Imperial College London
Department of Computing

Verifying a balanced-tree index implementation in VeriFast

James Fisher

Submitted in partial fulfilment of the requirements for
the MSc Degree in Computing Science of Imperial College London

September 2012

Table of contents

Abstract

The index is a fundamental abstract data type used everywhere. Being tuned for high performance, implementations are complex, making verification desirable. *Practicable* verification is the goal, and hand proofs fail in this respect. VeriFast is one promising machine assistant for program verification. This report documents my verification of a Left-Leaning Red-Black Tree implementation in C using VeriFast. I cover five standard index operations: search, insert, remove the minimum, remove, and iterate. I find that, despite shortcomings in my verification and in the tools available, machine-assisted verification is practicable, at least for the class of well-understood separation-logic-friendly data structures.

This report is dedicated to Barclays PLC, without whose loan at usurious interest rates none of this would have been possible.

Overview

The index is an abstract data structure used by nearly all substantial real-world programs. Crucial to the reliability of real-world software, then, are reliable index implementations. However, index implementations are many, with each having its own desirable characteristics, and most are algorithmically complex.

Index structures and algorithms are therefore prime targets for verification. Most families of index structures have manual proofs of correctness for their basic algorithms. Yet this leaves room for errors in the manual proof, errors in deviation from and extension of the algorithm, and errors in implementation.

This implies we need specific proofs of correctness for each specific implementation. For this task, hand proofs are impractical: they are subject to error, they require in-depth knowledge of the semantics of the implementation language, and they are subject to divergence from the implementation as it changes.

A potential solution is machine-assisted verification. The verification tool provides confidence that proofs are correct, defines the language semantics, and can be run with little effort when the implementation changes. One such tool for this is VeriFast, which provides the means for specification and verification of programs written in C.

This report is a case study in practical verification: it covers the use of VeriFast to specify and verify a specific implementation in C of a particular index data structure: the Left-Leaning Red-Black Tree (LLRB). This data structure is a simple variant on Red-Black Trees, which I chose to study because they are the most popular index structure according to my survey of standard libraries.

The report makes no assumptions beyond a basic knowledge of C and mathematics. As such, it is intended as an introduction to multiple topics: the LLRB data structure and algorithms, how to write a good abstract specification of it, what it means to verify a program against the specification, and how to practically approach this in VeriFast.

I start with a background on the history and the state-of-the-art. I apply this to index algorithms, and then to program verification. I continue with a formal, but general, discussion of both: what does it mean to be a mathematical index? what does it mean for a program to truly be verified?

The report proper then begins. Rather than introduce the syntax and semantics of VeriFast up-front before using it, I use the LLRB material as a tutorial on VeriFast. Nor do I introduce the LLRB data structure and algorithms up-front; they are introduced gradually throughout the report.

I begin by formalizing our concept of an index using the VeriFast specification language. The next sections introduce five standard index operations: search, insert, remove the minimum, remove, and iterate. For each, I start with a discussion of the operation, and then a formalization of it in VeriFast as an operation on our defined index data structure. I continue with an informal description of the algorithm on binary search trees, and then how this has to be altered to work on LLRBs. I finish each section with a walk through of verifying my C implementation of that algorithm using the tools offered by VeriFast.

I finish the report with an evaluation of my project and the tools it uses. To what extent was my attempt at verification a model for how it should be done? The algorithms I chose to study were verified against the specification that I wrote, and in this narrow sense the project was successful. However, there are faults in both my work and in the tools available, and I make suggestions for how both could be improved. However, I also find that VeriFast represents, for some classes of programs, the closest that a real-world imperative programmer might get to practical verification as an aid to programming.

Background

What is an index? What can they be used for? What kinds are there? Where are they used? Which are better? What is a "Left-Leaning Red-Black Tree"? What is verification? Why do we need it? What is the relationship between indexes and verification? Here I address these general questions that may arise when attempting to explain what this project is about.

What is an index?

An index is a precomputed partial function. This means it contains, in some sense, an explicit representation of each input-output pair. Such precomputed functions are everywhere. A function might not be computable in terms of the input, as in an association of names to telephone numbers, in which case an index is an appropriate representation. A pure computation might be too expensive to perform multiple times, in which case the result can be stored for possible future use — a technique known as "caching." [1]

What can we do with an index? The most fundamental thing we can do is use it as a function: input a value and get a value out. A notable difference is that whereas the domain of a function can be infinite, the domain of an index is bounded by memory use. Therefore an index is a *partial* function: a given value in the domain type might not have an associated value in the range type of the index.

We can also mutate the index. Functionally, this means there exist functions from indexes to indexes. Some of these mutations have equivalents in the realm of ordinary functions: for example, function composition is equivalent to mapping over the range of a function. Others are more unique. An *insert* function, given a new key and new value, takes an index to a new index in which the new key is paired with the new value. A *delete* function, given a key, takes an index to a new index in which the given key is not paired with a value. A *union* function is a binary operation on indexes that performs set union with respect to keys (and either assumes distinct keys or, where the same key is present in both, has a preference for one of the indexes). A *subtract* function is another binary function that performs set subtraction with respect to keys. An *iterate* procedure, in an imperative context, takes a procedure that accepts a key-value pair, and passes all the pairs to it in order.

Index algorithms

There is a huge number of data structures implementing an index. The following is a very partial list, and most have many variations:

Array	Association list	Radix tree	van Emde Boas tree
Skip list	Hash table	Binary search tree	Splay tree
AVL tree	Bloomier filter	Red-Black Tree	AA-tree
LLRB-tree	B-tree	B^+-tree	B^*-tree
Trie	Linked list		

Most of these data structures are simply modified *set* structures: the key-value pair simply replaces the set element.

Each approach has its own performance and memory characteristics. A good implementation of a *generic* index—one that implements the API we described, for any universe of keys with an ordering—requires $O(n)$ memory and guarantees $O(log(n))$ time, for each operation (where n is the number of keys in the index). However, some data structures are designed for specific key types that can be exploited for better performance characteristics.

Let's now run through the features of a few index data structures. We will then look at specifying a few more formally.

Array

For each possible key, we allocate memory representing whether that key is present, and if so what the associated value is.

This index scheme has a couple of advantages. The first is simplicity: it is undoubtedly the simplest index data structure and the algorithms upon it are similarly simple. The second is speed: search, insert and remove all have constant time complexity, as the location of the key in memory is a simple offset operation.

However, it has heavy disadvantages. There must be a unique map from keys to address locations. The most serious shortcoming is that memory use is proportional to the key domain size. Therefore, the domain of keys must be finite — eight-bit characters are OK, for example, but strings are not. Even where the domain is finite, it is usually too large for practical use: for example, the set of 32-bit integers. Most index usage patterns are *sparse*, making this data structure inefficient in the general case.

Sorted array

The data structure is a contiguous array of key-value pairs in the index sorted by the key. Essentially, this is the previous array structure with the missing keys elided.

This solves the disadvantages of the array of all possible keys: memory use is proportional to the number of stored keys.

However, it loses the speed advantage of the array of all possible keys. Search is no longer a simple offset; we must instead use binary search, which is logarithmic in the size of the array. Worse, insert and remove have to rebuild the entire array, and so are linear in the size of the array. This is impractical in the general case.

Association list

This is a linked list of key-value pairs. It removes the disadvantages of the array and the sorted array: memory use is proportional to the number of pairs, and mutation operations do not have to rebuild the entire data structure. It pays for this with a linear time search operation. The association list might be desirable for its simplicity, and where the number of elements is expected to be short.

Hash table

A hash table from key type K to value type V uses a *hash function* from K to a hash type H and a map from H to another index from K to V. The key-value pairs are therefore “bucketed” in lots of smaller indexes, which hopefully are of roughly equal size. Search, insert and remove are then a multi-step processes: find the hash of the key, get the index at that hash, then perform the operation on that index. This still leaves us with the decision of what hash function to use and what index data structure to use for each bucket.

This allows us to choose the H and storing more than one key at each location.

The hash table has disadvantages. The key type has to be hashable. In the worst case, all keys fall in the same bucket, in which case the hash table just adds extra memory and time. By *design*, the hash table has poor locality of reference: a good hash function spreads similar keys evenly over buckets. There is no obvious ordered iteration algorithm.

Binary search tree

The BST can be seen as an improvement of the linked list, arising from the observation that the nodes of an linked list can be generalized to contain any number of links. Following a link in a linked list only reduces the set being searched by one element; following one link out of a possible two, however, potentially halves the number of elements in the candidate set.

The division made at each node is into elements smaller than the element being looked at and elements larger than it.

The division that takes place at every step gives the BST a logarithmic complexity for all operations in the best-case. However, the worst-case scenario, in which one of the two sets at every node is the empty set, is equivalent to a linked list, and thus the BST still has linear complexity. This is by no means a corner case: data is often inserted in an "almost-sorted" order, creating an almost-linked-list.

Splay tree

One way to tackle the worst-case of the BST is simply to ensure it is a practical corner-case.

The Splay tree tries to make the worst-case less relevant by ensuring it does not happen repeatedly. It does this by, on every operation, restructuring the tree to move the latest-accessed element to the root. This does not change the complexity of the operations, and instead exploits locality of reference. The Splay tree is (relatively) simple: the data structure is the same as the BST, and the algorithms are the same except for the addition of the `splay` method.

Its disadvantages are significant though. Unlike other index data structures here, its `search` algorithm mutates the tree. The worst-case is still exhibited in common situations: for example, after searching for all elements in order. [2]

AVL tree

The AVL tree is a "balanced" tree. This means that it maintains invariants on the data structure that ensure it does not exhibit the BST worst case. This finally reduces all operations to logarithmic time.

Th e "height" of a tree is the longest path from the root to a leaf. At each node we store the "balance factor", which is the difference between the heights. The AVL tree ensures that $-1 \leq \textit{balance factor} \leq 1$. The AVL tree therefore requires extra two bits at each node. [3]

***K*-ary tree**

Th e *K*-ary tree is primarily motivated by a different problem with the BST: minimizing the number of storage reads. [4] For media with slow seek times and low granularity (such as a hard disk), the bottleneck is the seek time to the next read, and reading a single binary tree node is wasteful. The *K*-ary tree generalizes the binary tree to store K subtree pointers (the value K can then be optimized to the specific latency of the medium the index is stored on). Similarly, the algorithms on the *K*-ary tree are generalizations of the algorithms on the BST. A BST is then a *K*-ary tree where $K=2$. The tree then stores K-1 key-value pairs in one node. But it is infeasible to demand that each node holds this maximum amount, because then we could only store multiples of *K*-1 elements! Instead, the nodes are allowed to have *up to K* children.

B-tree

A B-tree is a perfectly balanced *K*-ary tree with the restriction that each node is between half-full and full (with the exception of the root node).

Its balancing strategy exploits the fact that, unlike nodes in the BST, those in the B-tree can be *split* and *merged*. The tree is always balanced because the height is only ever increased by splitting the root node (which increases the height along all paths to the leaves).

Indexes in the wild

The profusion of invented index data structures makes it difficult to decide on which to use and which to study. One metric we can use to make this decision is popularity: which data structures are actually in use in real environments? I have done a short survey of various language implementations' standard libraries to see which data structures they use.

GNU implementation of the C++ Standard Template Library	Red-Black trees for `std::set`, `std::multiset`, `std::map`, and `std::multimap`. [5]
Java Class Library's `TreeMap`	Red-Black tree [6]
Mono Project implementation of the .NET standard library's `SortedDictionary` and `SortedSet`	Red-Black tree [7]
Linux kernel	Red-Black tree for I/O scheduling and for virtual memory. It uses a pseudo-templating style in C, so there's only one implementation: `lib/rbtree.c`. In the same directory one will find `btree.c`, implementing a B+tree, and `radix-tree.c`, implementing a radix tree. [8]
ext3 filesystem directory entries	Red-Black tree
“CPython” implementation of Python	Hash [9]
Ruby	Hash
GHC implementation of Haskell `Data.Map`	Size-balanced binary tree. [10]

Red-Black Trees and the LLRB

It appears that most standard libraries and low-level codebases prefer the Red-Black Tree to the many alternatives. My decision to study a variant of Red-Black trees is motivated by their ubiquity.

What is the Red-Black Tree (RBT)? The RBT is a variant of the Binary Search Tree (BST). The BST is the starting point for many other data structures. In the case of the RBT, the modifications to the BST are motivated by the worst-case performance of the BST. Specifically, the RBT ensures that the tree is *balanced*, which allows the algorithms to perform binary search, which is the operation that makes it efficient.

