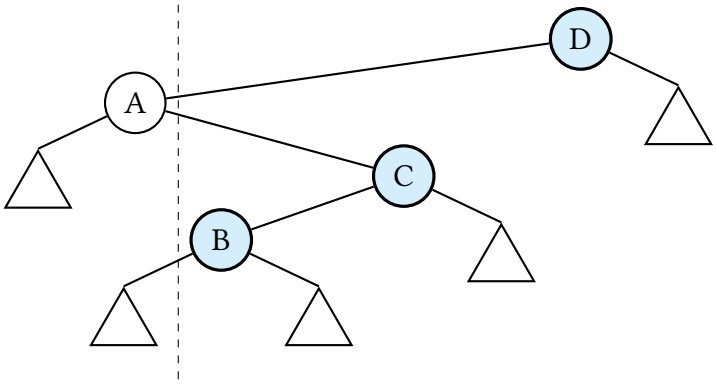


Figure 6.3: State of the tree before calling `Next`

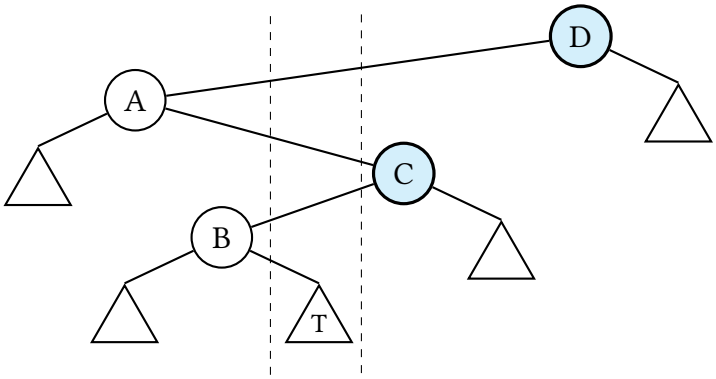


Figure 6.4: State after consuming the next node

node to the stack in the process, as shown in Figure 6.5. We call this operation *descend and push*. With this operation we have achieved two goals: the top of the stack now points to the least element of the non-traversed elements, and all the nodes of T have been added to the right part of the diagram, as shown in Figure 6.6, where T' represents the T tree after *descend and push* has been executed on it. We have succesfully restored the invariant after consuming the next node.

The code for the algorithm we have described is in Figure 6.7. The inorder of the tree is divided as we have explained, and the rest of the algorithm is a matter of modifying correctly the fields and calling `DescendAndPush`. The constructor of iterators, shown in that figure too, also calls `DescendAndPush` to build the initial stack, with no additional operations apart from initializing

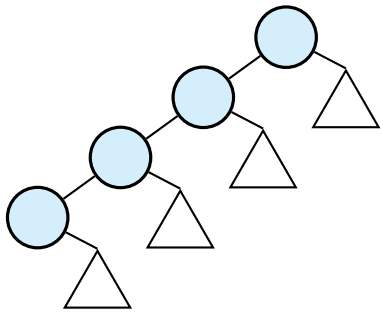


Figure 6.5: Result of *descend and push*

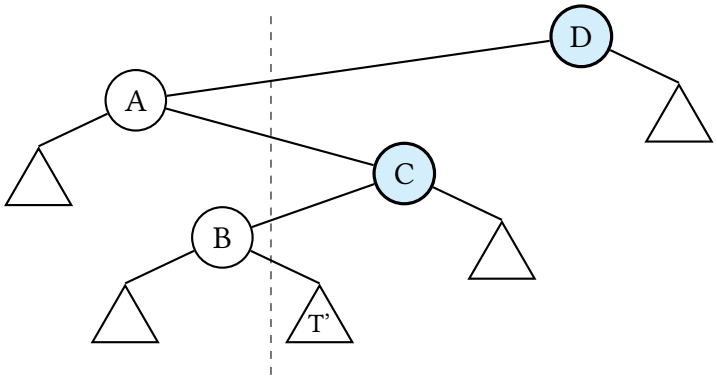


Figure 6.6: State after calling `Next`

the fields. The code for `DescendAndPush` is shown in Figure 6.8: a simple `while`-loop that restores the invariant.

With this implementation and verification completed, we can adapt these ideas to our linked trees. The new verification should thoroughly specify how the tree is modified during the descend and push and other operations, a challenge that we leave for future work.

```
method Next() returns (p: pairKV)
  modifies this
  requires HasNext()
{
  var top := stack[|stack| - 1];
  p := (top.key, top.value);

  calc = {
    inorderParent;
    traversed + inorderStack(stack);
    traversed + inorderStack(stack[..|stack| - 1] + [top]);
    traversed + ([p] + inorder(top.right) + inorderStack(stack[..|stack| - 1]));
    (traversed + [p]) + inorder(top.right) + inorderStack(stack[..|stack| - 1]);
  }
  stack := stack[..|stack| - 1];
  traversed := traversed + [p];
  traversedKeys := traversedKeys + [top.key];

  assert traversed + inorder(top.right) + inorderStack(stack) = inorderParent;
  DescendAndPush(top.right);

  assert traversed + inorderStack(stack) = inorderParent;
}

constructor (parent: UnorderedMap)
  requires parent.Valid()
  ensures Valid()
  ensures Traversed() = {}
  ensures Parent() = parent
{
  this.parent := parent;
  this.traversed := [];
  this.traversedKeys := [];
  this.stack := [];
  this.inorderParent := inorder(parent.tree);
  DescendAndPush(parent.tree);
}
```