Many other data structures do essentially the same thing: take the BST and augment it to ensure balance. What makes the Red-Black scheme more popular than the others? It *guarantees* balancedness, where some other data structures (such as splay trees) have the same worst case performance as the BST. It only stores a single bit of information per node on top of the BST data; many other data structures use more than this. It is old, being invented in 1972;[11] standard libraries are governed by inertia and a desire for battle-tested algorithms.

Despite being well-understood, the Red-Black Tree has a reputation for being tricky to implement. A main reason for this is the sheer number of cases one has to consider in most operations that mutate the tree. The unreasonable number of cases results partly from symmetry (each case has a symmetrical case which also has to be implemented) and partly from lax tree invariants that could be tightened.

In 2008, Robert Sedgwick introduced a version of the Red-Black Tree with additional invariants that considerably cut down on the number of cases. [12] This is the Left-Leaning Red-Black Tree. The operations on the LLRB are less complex than those on the RBT, yet retain their overall structure and spirit.

It is for that reason that I choose the LLRB to study in this project.

What is verification?

What does it mean to say such a program "has been verified"? Roughly, it means that the program is "correct", but the phrase is misleading because it omits to say that verification is always *with respect to a specification*. A specification is a description of what the program should do. To say that " program *P* has been verified with respect to specification *S*" simply means that *S* is a true description of *P*. The shorthand " program *P* has been verified" then means that there exists some *S* that *P* has been verified with respect to.

Informal reasoning

Most program specifications are written in English. For example, the standard UNIX tool `grep` is documented as

> `grep` searches the named input files (or standard input if no files are named, or if a single hyphen-minus (-) is given as file name) for lines containing a match to the given pattern. By default, `grep` prints the matching lines. [13]

Procedures can also be given specifications. For example, the procedure **int** *atoi*(**const int** * *nptr*) in the C standard library is described as converting " the initial portion of the string pointed to by *nptr* to **int** ." [14]

A specification can be more or less descriptive of the true behaviour. The above description of *atoi* could be more general, perhaps omitting to specify that it only uses the initial portion of the string. It could also be more precise, perhaps specifying what is meant by the "initial portion", and how exactly this maps to the integer that is returned. Again, this might be expressed in English, or more formally, perhaps using predicate logic.

Program "verification" could also be written in English, as a prose proof sketch, or just as comments in the program source. However, when we talk of "verified programs", we usually mean that the verification is a rigorous mathematical proof, and that is what I mean in this paper.

Type systems

One form of machine-assisted specification and verification is the C type system. For example, in the *atoi* procedure, the *nptr* parameter is typed as **const int** * . This, as a specification, says that the parameter is the size of a pointer into memory and that the memory at that address is not modified by the body of *atoi* . If the program compiles, this can be interpreted as a verification of that specification (this said, the C type system has many unsoundnesses, the most obvious being a deliberate unsoundness, that variables can be "cast" from one type to any other). Note that no explicit proof is given for that theorem: all theorems expressible in the C type system are simple enough that the compiler can produce proofs for them.

Other languages have "stronger" type systems than C. A large number of things we might wish to express about our program are not expressible in the C type system. A notoriously basic example is that we cannot express that a pointer is not equal to `NULL` (a "nothing" value that inhabits all pointer types). In the *atoi* example, a `NULL` input value will cause a runtime error, and yet programs using the procedure in this way are accepted by the compiler.

Most functional languages, such as Haskell, instead have algebraic types, meaning that nullability becomes part of the type and that it must be explicitly handled. Haskell's type system in particular places emphasis on another property for specification: purity. Roughly, a function in Haskell provably has no side-effects unless side-effectingness is expressed in the type. Other languages, such as Agda,[15] have yet stronger type systems that allow one to express full functional correctness using the type system.

VeriFast

VeriFast[16] provides the means for specification and verification. It is arguably closer to the inline-comment form of specification and verification than it is to a traditional type system. Both are written by the programmer in the same way: as special comments in the program source. The specification language is somewhat like a type system in that it places constraints on the expect input and output of procedures. The kinds of things it can specify, however, are much more complex. For example, neither the type system nor the documentation for *atoi* expressed how the returned **int** value relates to the initial substring of the parameter. An implementation of *atoi* that always returns zero would be incorrect, but would be accepted by the compiler without complaint. VeriFast offers the possibility to *fully* specify and verify *atoi* (with caveats that I discuss later). Verification, however, usually requires an explicit proof written by the programmer.

Technical background

This section provides a preparation for the core material of this report. I do not describe the LLRB here, as this in a sense is itself the topic of the report, and it is intertwined with the verification itself. Here I only provide the necessary background, historical and technical, to program verification. I start by describing *Hoare logic*, a simple formal system for reasoning about programs written in imperative languages like C. I then show that, on its own, Hoare logic has difficulties — in particular, it struggles to describe data on the heap. A proposed solution to this, a recent extension to Hoare logic called separation logic, is then explained. Finally, I give an overview of what VeriFast is and how it relates to those formal systems. The gory details of VeriFast, like the details of the LLRB, are left to the body of the report.

Hoare logic

Simple properties like non-nullability can be expressed in a conventional type system, but for the latter example we need something more. Beyond a certain level of complexity of the property, the compiler cannot automatically produce proofs for us — we need to write them ourselves. What would such a logic look like?

One logic for proving properties of programs written in imperative languages like C is *Hoare logic*. [17] I claim that Hoare logic is just a formalization of how programmers reason about their programs informally. Consider how a programmer might comment an implementation of a factorial procedure:

```
returns n * (n-1) * (n-2) * ... * 2 * 1
int factorial(int n)
{
    n >= 0

    int result = 1;
    int i = 0;

    while (i < n) {
        result == factorial(i)
        i++;
        result = i * result;
    }

    return result;
}
```

The commentary is an appeal for the correctness of the procedure, and the only difference between this and a full formalization in Hoare logic is that it leaves many things implicit. The programmer starts by saying that the procedure "returns n * (n-1) * (n-2) * ... * 2 * 1". This is an informal expansion of the function *factorial*(*n*) defined as *factorial*(0) == 0 and *factorial*(*n*) == *n* * *factorial*(*n*-1) . (Note that this *factorial* function is separate from the *factorial* procedure that is being implemented.)

This function is total because *n* is implicitly a natural number. The programmer notes this at the top of the procedure body. In Hoare terminology, this is called the *precondition*.

Two variables are then initialized. There is a relationship between them which is left implicit: $result == factorial(i)$. The naming of *result* is a comment: the implication is that this variable will be returned, so that by the end of the procedure, we must have $result == factorial(n)$. In Hoare terminology, this is called the *postcondition.*

There is only one other comment: at the top of the loop, $result == factorial(i)$. Here the programmer is making two claims: first, this assertion is true when we first enter the loop, and second, it is is true at the end of the loop body ready to enter it again. In Hoare terminology, this is called the *loop invariant*.

The first claim is easy to verify: it is the relationship between *result* and i after initialization. The second claim requires the use of the implicit definition of *factorial* .

How do we get from there to the postcondition? When the loop completes, the assertion still applies: $result == factorial(i)$. However, we want that $result == factorial(n)$. The reader concludes that $i == n$.

How is this justified? As we exit the loop, the loop test must no longer hold true; we therefore have that $i \geq n$. The programmer also knows that $i \leq n$, and this is an additional part of the loop invariant that was left implicit. From these two facts we can conclude $i == n$.

To be precise, this does not fully prove the procedure is correct. There is an additional implicit claim that the loop terminates at all; this claim is rarely made explicit because this is the desired behaviour for almost all loops. The intuitive reason that we eventually exit the loop is that eventually, "there is no more work to do". The programmer would say that i approaches and eventually reaches n .

If we denote "the amount of work left" by v , then in a properly terminating loop, v becomes smaller at each new loop iteration, and eventually reaches zero, at which point there is no more work to do, and we exit the loop. In the *factorial* procedure, the metric for "the amount of work left" is the distance between i and the goal n — i.e., $v = n\text{-}i$. This definition meets both requirements: each iteration increases i and so decreases $n\text{-}i$, and eventually i reaches n at which point $v = n\text{-}n = 0$.

In Hoare terminology, v is called the *loop variant*. Without the loop variant verification, we have proved what is called *partial correctness*, which just means that *if* the procedure terminates *then* the postcondition holds true. With the loop variant, we have proved *total correctness*, which means that the procedure eventually terminates at which point the postcondition holds true.

Notice, however, that we have said nothing about *how long* it takes to terminate. *Performance* is usually considered to be a separate concern to *correctness.* Total correctness is what we would normally think of as "correctness".

In short, Hoare logic boils down to a formalization of intuitive claims about standard control structures that enable us to prove the correctness of imperative programs.

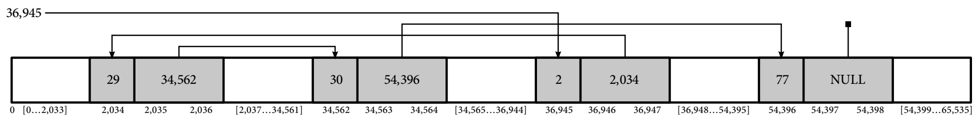
Separation logic

However, using Hoare logic in this basic form is impractical for most standard programs. Notice that in the factorial example, all variables were stored on the *stack*. Most real-world programs, however, make extensive use of the *heap* (and many languages store all values there). This is because the stack is limited to only holding fixed-sized non-recursive data structures. This means that something as simple as a linked list cannot be stored on the stack: we must allocate heap memory. To reason about this, we need to reason about the heap.

What is the heap? It is an index! Given an address of a memory cell, we either get back the data stored there or we get nothing because nothing is stored there (usually causing a "segmentation fault" and the termination of the program). This is simply a partial function.

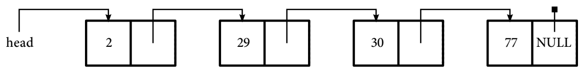
Let us take a linked list as an example of how to formally describe heap structures. The list consists of a number of chunks of heap memory, each of which contains one value. These chunks are at non-deterministic non-overlapping locations. This means we need to add the ability to access the chunks and give them an ordering. We do this by giving each node a second field which is a pointer to another node. Then, given one pointer, we can access one node and another pointer; and continue this process until we have exhausted the list.

Here is one example linked list:



We have four list nodes in a heap of 65,536 cells. Each node takes up three cells: one for the value, and two for the pointer.

As the chunk locations are non-deterministic, we cannot embed them in our reasoning about our programs. Instead we must abstract away the locations:



In doing so, there is a crucial property of the locations that we must maintain: they are distinct and non-overlapping. Using Hoare logic, this quickly becomes unwieldy. This is where *separation logic* enters with a notation that allows us to express these four nodes succinctly. [18]

The notation $head \mapsto 2$ says that at address *head* in the heap there is an allocated value 2. However, it says more than that. The expression $head \mapsto 2$ *stands for* a small heap with a single address *head* which maps to the value 2.

Now we wish to describe the *tail* pointer of the first node. What is its address? Using C notation to abstract away the specific lengths of fields, we say its address is *head->tail*. As chunk locations are non-deterministic, we do not know the value at that address. We can, however, say *there exists* a t_0 such that $head\text{->}tail \mapsto t_0$.

We then have two "mini-heaps", or "heap chunks", $head \mapsto 2$ and $head\text{->}tail \mapsto t_0$. We need to combine these to describe the whole list node. Separation logic provides the means for this by a binary operator, $*$, which is known as the *separating conjunction*. We can, for example, *separate* our two heap chunks as the expression $head \mapsto 2 * head\text{->}tail \mapsto t_0$. This does two things:

- asserts that the addresses of the two heap chunks are disjoint, and
- yields the (disjoint) union of those heap chunks.

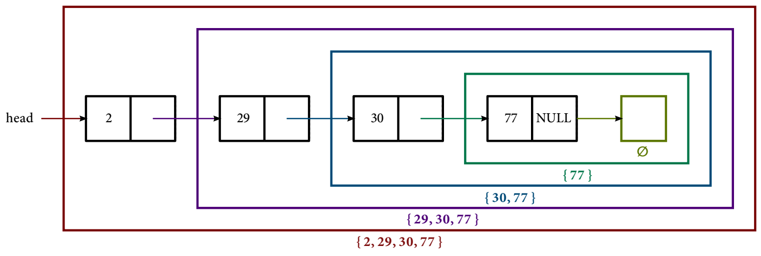
We can then proceed to describe the next node in the list in the same fashion: there exists t_1 such that $t_0 \mapsto 29 * t_0\text{->}tail \mapsto t_1$. This can then be separated from the first node: $head \mapsto 2 * head\text{->}tail \mapsto t_0 * t_0 \mapsto 29 * t_0\text{->}tail \mapsto t_1$. Notice that, as disjoint union is associative and commutative, we omit the parentheses when stringing together multiple heap chunks.

However, things are quickly getting out of hand. There is a clear pattern here. For a start, each node is conceptually a unit, and the only thing we care about is that it has some start address *head* and contains fields with values *v* and *tail* .

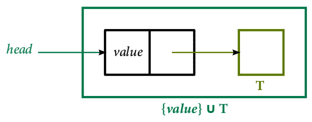
This is an ideal use for a *predicate*. Predicates abstract away details. We can define a predicate $ListNode(head, v, tail)$, and its definition is simple: $head \mapsto v \ast head\text{->}tail \mapsto tail$. The predicate is itself then a heap chunk. This means that we can write the above description of the first two nodes as $ListNode(head, 2, t_0) \ast ListNode(t_0, 29, t_1)$.

As we add more nodes, this list is going to get long. Notice that we don't really care about those existentials: t_0 , t_1 and so on. All we really care about is the *head* pointer and the values in the list [2, 29, 30, 77].

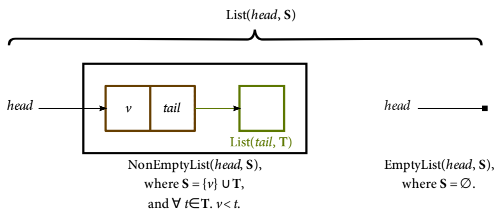
We really want a predicate $List(head, [2, 29, 30, 77])$. For this we first need to represent the list. We have a standard way of defining lists: inductively using nil and cons. The list [2, 29, 30, 77] is represented as 2:(29:(30:(77:nil))). Notice that the linked list has a similar substructure:



In the same way as we defined the list inductively, we want to define the linked list inductively. Each "cons" in the above diagram has the following structure:



Together with the "nil", the $List(head, L)$ predicate can then be represented visually as a disjunction:



This maps trivially to separation logic: $List(head, L_0) = (head == 0 \ast L_0 == Nil) \lor (there\ exists\ v, tail, L_1$. Notice that several conjuncts, such as $L_0 == v{:}L_1$, are separated using the separating conjunction, despite not representing a heap. This is a bit of notational sloppiness in which the assertion holds as well as representing the empty heap (denoted **emp** in separation logic).

VeriFast

VeriFast is a program verifier for programs written in the C programming language. [19] C is a low-level procedural language with manual memory management. A C program is a set of data type definitions and procedure definitions. One distinguished procedure called *main* is called at the beginning of the program.

VeriFast provides the means for much stronger formal specification and formal verification. VeriFast allows us to specify and verify individual C *procedures*. Verification of a procedure requires verification of the procedures it calls. When we say that the program is verified, we mean that its `main` procedure has a verified specification.

Both the specification and the verification are written manually as special comments mixed in with the C program source. Essentially, VeriFast is a pure function from this source with comments to either a success message or a failure message, indicating whether it can determine, with the aid of the manual proofs if necessary, that the program's procedures meet their specifications.

VeriFast's formal basis is separation logic. It aims to make program verification using separation logic a practical possibility. The chief way in which it makes this possible is by doing limited proof search on its own, so the more tedious parts of the proof remain implicit, just as they are in an informal proof.

I gradually introduce the details of VeriFast's specification language and verification language by way of example: each feature is introduced when motivated by a section of the LLRB case study.

An index specification

The index is an abstract data type, and the operations on it are abstract operations. That is, it does not reveal an internal representation. Instead, it is defined by what can be done with it. That is, it is defined as a partial function whose domain can be iterated.

Nevertheless, we need to choose an index representation as our specification. This is because we need to be able to say things like, " this tree represents an index that maps 4 to 5 and 7 to 13 . "

Mathematically, we might write this index as {4:5, 7:13} , which is basically sugar for a set of pairs {(4,5), (7,13)} where the keys (the first value of each pair) are unique.

The task is to represent this using the datatype system that VeriFast provides.

Datatypes

VeriFast's datatypes are traditional type-parameterized algebraic types as you might find in Haskell or ML. The basic tools of algebraic types are sum types and product types. A sum type represents a disjunction: a value of this type is a value of precisely one of the disjunct types. Unlike some language's sum types (like those of C), VeriFast's are *discriminated*, meaning that given a value of some sum type, we can determine which subtype it is. A (Cartesian) product type represents a conjunction: a value of this type contains values of each of the conjunct types, with a specified ordering. Type-parametricity means that the conjunct and disjunct types can instead be *type variables*, which we provide later in order to instantiate the type.

To introduce these concepts, let's define the basic datatypes necessary for our index. We have said that we can represent the index as a set of pairs, and so a basic data type I require is a "Pair". In VeriFast, we write:

inductive *Pair<K,V>* = *Pair*(*K*,*V*);

This says that

- *Pair* is a type constructor that takes two base types *K* and *V* and returns a base type *Pair<K,V>* . (This is type-parametricity.)
- The base type *Pair<K,V>* has a single data constructor (disjunct), called *Pair*, that takes two values (conjuncts) of types *K* and *V* and returns value of type *Pair<K,V>* . (This is a product type.)

Next, we require representation of a set. All algebraic values of a non-fixed size, like lists, trees, indexes, and sets, must be represented using induction. This means that one or more of the disjuncts or conjuncts is the same type as that which we are defining. For example, a non-empty list consists of a value at the head of a list followed by another list. All such inductive types in VeriFast must be finitely sized: at some point, we must get to the end of the list. Therefore at least one disjunct must be non-inductive: the list must have a *Nil* disjunct.

Mathematical sets are difficult to define and work with, because algebraic data demands a *sequencing*, where set elements have no defined order. For this reason, I choose to represent an index not as a *set* of pairs but as a *list* of pairs. The next datatype I need, then, is a list. In VeriFast, we write:

inductive *List<T>* =

```
| Nil
| Cons(T, List<T>);
```

Unlike *Pair*, *List* has two constructors: one for the empty list, which has no element, and another for non-empty lists, which consists of one element followed by a list of the rest.

The type of an index from domain *K* to range *V* can now be written as *List<Pair<K,V> >* . As an example, the values in the mathematical index { ("one", 1), ("two", 2), ("three", 3) } might be serialized as the list *Cons*(*Pair*("one", 1), *Cons*(*Pair*("two", 2), *Cons*(*Pair*("three", 3), *Nil*))) , of type *List<Pair<***string,int***> >* .

In an ideal world, VeriFast would then allow me to write:

typedef *Index<K,V> = List<Pair<K,V> >*;

However, it does not, and we are stuck with *List<Pair<K,V> >* .

Sorted lists

This does not fully capture the definition of an index, which requires that the keys be unique — the function is deterministic. This property cannot be captured using VeriFast's inductive datatypes. Instead, VeriFast provides two mechanisms that will allow us to enforce properties like this: predicates and fixpoint functions. I introduce predicates later, and here choose to use fixpoint functions. Ignore the "fixpoint" terminology: a fixpoint function is just a function operating over values of types defined in both the C and the VeriFast type system. It is a function in the mathematical sense: it has no "runtime" semantics, no evaluation order, no side-effects, and it is *total* in that it accepts all values in the domain and it terminates on all those inputs. It is functional in the "functional programming" sense: we cannot use loops or mutable variables, only recursion.

Uniqueness

For example, we can define a fixpoint function, *uniqueKeys* , that takes an index and returns whether the keys are unique:

```
fixpoint bool uniqueKeys<K,V>(List<Pair<K,V> > I0) {
    switch (I0) {
        case Nil: return true;
        case Cons(e0, I1):
            return switch (e0) {
                case Pair(k0,v0): return
                    uniqueKeys(I1) ∧
                    keyNotIn(k0, I1);
            };
    }
}
```

Let's step through this. *uniqueKeys* is of type *List<Pair<K,V> >* ; this is effectively a predicate.

The body of the function pattern-matches on the input list I_0 . Do not confuse the **switch** construct of the VeriFast annotation language with the **switch** construct of C: VeriFast's performs pattern-matching on inductive data types, and it is an expression and not a statement. Here we are using the VeriFast **switch** statement.

The **switch** has two cases for the two constructors of *List<Pair<K,V> >* . If I_0 is empty, the keys are unique: this is by definition. Otherwise, I_0 consists of an element e_0 followed by a list I_1 .

The element e_0 is a *Pair* , and it would be nice to pattern-match on it directly, but VeriFast does not currently support multi-level pattern-matching. Therefore we have to perform another **switch** on e_0 to gain access to the first key k_0 and its corresponding value v_0 .

At this point we can declare the conditions for which this non-empty list has unique keys. We have to define this inductively: the first key does not occur in the rest of the list, and the rest of the list is also unique. These two facts are expressed in VeriFast as two fixpoint calls: the first a recursive call to the fixpoint we are defining, and the second call to a (to-be-defined) fixpoint, *keyNotIn* .

Notice that the recursive call to *uniqueKeys* does not require the type arguments to be specified. VeriFast infers them from the types of the function arguments. (However we can provide the type arguments if we wish.)

Consider the first recursive call, to *uniqueKeys*(I_1) . To accept this recursive call, VeriFast must see that it terminates. In this case, VeriFast uses a simple reasoning: the recursive call is on a "smaller" argument, and because all datatypes are finite, it must at some point reach a base case. VeriFast sees that I_1 is smaller than I_0 because it the former is a structural component of the latter. We cannot manually tell VeriFast that the argument is smaller or in what sense it is smaller; there is no ability to define a termination-checking strategy in VeriFast. Termination-checking is in-built.

The second call is to *keyNotIn* , which is defined as follows:

```
fixpoint bool keyNotIn<K,V>(K k, List<Pair<K,V> > I₀) {
    switch (I₀) {
        case Nil: return true;
        case Cons(e₀,I₁):
            return switch (e₀) {
                case Pair(k₀,v₀): return
                    k ≠ k₀ ∧
                    keyNotIn(k, I₁);
            };
    }
}
```

The overall structure of *keyNotIn* is similar to that of *uniqueKeys* , and it is one we will see repeatedly: pattern-matching on the parameters to expose their structure before declaring the condition that defines the function. In this case, *keyNotIn* does not rely on further fixpoints, and so we are done.

Note that I presented these fixpoint definitions in a "top-down" order for didactic purposes. In code, VeriFast requires them to be presented in strictly bottom-up order. This prevents the definition of mutually recursive functions.

Sortedness

While uniqueness is sufficient for the definition of an index, I in fact use a stronger property: strict sortedness, meaning the keys are in strictly sorted order. There are multiple reasons for this:

- Sortedness enforces a unique representation of an index. Conceptually, the index {('a', 1), ('b', 2)} is the same as the index {('b', 2), ('a', 1)}. Yet represented as a list with the requirement of key uniqueness, there are multiple representations. This means we would have to define an equivalence relation which would make things ugly. Enforcing the sortedness of the keys enables a one-to-one relation between indexes and their list representations.
- In practice, the extra requirement of an ordering relation over keys is not a difficult one to satisfy. Even first-class functions could be ordered by their location in memory (though this would not of course respect any functional equivalence).
- The algorithms we will be verifying require an ordering relation, because comparison of and the sortedness of keys is necessary for the binary search technique that makes them efficient.

From here on, I specialize the type of the key to **int** in order to gain access to C's comparison operators. The sortedness of keys is defined in a similar manner to the uniqueness:

```
fixpoint bool lowerBound<V>(int lb, List<Pair<int,V> > I0) {
    switch (I0) {
        case Nil: return true;
        case Cons(e0, I1):
            return switch (e0) {
                case Pair(k0, v0): return
                    lb < k0 ∧
                    lowerBound(lb, I1);
            };
    }
}

fixpoint bool sorted<V>(List<Pair<int,V> > I0) {
    switch (I0) {
        case Nil: return true;
        case Cons(e0, I1):
            return switch (e0) {
                case Pair(k0, v0): return
                    lowerBound(k0, I1) ∧
                    sorted(I1);
            };
    }
}
```

Sortedness implies uniqueness

Notice that these two fixpoints are identical to the previous ones defining uniqueness, except for the use of the less-than operator in place of the not-equal-to operator. The proof that sortedness implies uniqueness turns on the simple fact that less-than implies not-equal-to, since it is a stronger assertion. We can use VeriFast to show that this is true.

We can take for granted that less-than implies not-equal-to, as this is something VeriFast can deduce. The proof proceeds in two parts. We first use this fact to prove that *lowerBound* implies *keyNotIn* . We then use this proof to prove that

```
fixpoint bool sorted<V>(List<Pair<int,V> > I0) {
    switch (I0) {
            case Nil: return true;
            case Cons(e_0, I_1):
                return switch (e_0) {
                    case Pair(k_0, v_0): return
                        lowerBound(k_0, I_1) ∧
                        sorted(I_1);
                };
        }
}
```

implies *uniqueKeys* .