Figure 6.7: `Next` method and the iterator constructor

```
method DescendAndPush(t: Tree)
  modifies this`stack
  requires ∀ i | 0 ≤ i < |stack| • stack[i].Node?
  ensures  ∀ i | 0 ≤ i < |stack| • stack[i].Node?
  requires traversed + inorder(t) + inorderStack(stack) = inorderParent
  ensures traversed + inorderStack(stack) = inorderParent
{
  var t' := t;
  while t'.Node?
    decreases t'
    invariant ∀ i | 0 ≤ i < |stack| • stack[i].Node?
    invariant traversed + inorder(t') + inorderStack(stack) = inorderParent
  {
    stack := stack + [t'];
    t' := t'.left;
  }
}
```

Figure 6.8: `DescendAndPush` method

Chapter 7

Conclusions

In this chapter we will give a general review of this work, specially regarding the goals set in Chapter 1. The results of this work are the following:

- We have formalized the specification of sets, multisets and maps.
- We have implemented and verified binary search trees, including search, insertion and deletion operations.
- We have implemented and verified red-black trees and insertion and deletion on them.[1]
- We have verified the implementation of the map ADT using linked lists.
- We have verified the implementation of the operators (without iterators) of the map ADT with binary search trees (not with red-black trees, but their interfaces are the same).
- We have verified a test implementation of iterators in binary search trees. This implementation uses immutable trees instead of the heap-allocated trees used in the rest of our work. We hope to use this transient verification to verify the final implementation of the iterators for the linked trees that we have developed. This verification is already in progress and its completion is left for future work.

[1] Some parts of the verification of deletion have not been completed, but they are not expected to cause problems or involve new techniques. In particular, the verification of the changes made to the model.

7.1 Difficulties encountered and lessons learned

Dafny may be an automatic verifier, but that automation is served at the expense of interactivity. Verification times of several minutes were usual (breaking almost completely the feedback loop), and sometimes the verification times reached 10 to 20 minutes, or one hour in the most extreme case (like `RemoveRec` of Chapter 5). These issues are not inconveniences that should be overlooked. If we want to make formal verification mainstream (or at least as mainstream as it can be), the user experience should be on par on what a programmer expects from non-verified programming environments. Verification environments should also be predictable and reliable. If something cannot be proven automatically, the verifier should fail quickly and guide the programmer to write the proofs manually.

Red-black trees are an elegant data structure that simplifies the implementation and understanding of self-balancing trees, but they are still a much more complex data structure than the usual binary search trees. Since this work focused on verification, I needed to not only understand red-black trees, but to be able to express their key properties and the invariants of their operations in a sufficiently simple way that Dafny could use for verification. This involved discovering invariants that were not initially found in the literature and going deep into each and every case of the correctness proofs. All in all, I think I now have a much deeper understanding of red-black trees than the one I had when I initially studied them in an undergraduate course.

7.2 Future work

We left for future work finishing the uncomplete parts of this work, namely the implementation and verification of iterators for binary search trees and the last parts of the verification of deletion for red-black trees. Some other extensions of the current work are the following:

- Implementation and verification of threaded binary trees, a version of binary trees that should help with the implementation of iterators for red-black trees. In this version, a new field is added to nodes that points to the next node in the traversal of the tree. These links pose a challenge to the implementation and verification since the have to be maintained after tree operations.

- Specification, implementation and verification of graphs, using the map and set ADTs developed in this work.

Bibliography

[1] Jorge Blázquez. Verification of linked data structures in Dafny. BS thesis, Facultad de Informática, Universidad Complutense de Madrid, 2021. Available at `https://eprints.ucm.es/docencia.html`.

[2] Interface of the set ADT in the C++ Standard. `https://en.cppreference.com/w/cpp/container/set`.

[3] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35:583–599, 5 2005.

[4] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, 2004.

[5] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Proceedings*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.

[6] K.R.M. Leino. Developing verified programs with Dafny. In *VSTTE 2012*, pages 82–82, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Rustan Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School 2008*, pages 1–36.

[8] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[9] Ricardo Peña. An Assertional Proof of Red–Black Trees Using Dafny. *Journal of Automated Reasoning*, 64(4):767–791, April 2020.

[10] Nadia Polikarpova, Julian Tschannen, and Carlo A. Furia. A fully verified container library. *Formal Aspects of Computing*, 30(5):495 – 523, 2018.

[11] Robert Sedgewick and Kevin Wayne. *Algorithms, Fourth Edition.* Addison-Wesley Professional, 2011.