Let us first prove that if some key *k* is a lower bound of an index then it is not in the index. More formally, we want to prove that

$$\forall\ \textbf{int}\ k\ ,\ List{<}Pair{<}\textbf{int},V{>}\ {>}\ I_0\ .\ lowerBound(k, I_0) \rightarrow keyNotIn(k, I_0)$$

The construct we will use to do this is called a *lemma function*. The theorem takes the form of a function contract written in the VeriFast assertion language, and its proof takes the form of a lemma function with that contract. Here is the above theorem stated in the VeriFast language as a lemma function declaration:

```
lemma void lowerBound_implies_keyNotIn<V>(int k, List<Pair<int,V> > I_0);
    requires lowerBound(k, I_0) == true;
    ensures keyNotIn(k, I_0) == true;
```

Th e **requires** keyword introduces the function precondition, and **ensures** introduces the postcondition. The clumsy syntax “ == **true** ” is necessary to distinguish the fixpoint function call from a predicate.

The body of the function, which constitutes the proof of the theorem, follows the contract:

```
lemma void lowerBound_implies_keyNotIn<V>(int k, List<Pair<int,V> > I_0)
    requires lowerBound(k, I_0) == true;
    ensures keyNotIn(k, I_0) == true;
{
    switch (I_0) {
        case Nil:
        case Cons(e_0, I_1):
            switch (e_0) {
                case Pair(k_0, v_0):
                    lowerBound_implies_keyNotIn(k, I_1);
            }
    }
}
```

VeriFast is doing a bit of work behind-the-scenes here in order to verify this proof, so let's step through it. The **switch** on the parameter I_0 means we are using proof-by-cases: we shall produce one proof for each of the constructors of I_0 .

Under the assumption that I_0 == *Nil* , we are required to prove that *keyNotIn*(*k*, *Nil*) . By expanding the definition of *keyNotIn* and taking the *Nil* Branch of the **switch** expression, VeriFast can see that this is true, and so we have no work to do in the *Nil* case.

Under the assumption that I_0 == *Cons*(e_0, I_1) , we then expand e_0 to *Pair*(k_0,v_0) . We have that *lowerBound*(*k*, *Cons*(*Pair*(k_0,v_0), I_1)) , and we are required to prove that *keyNotIn*(*k*, *Cons*(*Pair*(k_0,v_0), I_1)) . Then the pre- and post-conditions can be expanded by their definitions: we have that $k < k_0$ ∧ *lowerBound*(*k*, I_1) , and need to prove that $k \neq k_0$ ∧ *keyNotIn*(*k*, I_1) . VeriFast can see that $k < k_0$ implies $k \neq k_0$.

For the final piece of the proof, we perform a recursive call to *lowerBound_implies_keyNotIn* , which transforms our proof of *lowerBound*(*k*, I_1) into a proof that *keyNotIn*(*k*, I_1) , and this completes our postcondition. As with fixpoints, VeriFast requires that lemma functions terminate, and in this case accepts this by induction on derivation depth.

The proof that *sorted* implies *uniqueKeys* follows an outline that is becoming familiar:

```
lemma void sorted_implies_uniqueKeys<V>(List<Pair<int,V> > I0)
    requires sorted(I0) == true;
    ensures uniqueKeys(I0) == true;
{
    switch (I0) {
        case Nil:
        case Cons(e0, I1):
            switch (e0) {
                case Pair(k0, v0):
                    lowerBound_implies_keyNotIn(k0, I1);
                    sorted_implies_uniqueKeys(I1);
            }
    }
}
```

In the *Cons* case, we have that *sorted*(*Cons*(*Pair*(k_0, v_0), I_1)) and need to prove *uniqueKeys*(*Cons*(*Pair*(k_0, v_0), I_1)) . By expansion, we have *lowerBound*(k_0, I_1) ∧ *sorted*(I_1) , and need to prove *keyNotIn*(k_0, I_1) ∧ *uniqueKeys*(I_1) . We use *lowerBound_implies_keyNotIn* to prove from *lowerBound*(k_0, I_1) that *keyNotIn*(k_0, I_1) . Again, we recurse on the tail of the list to gain the equivalent proof for the tail of the list.

Search

In this chapter we will do our first verification of C in VeriFast. We do this for the first and most fundamental index operation: search. I introduce this gradually as follows. First, I describe search, informally and then mathematically. I move on to a definition within VeriFast, and discuss whether this definition is “correct”. Then we can write our first C and verify it against this specification! To warm up, I do this first with a linked list implementation. I continue with binary tree search, with both recursive and iterative implementations. We are then done with search, as the LLRB uses standard binary tree search as its search algorithm.

A search specification

As an index is a function, the most fundamental operation one can do is function application. In the context of indexes, which are partial functions, function application is called *search*: we lookup a key from the range to find a value from the domain. Having defined what an index is, we now need to define what it means to search that index. This, too, we do within VeriFast.

A search fixpoint

Because search is a *partial* function, there is the possibility of *failure*: the index might not return a value, because none exists. This kind of failure is described in functional languages with a *Maybe* , or *Option* , type, which is defined as

```
inductive Maybe<T> =
    | Nothing
    | Just(T);
```

This is similar to a nullable type in languages that have this mis-feature.

index_search is then a fixpoint that takes an index of type *List<Pair<***int***,V> >* , a key of type **int** to search for, and returns a *Maybe<V>* . That is, if the key exists in the index, it returns the value associated with the key as *Just(v)* , else the key does not exist in the index and it returns *Nothing* .

To search, we run through the list, find the first occurrence of the key, and then return the associated value:

```
fixpoint Maybe<V> index_search<K,V>(K needle, List<Pair<K,V> > I0) {
    switch (I0) {
        case Nil: return Nothing;
        case Cons(kv, I1):
            return switch (kv) {
                case Pair(k, v):
                    return k == needle ? Just(v) : index_search(needle, I1);
            };
    }
}
```

Notice that *index_search* does not have a contract: VeriFast does not allow contracts on fixpoint functions. In particular, we do not specify that it operates on a list with unique, or sorted, keys. Instead, we will specify that the list is sorted as a precondition whenever we use *index_search* .

So, is this fixpoint definition "correct"? Modelling an index as a set of pairs, a definition of index search might look like this:

```
search(k, I) = Just(v)   where (k,v) ∈ I
               Nothing otherwise.
```

However, we did not define our index using sets, and so this definition is not available to us. I actually offer no verification of search, and leave it to the reader's credulity. There is a basic reason for this: at some point, we have to appeal to definition, and for this project, the natural point for this is the definition of *index_search* . There are no properties of search that I can verify about this function without defining yet more fixpoints (e.g., the member-of relation) which will then have to somehow be "verified". There *are* provable properties of search in conjunction with other operations, like insertion and removal, but I introduce those with those operations and not here.

Representing algebraic data in C

Of course, the whole point of this exercise is to define a procedure in C — but C does not possess types like *Maybe<V>* . It does possess a **union** construct, but the union is not discriminated, and in any case is overkill for the task at hand. Therefore we have to decide on a representation of this type in C.

One way to do this is by using a boolean value and a value of the type *V* . The boolean does the discrimination — that is, it tells you which of the constructors is being used. The second value then only has an interpretation if the *Just* constructor is used. We can wrap this up in a predicate:

```
predicate IsMaybe<T>(bool is_just, T v, Maybe<T> m) =
    switch (m) {
        case Nothing: return is_just == false;
        case Just(v′): return is_just ∗ v == v′;
    };
```

How is this then used? The C procedure takes a pointer to allocated memory in which it should put the associated value if it finds the key. It then returns a boolean value that indicates whether the value at that location is the found key. The returned boolean and the value at the allocated memory together represent the maybe value.

A better representation

This has one significant disadvantage. The *IsMaybe* predicate does not fully capture the behaviour of the search algorithms that I will be writing and that exist in the wild. There, if the search key is not found, no change is made to the value memory. But the predicate does not state this: in the case that the returned value is false, the value at the memory location is non-deterministic. We want to prove as much as possible about the procedure as we can.

One way we can do this is with an *Either* type.

```
inductive Either<T1,T2> =
    | Left(T1)
    | Right(T2);

predicate IsEither<T>(bool is_right, T v1, T v2, Either<T,T> e) =
    switch (e) {
        case Left(v1'): return is_right == false * v1' == v1;
        case Right(v2'): return is_right * v2' == v2;
};
```

Using this predicate, the value at the value location is determined in both cases.

Searching linked lists

I will now introduce VeriFast's C verification abilities by writing a C procedure that implements *index_search* . I introduce the C data type for linked lists, and then a simple recursive procedure operating on that data type, which verifies with ease.

A linked list

A linked list is a data structure used to represent a sequence; a sequence of *n* elements is represented in memory by *n* chunks of memory, called *nodes*, where each node represents one element. In our case, this means that each node contains one key-value pair; i.e., it has a *key* field and a *value* field.

We have seen linked lists earlier in this report when introducing separation logic; and now I wish to define it in C and show how VeriFast uses separation logic to place the proper constraints on values of the linked list type.

In C, we declare the node type as follows. Note that we embed the key and value directly in the node. We could have declared another datatype for key-value pairs, but as C does not allow us to write a generic linked list type, there is nothing to be gained from this.

```
struct ListNode;
typedef struct ListNode ListNode;

struct ListNode {
        int key;
        int value;
        ListNode * tail;
};
```

We will be working with pointers to *ListNode* s. We frequently want to say that " such-and-such a pointer points to an allocated *ListNode* that contains some *key* , some *value* , and some *tail* pointer " . VeriFast allows us to say this succinctly with a **predicate** :

```
predicate IsListNode(ListNode * node; int key, int value, ListNode * tail) =
        node->key ↦ key *
        node->value ↦ value *
        node->tail ↦ tail *
        malloc_block_ListNode(node)
        ;
```

After the introduction of separation logic, most of this should look familiar. The predicate *IsListNode*(*node*, *key*, *value*, *tail*) *stands for* a heap chunk in which those values are present; That is, it can be seen as a finite map from heap addresses to their values, where that map satisfies the body of the predicate.

The body of the *IsListNode* heap chunk is itself composed of four heap chunks. The first three are the familiar "points to" chunks I have introduced. These are each one heap chunk containing one memory cell. (VeriFast abstracts away the length of fields in memory.) Thus the expression *node->key* $\mapsto$ *key* is a heap chunk containing one memory cell with address *node->key* and value *key* .

The final predicate call of *IsListNode* is *malloc_block_ListNode*(*node*) . This is not a heap chunk. Rather, it represents the fact that *node* is a pointer to memory that has been allocated by a call to *malloc* , and thus that it can be deallocated with a call to *free*(*node*) . The predicate *malloc_block_ListNode* is automatically created by VeriFast when it sees the **struct** *ListNode* declaration. The malloc predicate is threaded through the program to a point at which the memory is freed using the C procedure *free*() . This prevents accidentally "double-freeing" allocated memory, an operation that invokes undefined behaviour.

Neither our predicate nor the C **struct** type fully specifies a linked list. Our **struct** *ListNode* s could be used to construct arbitrary directed graphs. The *tail* pointers could point to memory outside the graph. We do not yet have a guarantee that the *tail* pointers string together a *list* of nodes ending with a *null* pointer.

I write a predicate *IsList* which asserts that a chunk of memory pointed into by some *ListNode* pointer *head* represents an algebraic list *L* .

Just as the *IsListNode* heap chunk is composed of smaller heap chunks, multiple *IsListNode* s can be composed to form an *IsList* . We can in fact split up this composition further: multiple nodes can be composed to form a list *segment*, and multiple list segments can be composed to form a list.

A list segment is essentially a linked list terminated with a specific pointer instead of a null pointer. That is, it is a linked list with a pointer going *into* the list and a pointer coming *out* of the list. We can summarize this with a new predicate, *IsListSeg*(*in*, *out*, *L*) , which is a heap chunk that represents the list *L* , has a pointer into the heap chunk *in* , and has a pointer *out* from the last node. That is, it should look like this:

```
predicate IsListSeg(ListNode * in, ListNode * out; List<Pair<int,int> > L);
```

Just as we defined the algebraic *List* recursively ("inductively"), so we define *IsListSeg* recursively. Either the *in* pointer is equal to the *out* pointer, or it is not. If they are equal, the list *L* is empty. Otherwise, *in* points to an allocated node, and we can express this as *IsListNode*(*in*, ?*key*, ?*value*, ?*tail*) . The first element in the list *L* is then *Pair*(*key*, *value*) . Call the rest of the list L_1 . Here comes the recursive part: the node's *tail* pointer is a pointer into a list segment representing the rest of the list L_1 . The tail of the list segment has the same *out* pointer as the entire list segment. We express this as *IsListSeg*(*tail*, *out*, ?L_1) . This gives us out entire definition for *IsListSeg* :

```
predicate IsListSeg(ListNode * in, ListNode * out; List<Pair<int,int> > L) =
    in == out ?
        L == Nil
    :
        IsListNode(in, ?key, ?value, ?tail) ∗
        IsListSeg(tail, out, ?L1) ∗
        L == Cons(Pair(key, value), L1)
    ;
```

It is crucial to understand the role of the separating conjunction in making this definition correct. That is, the predicate has two parameters, *head* and *L* , and the fact is represented as *IsList*(*head*, *L*) .

```
predicate IsList(ListNode * head; List<Pair<int,int> > L) = IsListSeg(head, 0, L);
```

A linked list search procedure

We will write a C procedure, *list_search* , which in some sense implements the fixpoint function *index_search* .

```
bool list_search(int needle, int * value, ListNode * listNode);
requires IsList(listNode, ?L) ∗ integer(value, ?v0);
ensures IsList(listNode, L) ∗ integer(value, ?v1) ∗ IsMaybe(result, v1, index_search(needle, L));
```

Notice that the procedure does not require *sorted*(*L*) . Its job is just to implement the fixpoint function.

The first thing you may notice is that procedure contracts look the same as lemma theorems. There is a fundamental reason for this: the Curry-Howard isomorphism, which says that types can be interpreted as theorems, of which their inhabitants are proofs.

The body of *list_search* follows.

```
bool list_search(int needle, int * value, ListNode * listNode)
requires IsList(listNode, ?I) ∗ integer(value, ?v0);
ensures IsList(listNode, I) ∗ integer(value, ?v1) ∗ IsMaybe(result, v1, index_search(needle, I));
{
    open IsList(listNode, I);
    open IsListSeg(listNode, 0, I);

    if (listNode == 0) {
        close IsMaybe(false, v0, index_search(needle, I));
        return false;
    }
    else {
        if (listNode->key == needle) {
            *value = listNode->value;
            close IsMaybe(true, listNode->value, index_search(needle, I));
            return true;
        }
        else {
            return list_search(needle, value, listNode->tail);
        }
    }
}
```

Searching trees

Having warmed up with linked lists, I now move on to something resembling the LLRB which is the ultimate goal of this project. This something is the Binary Search Tree (BST), and in fact, searching a BST is the same as searching an LLRB. The BST is a standard data structure that improves on the linked list in the sense that its best-case performance is better. I first define the binary tree and binary search in VeriFast, and show how this meets our index search specification. Then, just as we did for the linked list, we implement this in C. I demonstrate two implementations of the same procedure contract. The first is a recursive one, which is easier to verify, but somewhat inefficient at runtime. The second is an iterative implementation, which is more efficient at runtime but harder to verify.

The Tree datatype

I now introduce another inductive datatype: the binary search tree (BST). This is the basis for an efficient implementation of search.

Algebraic trees

The binary tree is an inductive data structure. A tree is a leaf or a branch. A leaf value is similar to the nil value for lists: it is the "bottom" of the structure, and contains no data. A branch is similar to a cons in a list, except it contains *two* subtrees where the list contains just one sublist.

My *Tree* definition differs from this description in two ways. The first difference is that, rather than embed a *Pair*<**int**,*V*> value at each branch, I embed the key and value directly. This is a pragmatic measure motivated by VeriFast's current lack of multi-level pattern-matching. The second difference is that each branch also contains another value, called its "color", which is either "red" or "black". This value is used later when we discuss the LLRB and we shall not discuss it further here.

Again, the tree structure can be defined as a VeriFast inductive datatype:

```
inductive Tree<K,V> =
    | Leaf
    | Branch(Tree<K,V>, K, V, Color, Tree<K,V>);
```

How does this tree represent an index? A tree can be "flattened" to a list of its nodes by an in-order traversal. We define a fixpoint that takes a tree to its list of key-value pairs:

```
fixpoint List<Pair<K,V> > flatten<K,V>(Tree<K,V> t) {
    switch (t) {
        case Leaf: return Nil;
        case Branch(l, k, v, c, r): return append(flatten(l), Cons(Pair(k,v), flatten(r)));
    }
}
```

The keys in the tree are "in-order": all keys in the left subtree are less than key at the root, which is less than all the keys in the right subtree. As we will see, to say this about a tree T, it is enough to say *sorted*(*flatten*(T)) .

VeriFast does not support assertions that a variable is equal to some pattern. Therefore I use predicates that express that a variable uses such-and-such constructor on such-and-such data. There is one predicate per data constructor. For the above datatypes, we have:

```
predicate Cons<T>(List<T> $l_0$, T $e_0$, List<T> $l_1$) =
    switch ($l_0$) {
        case Nil: return false;
        case Cons($e_0'$, $l_1'$): return $e_0$ == $e_0'$ * $l_1$ == $l_1'$;
    };

predicate Pair<K,V>(Pair<K,V> p, K k, V v) =
    switch (p) {
        case Pair(k′, v′):
            return k == k′ ∗ v == v′;
    };
```

As previously, I define a predicate that expresses that a tree is a particular constructor:

```
predicate Branch<K,V>(Tree<K,V> t, Tree<K,V> l, K k, V v, Color c, Tree<K,V> r) =
    switch (t) {
        case Leaf: return false;
        case Branch(l0, k0, v0, c0, r0):
            return
                l == l0 ∗
                k == k0 ∗
                v == v0 ∗
                c == c0 ∗
                r == r0;
    };
```

Tree representation in C

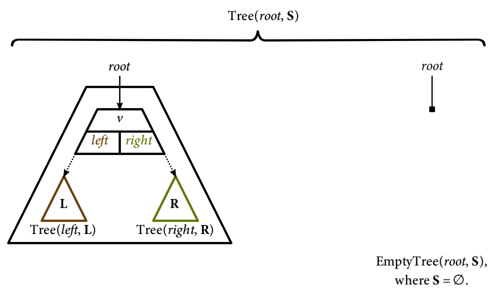
In order to implement anything in C, we first need a C data type. The following will look very similar to the linked list:

```
struct TreeNode;
typedef struct TreeNode * TreeNodePtr;

struct TreeNode {
    int key;
    int value;
    Color color;
    TreeNodePtr lft;
    TreeNodePtr rgt;
};
```

Again, for the same reasons, the C data type is not strong enough: we need to express that it *must be used to represent a tree* and not some other graph-like or invalid structure. Just as we defined *IsList* for the inductive *List* type, we now define *IsTree* for the inductive *Tree* type:

```
predicate IsTree(TreeNodePtr root; Tree<int,int> tree) =
    root == 0 ?
        tree == Leaf
    :
        root->key ↦ ?key ∗
        root->value ↦ ?value ∗
        root->color ↦ ?color ∗
        root->lft ↦ ?lft ∗
        root->rgt ↦ ?rgt ∗
        malloc_block_TreeNode(root) ∗
        IsTree(lft, ?lftTree) ∗
        IsTree(rgt, ?rgtTree) ∗
        tree == Branch(lftTree, key, value, color, rgtTree)
    ;
```

A visual representation of this predicate should help:



BST search specification

Search on an LLRB is just binary search: it works on any arbitrarily imbalanced tree, and so does not require the LLRB invariants for the algorithm to work.

Binary search can be written as another fixpoint. This means I can write our verification of LLRB search in two steps:

- Show that binary search is equivalent to list search;
- Show that our C algorithm performs binary search.

BSTs are sorted

The correctness of binary search on trees relies on the keys in the tree being sorted. In tree terminology, the tree is a BST. Now, we could simply use *sorted*(*flatten*(*t*)) to assert that a tree is a BST. However, this is rather unnatural, and so we wish to define it directly. Just as we defined *sorted* on lists, I now define *bst* on trees.

```
fixpoint bool tree_lowerBound<V>(int lb, Tree<int,V> t)
{
    switch (t) {
        case Leaf: return true;
        case Branch(l, k, v, c, r): return
            tree_lowerBound(lb, l) ∧
            tree_lowerBound(lb, r) ∧
            lb < k;
    }
}

fixpoint bool tree_higherBound<V>(Tree<int,V> t, int hb)
{
    switch (t) {
        case Leaf: return true;
        case Branch(l, k, v, c, r): return
            tree_higherBound(l, hb) ∧
            tree_higherBound(r, hb) ∧
            k < hb;
    }
}

fixpoint bool bst<V>(Tree<int,V> t)
{
    switch (t) {
        case Leaf: return true;
        case Branch(l, k, v, c, r):
            return
                tree_higherBound(l, k) ∧
                tree_lowerBound(k, r) ∧
                bst(l) ∧ bst(r);
    }
}
```

We now have a proof obligation that *bst*(*t*) implies *sorted*(*flatten*(*t*)) . The lemma for this follows. Its body is a little intimidating. We have two inductive calls to the lemma for each of the subtrees. With two sorted lists, how can we show that the two lists appended with the root yield a sorted list? This depends on the two lists being strictly "less than" and "greater than" the root key. This is true, and we have two more lemma calls to demonstrate this. Finally we have the lemma call that shows that, because the left and right sublists are sorted, and the left, root and right elements are in the right order, then the whole thing appended is sorted.

```
lemma void bst_is_sorted<V>(Tree<int,V> t)
    requires bst(t) == true;
    ensures sorted(flatten(t)) == true;
{
    switch (t) {
        case Leaf:
        case Branch(l, k, v, c, r):
            bst_is_sorted(l);
            bst_is_sorted(r);
            tree_higherBound_is_list_higherBound(l, k);
            tree_lowerBound_is_list_lowerBound(k, r);
            sorted_sides_sorted_whole(flatten(l), k, v, flatten(r));
    }
}
```

The first two lemmas are establish that tree bounds are equivalent to list bounds.

```
fixpoint bool higherBound<V>(List<Pair<int,V> > l0, int hb) {
    switch (l0) {
        case Nil: return true;
        case Cons(kv, l1):
            return switch (kv) {
                case Pair(k, v): return k < hb ∧ higherBound(l1, hb);
            };
    }
}
```

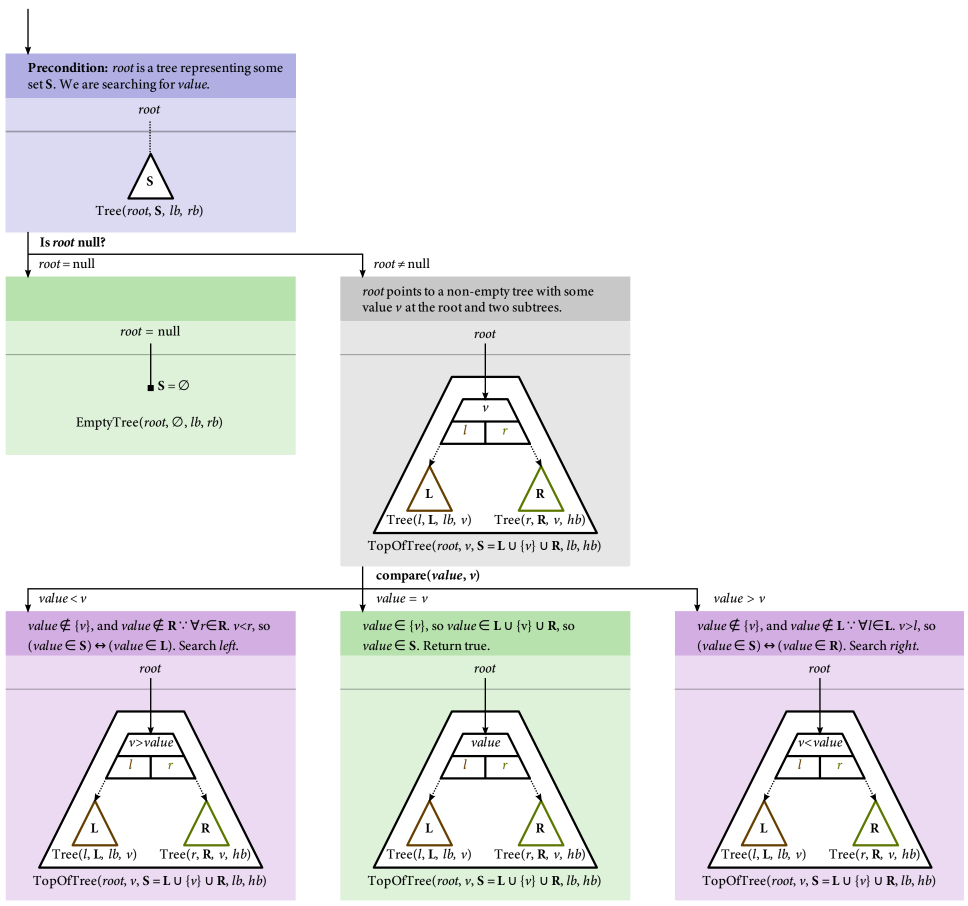
```
lemma void tree_higherBound_is_list_higherBound<V>(Tree<int,V> t, int hb)
    requires tree_higherBound(t, hb) == true;
    ensures higherBound(flatten(t), hb) == true;
{
    switch (t) {
        case Leaf:
        case Branch(l, k, v, c, r):
            tree_higherBound_is_list_higherBound(l, hb);
            tree_higherBound_is_list_higherBound(r, hb);
            higherBound_sides_is_higherBound_whole(flatten(l), k, v, flatten(r), hb);
    }
}

lemma void tree_lowerBound_is_list_lowerBound<V>(int lb, Tree<int,V> t)
    requires tree_lowerBound(lb, t) == true;
    ensures lowerBound(lb, flatten(t)) == true;
{
    switch (t) {
        case Leaf:
        case Branch(l, k, v, c, r):
            tree_lowerBound_is_list_lowerBound(lb, l);
            tree_lowerBound_is_list_lowerBound(lb, r);
            lowerBound_sides_is_lowerBound_whole(lb, flatten(l), k, v, flatten(r));
    }
}


lemma void sorted_sides_sorted_whole<V>(List<Pair<int,V> > L0, int k, V v, List<Pair<int,V> >
r)
    requires sorted(L0) == true ∧ sorted(r) == true ∧ higherBound(L0, k) == true ∧
    lowerBound(k, r) == true;
    ensures sorted(append(L0, Cons(Pair(k,v), r))) == true;
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch (l0) {
                case Pair(k0,v0):
                    sorted_sides_sorted_whole(L1, k, v, r);
                    loosen_lowerBound(k0, k, r);
                    lowerBound_sides_is_lowerBound_whole(k0, L1, k, v, r);
            }
    }
}

lemma void loosen_lowerBound<V>(int lt, int lb, List<Pair<int,V> > S)
    requires lt ≤ lb ∗ lowerBound(lb, S) == true;
    ensures lowerBound(lt, S) == true;
{
    switch (S) {
        case Nil:
        case Cons(kv, l1):
            switch (kv) {
                case Pair(k,v): loosen_lowerBound(lt, lb, I1);
            }
    }
}

lemma void higherBound_sides_is_higherBound_whole<V>(List<Pair<int,V> > L0, int k, V v,
List<Pair<int,V> > r, int hb)
    requires higherBound(L0, hb) == true ∧ higherBound(r, hb) == true ∧ k < hb;
    ensures higherBound(append(L0, Cons(Pair(k,v), r)), hb) == true;
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch (l0) {
                case Pair(k0, v0): higherBound_sides_is_higherBound_whole(L1, k, v, r, hb);
            }
    }
}

lemma void lowerBound_sides_is_lowerBound_whole<V>(int lb, List<Pair<int,V> > L0, int k, V v,
List<Pair<int,V> > r)
    requires lowerBound(lb, L0) == true ∗ lowerBound(lb, r) == true ∗ lb < k;
    ensures lowerBound(lb, append(L0, Cons(Pair(k,v), r))) == true;
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch (l0) {
                case Pair(k0,v0): lowerBound_sides_is_lowerBound_whole(lb, L1, k, v, r);
            }
    }
}
```

Tree search is list search

Our definition of BST search is exactly how it would be defined in a functional language:

```
fixpoint Maybe<V> tree_search<V>(int needle, Tree<int,V> t)
{
    switch (t) {
        case Leaf: return Nothing;
        case Branch(l, k, v, c, r):
            return
                needle < k ?
                    tree_search(needle, l)
                : k < needle ?
                    tree_search(needle, r)
                :
                    Just(v);
    }
```

}


Precondition: root is a tree representing some set S. We are searching for value.
root
S
Tree(root, S, lb, rb)
Is root null?
root = null
root ≠ null
root = null
S = ∅
EmptyTree(root, ∅, lb, rb)
root points to a non-empty tree with some value v at the root and two subtrees.
root
v
l
r
L
R
Tree(l, L, lb, v)
Tree(r, R, v, hb)
TopOfTree(root, v, S = L ∪ {v} ∪ R, lb, hb)
compare(value, v)
value < v
value = v
value > v
value ∉ {v}, and value ∉ R ∵ ∀r∈R. v<r, so (value ∈ S) ↔ (value ∈ L). Search left.
value ∈ {v}, so value ∈ L ∪ {v} ∪ R, so value ∈ S. Return true.
value ∉ {v}, and value ∉ L ∵ ∀l∈L. v>l, so (value ∈ S) ↔ (value ∈ R). Search right.
root
v>value
value
v<value
l
r
L
R
Tree(l, L, lb, v)
Tree(r, R, v, hb)
TopOfTree(root, v, S = L ∪ {v} ∪ R, lb, hb)

We can now formalize what we want to prove:

```
lemma void tree_search_is_list_search<V>(int needle, Tree<int,V> t)
    requires bst(t) == true;
    ensures tree_search(needle, t) == index_search(needle, flatten(t));
{
    switch (t) {
        case Leaf:
        case Branch(l, k, v, c, r):
            bst_is_sorted(t);
            sides_of_sorted_are_sorted(flatten(l), Cons(Pair(k, v), flatten(r)));

            if (needle < k) {
                lt_lb_not_mem(needle, k, flatten(r));
                not_in_right_list_look_in_left_list(needle, flatten(l), Cons(Pair(k,v), flatten(r)));
                tree_search_is_list_search(needle, l);
            }
            else if (k < needle) {
                left_side_of_sorted_is_bound(flatten(l), k, v, flatten(r));
                gt_hb_not_mem(flatten(l), k, needle);
                not_in_left_list_look_in_right_list(needle, flatten(l), Cons(Pair(k,v), flatten(r)));
                tree_search_is_list_search(needle, r);
            }
            else {
                in_right_in_append(k, v, flatten(l), Cons(Pair(k,v), flatten(r)));
            }
    }
}
```

Getting there requires some substantial proofs on the properties of lists:

```
lemma void sides_of_sorted_are_sorted<V>(List<Pair<int,V> > L_0, List<Pair<int,V> > R_0)
    requires sorted(append(L_0, R_0)) == true;
    ensures sorted(L_0) == true ∧ sorted(R_0) == true;
{
    switch (L_0) {
        case Nil:
        case Cons(l_0, L_1):
            switch (l_0) {
                case Pair(k_0, v_0):
                    leftmost_is_lower_bound_of_left_list(k_0, v_0, L_1, R_0);
                    sides_of_sorted_are_sorted(L_1, R_0);
            }
    }
}

lemma void lt_lb_not_mem<V>(int lt_lb, int lb, List<Pair<int,V> > I_0)
    requires lt_lb < lb ∧ lowerBound(lb, I_0);
    ensures index_search(lt_lb, I_0) == Nothing;
{
    loosen_lowerBound(lt_lb, lb, I_0);
    lb_not_mem(lt_lb, I_0);
}

lemma void not_in_right_list_look_in_left_list<V>(
        int needle,
        List<Pair<int,V> > L_0,
        List<Pair<int,V> > R_0
        )
    requires index_search(needle, R_0) == Nothing;
    ensures index_search(needle, L_0) == index_search(needle, append(L_0, R_0));
{
    switch (L_0) {
        case Nil:
        case Cons(l_0, L_1):
            switch (l_0) {
                case Pair(k_0, v_0):
                    if (k_0 ≠ needle) not_in_right_list_look_in_left_list(needle, L_1, R_0);
            }
    }
}
```

```
lemma void left_side_of_sorted_is_bound<V>(List<Pair<int,V> > L0, int k, V v, List<Pair<int,V>
> R0)
    requires sorted(append(L0, Cons(Pair(k,v), R0))) == true;
    ensures higherBound(L0, k) == true;
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch (l0) {
                case Pair(k0, v0):
                    leftmost_less_than_other_member(k0, v0, L1, k, v, R0);
                    left_side_of_sorted_is_bound(L1, k, v, R0);
            }
    }
}

lemma void gt_hb_not_mem<V>(List<Pair<int,V> > I0, int hb, int gt_hb)
    requires hb < gt_hb ∧ higherBound(I0, hb);
    ensures index_search(gt_hb, I0) == Nothing;
{
    loosen_higherBound(I0, hb, gt_hb);
    hb_not_mem(I0, gt_hb);
}

lemma void hb_not_mem<V>(List<Pair<int,V> > I0, int hb)
    requires higherBound(I0, hb) == true;
    ensures index_search(hb, I0) == Nothing;
{
    switch (I0) {
        case Nil:
        case Cons(e0, I1):
            switch (e0) {
                case Pair(k0, v0): hb_not_mem(I1, hb);
            }
    }
}

lemma void not_in_left_list_look_in_right_list<V>(
        int needle,
        List<Pair<int,V> > L0,
        List<Pair<int,V> > R0
        )
    requires index_search(needle, L0) == Nothing;
    ensures index_search(needle, R0) == index_search(needle, append(L0, R0));
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch (l0) {
                    case Pair(k_0, v_0): not_in_left_list_look_in_right_list(needle, L_1, R_0);
                }
        }
}

lemma void in_right_in_append<V>(int needle, V v, List<Pair<int,V> > L_0, List<Pair<int,V> > 
R_0)
        requires sorted(append(L_0, R_0)) == true ∧ index_search(needle, R_0) == Just(v);
        ensures index_search(needle, append(L_0, R_0)) == Just(v);
{
        switch (L_0) {
                case Nil:
                case Cons(l_0, L_1):
                        in_right_list_hb_left_list(needle, v, L_0, R_0);
                        hb_not_mem(L_0, needle);
                        not_in_left_list_look_in_right_list(needle, L_0, R_0);
        }
}
```

Recursive BST search in C

Now we have defined a binary search fixpoint, we can do the same as we did with linked list search: write a procedure that satisfies that fixpoint over C data types.

A recursive search procedure

The following C algorithm should be strongly reminiscent of the fixpoint, and highly familiar to a basic C programmer:

```
bool llrb_search_recursive(int needle, int * value, TreeNodePtr root) {
        if (root == 0) {
                return false;
        }
        else {
                int key = root->key;
                bool result;
                if (needle < key) {
                        result = llrb_search_recursive(needle, value, root->lft);
                }
                else if (key < needle) {
                        result = llrb_search_recursive(needle, value, root->rgt);
                }
                else {
                        result = true;
                        *value = root->value;
                }
                return result;
        }
}
```

The first thing this needs is a contract. In the case of *llrb_search* , I identify three key conditions necessary for *llrb_search* to work properly:

- The *TreePtr root* parameter points into a binary-tree-shaped structure (*e.g.*, the structure does not contain loops).
- The tree structure is a BST (*i.e.*, the keys are ordered).
- The **int** * *value* parameter points to a valid integer in memory (*i.e.*, the pointer is not dangling).

Here is the contract for *llrb_search* . Notice that it is extremely similar in structure to that for *list_search* ; this is a good thing, because it is indicative of our specification being implementable by a wide range of data structures.

```
bool llrb_search(int needle, int * value, TreeNodePtr root);
requires IsTree(root, ?t) ∗ integer(value, ?v0) ∗ bst(t) == true;
ensures IsTree(root, t) ∗ integer(value, ?v1) ∗ IsMaybe(result, v1, tree_search(needle, t));


bool llrb_search_recursive(int needle, int * value, TreeNodePtr root)
requires IsTree(root, ?t) ∗ integer(value, ?v0) ∗ bst(t) == true;
ensures IsTree(root, t) ∗ integer(value, ?v1) ∗ IsMaybe(result, v1, tree_search(needle, t));
{
    open IsTree(root, t);
    if (root == 0) {
        close IsTree(root, t);
        close IsMaybe(false, v0, tree_search(needle, Leaf));
        return false;
    }
    else {
        assert root->lft ↦ ?lft ∗ IsTree(lft, ?l) ∗ root->rgt ↦ ?rgt ∗ IsTree(rgt, ?r);
        int key = root->key;
        bool result;
        if (needle < key) {
            result = llrb_search_recursive(needle, value, root->lft);
            open IsMaybe(result, ?v1, tree_search(needle, l));
            close IsMaybe(result, v1, tree_search(needle, t));
        }
        else if (key < needle) {
            result = llrb_search_recursive(needle, value, root->rgt);
            open IsMaybe(result, ?v1, tree_search(needle, r));
            close IsMaybe(result, v1, tree_search(needle, t));
        }
        else {
            result = true;
            *value = root->value;
            close IsMaybe(result, root->value, tree_search(needle, t));
        }
        close IsTree(root, t);
        return result;
    }
}
```

Iterative BST search in C

```
inductive Ctx<K,V> =
    | NilCtx
    | LftCtx(Tree<K,V>, K, V, Color, Ctx<K,V>)
    | RgtCtx(K, V, Color, Tree<K,V>, Ctx<K,V>);

fixpoint Tree<K,V> ctx_compose<K,V>(Ctx<K,V> C_0, Tree<K,V> T_0) {
    switch (C_0) {
        case NilCtx: return T_0;
        case LftCtx(l, k, v, c, pctx):
            return ctx_compose(pctx, Branch(l, k, v, c, T_0));
        case RgtCtx(k, v, c, r, pctx):
            return ctx_compose(pctx, Branch(T_0, k, v, c, r));
    }
}

lemma void sorted_tree_sorted_subtree<V>(Tree<int,V> t, Ctx<int,V> ctx, Tree<int,V> subt)
    requires sorted(flatten(t)) == true ∗ ctx_compose(ctx, subt) == t;
    ensures sorted(flatten(subt)) == true;
{
    switch (ctx) {
        case NilCtx:
        case LftCtx(l, k, v, c, pctx):
            sorted_tree_sorted_subtree(t, pctx, Branch(l, k, v, c, subt));
            sides_of_sorted_are_sorted(flatten(l), Cons(Pair(k,v), flatten(subt)));
        case RgtCtx(k, v, c, r, pctx):
            sorted_tree_sorted_subtree(t, pctx, Branch(subt, k, v, c, r));
            sides_of_sorted_are_sorted(flatten(subt), Cons(Pair(k,v), flatten(r)));
    }
}

predicate IsCtx(TreeNodePtr root, TreeNodePtr tree, Ctx<int,int> ctx)
    = switch (ctx) {
            case NilCtx: return root == tree;
            case LftCtx(lft, k, v, c, pctx):
                return IsTreeNode(?node, ?lftptr, k, v, c, tree)
                    ∗ IsTree(lftptr, lft)
                    ∗ IsCtx(root, node, pctx)
                    ;
            case RgtCtx(k, v, c, rgt, pctx):
                return IsTreeNode(?node, tree, k, v, c, ?rgtptr)
                    ∗ IsTree(rgtptr, rgt)
                    ∗ IsCtx(root, node, pctx)
                    ;
        };

bool llrb_search(int needle, int * value, TreeNodePtr root)
{
    TreeNodePtr i = root;

    while (i ≠ 0) {
        int k = i->key;
        if (needle < k) {
            i = i->lft;
        }
        else if (k < needle) {
            i = i->rgt;
        }
        else {
            *value = i->value;
            return true;
        }
    }
    return false;
}

bool llrb_search(int needle, int * value, TreeNodePtr root)
requires IsTree(root, ?t) ∗ integer(value, ?v_0) ∗ bst(t) == true;
ensures IsTree(root, t) ∗ integer(value, ?v_1) ∗ IsMaybe(result, v_1, tree_search(needle, t));
{
    TreeNodePtr i = root;

    close IsCtx(root, i, NilCtx);

    while (i ≠ 0)
    invariant IsCtx(root, i, ?ctx) ∗ IsTree(i, ?subt) ∗ ctx_compose(ctx, subt) == t ∗
    tree_search(needle, t) == tree_search(needle, subt) ∗ integer(value, v_0);
    {
        open IsTree(i, subt);
        assert i->lft ↦ ?lft ∗ i->rgt ↦ ?rgt ∗ IsTree(lft, ?l_0) ∗ IsTree(rgt, ?r_0) ∗ i->value ↦ ?v ∗
        i->color ↦ ?clr;

        int k = i->key;

        TreeNodePtr prev_i = i; ugly

        if (needle < k) {
            i = i->lft;
            close IsTreeNode(prev_i, lft, k, v, clr, rgt);
            close IsCtx(root, lft, RgtCtx(k, v, clr, r_0, ctx));
        }
        else if (k < needle) {
            i = i->rgt;
            close IsTreeNode(prev_i, lft, k, v, clr, rgt);
            close IsCtx(root, rgt, LftCtx(l_0, k, v, clr, ctx));
        }
        else {
            *value = i->value;
            rebuild_tree(root, i, t);
            close IsMaybe(true, v, tree_search(needle, subt));
            return true;
        }
    }
    null_ptr_is_Leaf(i);
```

```
        $rebuild\_tree(root, i, t)$;
        **close** $IsMaybe($**false**$, v_0, tree\_search(needle, Leaf))$;
        **return false**;
}
```

The Left-Leaning Red-Black Tree

We just saw how the BST is "better" than the linked list because it has faster best-case and average-case performance. However, the worst-case performance of the BST is the same (in fact, a little worse) than the performance of a linked list. This is because the worst-case BST is the same shape as a linked list: at each node, just one link is non-empty.

This worst-case is not a corner-case: it arises, for example, after inserting elements into the tree in order. In any case, worst-case analysis is the standard way of summarizing an algorithm's performance, and in this light the BST does badly. The problem, in short, is that the tree does not have to be *balanced*.

What is balance? There is more than one way to define this, but one is called "height balance". A height-balanced tree has restrictions on the length of paths from the root to all leaves. The restrictions ensure that the lengths are roughly equal for all paths. Why *roughly* equal? Because a "perfectly" height-balanced tree has 2^n-1 elements, and so cannot represent most index sizes; some measure of imbalance is inevitable.

The Left-Leaning Red-Black tree, or LLRB, is one of many data structures that has this goal. This section describes the data structure (which is quite simple), then formalizes it in VeriFast, and finally uses VeriFast to prove that the tree as described indeed does ensure that the tree is balanced.

Description

Recall that "color" field I added to the *Tree* definition and said to ignore? That field is the key to the LLRB. The LLRB enforces balance by placing constraints on the arrangement of red nodes (i.e., nodes with the color field set to red) and black nodes in the tree.

It is easiest to first describe the rules and then to think about why they make the tree balanced. There are two LLRB rules. These rules are named the "black rule" and the "red rule". Neither individually enforces balance; balance is only enforced by their combination.

The black rule

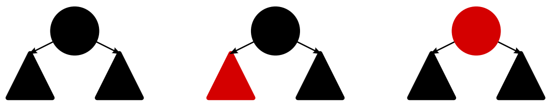
The "black rule" says that all paths from the root to a leaf contain the same number of black nodes. This immediately sounds like a rule that could enforce balance: if the tree *only* contains black nodes, then all paths are of the same length, which is a perfectly balanced tree. However, this rule places no limit on the non-black nodes: we could have an arbitrarily unbalanced tree composed entirely of red nodes.

The red rule

The limit on red nodes is done by the "red rule". The red rule says that red nodes can only appear as the left child of a black node.

It helps to think of " red-rooted" and "black-rooted" trees, where a tree with a red node at the root is red-rooted, and analogously for black. If we then make the definition that leaves are black-rooted, then all trees have a color.

Consider a branch. The root node has a color, the left subtree has a color, and the right subtree has a color. In an unrestricted tree, there would be $2^3 = 8$ color combinations. The red rule reduces this to just three:



There is a final rule in the LLRB, though it less fundamental than the other two: the root of the entire tree is colored black. This convention makes some reasoning and algorithms simpler.

Balance?

How does this enforce balance? What degree of balance? One way to think of balancedness is by comparing the longest and shortest paths in the tree. By doing so, we can establish that, in the LLRB, the longest path is no longer than twice the length of the shortest path.

Consider an LLRB with n black nodes on each path as enforced by the black rule. We call this the black-height of the tree. The shortest possible path is one composed entirely of black nodes, which would be length n .

In fact, *every* LLRB of black-height n has a path of length n , and it is always in the same place: along the right-hand spine of the tree. Why? All nodes along the right-hand spine are either the root node or a right subnode; convention enforces that the root is black, and the red rule enforces that the right subnodes are black (consult the above combinations).

Now consider the longest possible path in the same tree. This path will contain as many red nodes as possible to lengthen the path. The left subtree of the root will then be red. Its left subtree must be black, however, as red nodes must have black subnodes. This black subtree can again have a red left subtree. This simple pattern — black, red, black, red — repeats all the way to the leftmost leaf, which is black. There are therefore approximately as many reds as blacks on this path, making it length $2n$.

This longest possible path is therefore no longer than twice the length of the shortest path. This is the definition of "roughly balanced" that is enforced by the LLRB.

Formalization

The first thing we must define is the color itself. We do this in the C language proper, using an **enum** construction for it:

```
enum Color { Red, Blk };
typedef enum Color Color;
```

For the red rule, each LLRB tree is assigned a *color*. For the case that the tree is a *Branch* , this color is easy: it is the color stored at the root of the tree. For the case that the tree is a *Leaf* , the color is said to be black. The red rule then enforces that, for any node in the tree and its two child nodes, only three of the eight possible combinations of colors are present: at most one node may be red, and it can be either the root or the left node. This rule can be written as a VeriFast predicate:

```
predicate RedRule<K,V>(Tree<K,V> t; Color c) =
    switch (t) {
        case Leaf: return c == Blk;
        case Branch(l,k,v,c′,r): return
            c == c′ ∗
            RedRule(l, ?lc) ∗
            RedRule(r, Blk) ∗
            (c == Red ? lc == Blk : true);
    };
```

For the black rule, each LLRB tree is assigned a *black-height*, which is a natural number. For the case that the tree is a *Leaf* , the black-height is defined to be zero. For the case that the tree is a *Branch* , the black-height of both subtrees is the same height, say *h* . Then, if the tree is red at the root, the tree is defined to have height *h* , else it is black at the root, and it has height *h*+1 . In VeriFast we can write:

```
predicate BlackRule<K,V>(Tree<K,V> t; int h) =
    switch (t) {
        case Leaf: return h == 0;
        case Branch(l,k,v,c,r): return
            BlackRule(l, ?subh) ∗
            BlackRule(r, subh) ∗
            (c == Red ? h == subh : h == subh+1);
    };
```

We could then wrap these two rules up with one predicate asserting the color and height of the tree, like so:

```
predicate IsLLRB′<K,V>(Tree<K,V> t, Color c, int h) = RedRule(t, c) ∗ BlackRule(t, h);
```

In fact, I choose not to do this. The *RedRule* and *BlackRule* predicates, useful for explanatory purposes, are used so closely together that proofs are easier when they are expressed together. For this reason, I define the *IsLLRB* predicate directly, expressing both rules:

```
predicate IsLLRB<K,V>(Tree<K,V> t; Color c, int h) =
    switch (t) {
        case Leaf: return
            h == 0 ∗
            c == Blk;
        case Branch(l, k, v, c′, r): return
            c == c′ ∗
            IsLLRB(l, ?lftC, ?subH) ∗
            IsLLRB(r, Blk, subH) ∗
            ( c == Red ? h == subH ∗ lftC == Blk
            : c == Blk ∗ h == subH+1);
    };
```

Verifying balance

Together, the LLRB rules guarantee the balancedness of the tree. Specifically, they guarantee that the longest path from the root to a leaf is at most twice as long as the shortest path from the root to a leaf. Again, we can prove this in VeriFast!

First, we need definitions of shortest and longest paths. These are defined simply (*min* and *max* are defined as expected):

```
fixpoint int shortest_path<K,V>(Tree<K,V> t)
{
    switch (t) {
        case Leaf: return 0;
        case Branch(l, k, v, c, r):
            return min(shortest_path(l), shortest_path(r))+1;
    }
}

fixpoint int longest_path<K,V>(Tree<K,V> t)
{
    switch (t) {
        case Leaf: return 0;
        case Branch(l, k, v, c, r):
            return max(longest_path(l), longest_path(r))+1;
    }
}
```

The LLRB is not perfectly balanced, but "nearly" balanced, and we can give this a rigorous definition:

```
predicate NearlyBalanced<K,V>(Tree<K,V> t) =
    longest_path(t) ≤ 2 * shortest_path(t);
```

Our proof that LLRBs satisfy this predicate relies on two claims. First, the shortest path of an LLRB at black-height *h* is length *h* if black-rooted, or *h*+1 if red-rooted. Second, the longest path of this LLRB has an upper bound; this bound is 2**h* if black-rooted, and (2**h*)+1 if red-rooted. We prove these separately.

First, the shortest path of an LLRB runs down its right-hand edge: all right nodes are black and each adds one to the height.

```
lemma void llrb_shortest_path<K,V>(Tree<K,V> t)
    requires IsLLRB(t, ?c, ?h);
    ensures IsLLRB(t, c, h) ∗ shortest_path(t) == (c == Blk ? h : h+1);
{
    switch (t) {
        case Leaf: leaf_is_black_0();
        case Branch(l, k, v, c_0, r):
            close Branch(t, l, k, v, c_0, r);
            llrb_subtrees(t);
            llrb_shortest_path(l);
            llrb_shortest_path(r);
            close IsLLRB(t, c, h);
            return;
    }
}
```

What's going on here? To me, it looks VeriFast makes short work of a significant theorem. Because *t* is an LLRB, the subtrees have equal heights; call this *subh* . The inductive hypothesis on the right subtree, *r* , which is black, means *shortest_path*(*r*) == *subh* . VeriFast then, because of the ternary operator in the postcondition, does a proof-by-cases on the color *c* .

If *c* == *Red* , then *subh* == *h* , and so *shortest_path*(*r*) == *h* . This also means *l* must be black, so *shortest_path*(*l*) == *subh* == *h* . Then *shortest_path*(*t*) is expanded to *min*(*shortest_path*(*l*), *shortest_path*(*r*))+1 , which reduces to *h*+1 , which is the desired result.

Else, *c* == *Blk* , and *subh* == *h*-1 . There are then two cases for the color of *l* ; so *shortest_path*(*l*) is either *subh* or *subh*+1 , which is either *h*-1 or *h* . In either case, *min*(*shortest_path*(*l*), *shortest_path*(*r*)) is *h*-1 . Then, because *c* == *Blk* , *shortest_path*(*t*) = *h* .

The worst-case longest path of an LLRB is one that alternates between black and red nodes: it cannot be any longer without introducing two red nodes in a row, which violates the red rule. If such a longest path has *h* black nodes, it would in the worst case have *h*+1 red nodes, with reds at each end. This makes for a black-height of *h* , and a path length of *h*+*h*+1 or (2**h*)+1 if the root is red, else a path length of *h*+*h* or 2**h* if black.

```
lemma void llrb_longest_path<K,V>(Tree<K,V> t)
    requires IsLLRB(t, ?c, ?h);
    ensures IsLLRB(t, c, h) ∗ longest_path(t) ≤ (c == Blk ? 2*h : (2*h)+1);
{
    open IsLLRB(t, c, h);
    switch (t) {
        case Leaf:
        case Branch(l, k, v, c0, r):
            llrb_longest_path(l);
            llrb_longest_path(r);
    }
    close IsLLRB(t, c, h);
}
```

The proof structure is exactly the same as that for *shortest_path* . VeriFast's approach here is as follows. It does a check of both cases for the color *c* . If *c* == *Red* , then by the inductive hypothesis we get that *longest_path*(*l*) ≤ 2**h* and *longest_path*(*r*) ≤ 2**h* . VeriFast then sees that *max*(*longest_path*(*l*), *longest_path*(*r*)) ≤ 2**h* . Then *longest_path*(*t*) == (2**h*)+1 .

Else, *c* == *Blk* .

From here, it is easy for VeriFast to see that the LLRB satisfies balancedness:

```
lemma void llrb_nearly_balanced<K,V>(Tree<K,V> t)
    requires IsLLRB(t, ?c, ?h);
    ensures IsLLRB(t, c, h) ∗ NearlyBalanced(t);
{
    llrb_shortest_path(t);
    llrb_longest_path(t);
    close NearlyBalanced(t);
}
```

Insert

The first of the two basic mutation functions on indexes is *insert* . In the same way we defined *index_search* as our list search algorithm, we will define a *index_insert* fixpoint.

Specification

Semantics of insertion

Before we do so, we should clarify what insert conceptually does. At the level of types, it takes an index I_0 , and a key-value pair (k,v) , and returns a new index I_1 . That is, I_1 == *index_insert*(k, v, I_0) .

What is the relationship between I_0 and I_1 ? An index is a function, so the fundamental thing that one can do with it is apply it to an argument. In index terminology, this is *search* . Therefore we should define *insert* in terms of how it alters the behaviour of function application — that is, in terms of *index_search* .

Consider searching for a new key *kn* . What should *index_search*(*kn*, I_1) be? If *kn* == *k* , *index_search*(*kn*, I_1) should certainly be *Just*(v) , because that is the value we just inserted at k . This gives us our first rule: *index_search*(k, *index_insert*(k, v, I_0)) == *Just*(v) .

Otherwise, *kn* ≠ k . As *index_insert*(k, v, I_0) is only supposed to affect the function at k , search should be unaffected by the insert. This gives us our second rule: where *kn* ≠ k , *index_search*(*kn*, *index_insert*(k, v, I_0)) == *index_search*(*kn*, I_0) .

We can summarize these two rules in VeriFast using the ternary operator: *index_search*(*kn*, *index_insert*(k, v, L_0)) == (*kn* == k ? *Just*(v) : *index_search*(*kn*, L_0)) . The following lemma declaration is then our definition of the semantics of insertion, and is our proof obligation for any definition of *index_insert* .

```
lemma void index_insert_proof<V>(int k, V v, List<Pair<int,V> > I0, int kn);
    requires sorted(I0) == true;
    ensures index_search(kn, index_insert(k, v, I0)) == (kn == k ? Just(v) : index_search(kn, I0));
```

An index insert algorithm

This gives us a definition of insert, but it does not give us an *algorithm* that satisfies it. We need one because we might wish to assert that, for example, inserting (6,13) into [(3,4), (5,8)] gives us [(3,4), (5,8), (6,13)]. With just the above definition, we can't do this: it just tells us a property that the latter list must satisfy. There may be many such lists, and even if there is only one, VeriFast could not determine how to generate it.

An incorrect algorithm

Here is one *index_insert* algorithm that passes the above semantics test. This algorithm is extremely simple: prepend the new key-value pair on to the start of the list.

```
fixpoint List<Pair<int,V> > index_insert_2<V>(int k, V v, List<Pair<int,V> > I0)
{
    return Cons(Pair(k,v), I0);
}
```

Then *index_search* will, if the searched-for key is equal to the inserted key, find it at the beginning of the list and return the associated value. Else it will move past the prepended key-value pair and search the rest of the list, as the semantics demands. The correctness proof requires no work at all:

```
lemma void index_insert_2_proof<V>(int k, V v, List<Pair<int,V> > I0, int kn)
    requires true;
    ensures index_search(kn, index_insert_2(k, v, I0)) == (kn == k ? Just(v) : index_search(kn, I0));
{
}
```

It seems our specification is not strong enough. I included this example because I tripped up on it: it is easy to under-specify something and then be satisfied with your easy proof of correctness.

Recall that the index data type we defined has its key-value pairs in a sorted order. We could not express this in the type system, and so we instead defined a fixpoint, *sorted* , which we assert holds for any valid index. We have a second proof obligation for any function operating on indexes: given that the input indexes are sorted, any returned indexes are also sorted. That is, it maintains sortedness. The above *index_insert* algorithm preserves neither sortedness nor uniqueness and is therefore incorrect.

A correct algorithm

The correct algorithm is fairly intuitive: find the place in the list where all keys on the left are less than the inserted key and all keys on the right are greater than the inserted key, then either replace or insert the new value there. The definition works by matching on the list I_0 . If it is empty, insertion returns the single-element index containing the pair (k,v) . Otherwise, we have a first pair (k_0,v_0) and the rest of the index I_1 . If $k == k_0$, the defined semantics allow us to replace the old v_0 with the new v . If $k < k_0$, we can insert (k,v) at the head of the index and maintain the sortedness. Otherwise, $k_0 < k$, and we cannot perform the insert locally while maintaining the sorted list; we therefore recurse and insert into I_1 .

```
fixpoint List<Pair<int,V> > index_insert<V>(int k, V v, List<Pair<int,V> > I0)
{
    switch (I0) {
        case Nil: return Cons(Pair(k,v), Nil);
        case Cons(e0, I1): return
            switch (e0) {
                case Pair(k0, v0): return
                    ( k < k0 ? Cons(Pair(k,v), I0) Insert new pair at start
                    : k > k0 ? Cons(e0, index_insert(k, v, I1)) Insert into rest
                    : Cons(Pair(k, v), I1) Replace old value with new
                    );
            };
    }
}
```

Proof of semantics for *index_insert*

As before, we have two proof obligations: that this algorithm satisfies *index_insert_proof* , and that it maintains sortedness. Here is the body of the lemma declared above:

```
lemma void index_insert_proof<V>(int k, V v, List<Pair<int,V> > I0, int kn)
    requires sorted(I0) == true;
    ensures index_search(kn, index_insert(k, v, I0)) == (kn == k ? Just(v) : index_search(kn, I0));
{
    if (kn == k)
        search_k_in_insert_k_v_is_v(k, v, I0);
    else
        search_non_k_in_insert_k_v_unaffected(k, v, I0, kn);
}
```

We rely on two lemmas for the disjunction in the proof: one for where $kn == k$, and another for where $kn \neq k$. It is easy to see from their contracts that together they give the proof postcondition. As usual, the structure of both is inductive: if the property holds for the tail of the list, then it holds for the rest of the list.

The first lemma, *search_k_in_insert_k_v_is_v* , says that, after inserting a new key-value pair, searching for the key yields the value. Consider the index I_0 before insertion. If it is empty, we know from the definition of insertion that insertion yields a single-pair index, and then we know from the definition of search that searching for the key yields the value. If the index is non-empty, we have k_0 and v_0 at the head and a tail I_1 . If the key we are inserting is equal to k_0 , then the definition of insert says that we replace the value; the definition of search on that then says that we yield that value.

If the keys are not equal, the definition of search says that, when the keys are not equal, we search the tail of the list, I_1 . We are now in the inductive case: assume that the proposition is true for the tail I_1 . Then inserting the key-value pair into I_1 gives us a list that, if we search for the key in it, yields the value — and the definition of search says that that is what we do.

Induction is valid here because the list structure we are performing induction on is finite (as are all inductive data structures in VeriFast).

All cases but the inductive one are empty: VeriFast can do a short proof search. This makes the lemma fairly compact (and it would be considerably shorter with multi-level pattern-matching):

```
lemma void search_k_in_insert_k_v_is_v<V>(int k, V v, List<Pair<int,V> > I0)
    requires sorted(I0) == true;
    ensures index_search(k, index_insert(k, v, I0)) == Just(v);
{
    switch (I0) {
        case Nil:
        case Cons(kv0, I1):
            switch (kv0) {
                case Pair(k0, v0): search_k_in_insert_k_v_is_v(k, v, I1);
            }
        }
}
```

The second lemma, *search_non_k_in_insert_k_v_unaffected* , says that inserting a key-value pair does not affect search for other keys. The structure is similar, and I omit the description of the non-inductive cases. In the inductive case, we have that the inserted key is not equal to the head key, in which case search says to return search the tail, and we can assume that that search is unaffected by using the inductive assumption on the tail of the list.

```
lemma void search_non_k_in_insert_k_v_unaffected<V>(int k, V v, List<Pair<int,V> > I0, int kn)
        requires sorted(I0) == true ∧ kn ≠ k;
        ensures index_search(kn, index_insert(k, v, I0)) == index_search(kn, I0);
{
        switch (I0) {
                case Nil:
                case Cons(kv0, I1):
                        switch (kv0) {
                                case Pair(k0, v0): search_non_k_in_insert_k_v_unaffected(k, v, I1, kn);
                        }
        }
}
```

Proof of maintenance of sortedness

As discussed, the insert algorithm must maintain sortedness. This we must prove. This proof is a little more complex than the previous one, and itself requires two lemmas.

I describe the proof top-down.

```
lemma void index_insert_preserves_sorted<V>(int k, V v, List<Pair<int,V> > I0)
        requires sorted(I0) == true;
        ensures sorted(index_insert(k, v, I0)) == true;
{
        switch (I0) {
                case Nil:
                case Cons(e0, I1):
                        switch (e0) {
                                case Pair(k0, v0):
                                        if (k < k0) {
                                                loosen_lowerBound(k, k0, I1);
                                        }
                                        else if (k > k0) {
                                                index_insert_preserves_sorted(k, v, I1);
                                                index_insert_preserves_lowerBound(k, v, k0, v0, I1);
                                        }
                        }
        }
}
```

```
lemma void index_insert_preserves_lowerBound<V>(int k, V v, int k_0, V v_0, List<Pair<int,V> > I_1)
    requires sorted(Cons(Pair(k_0, v_0), I_1)) == true ∧ k_0 < k;
    ensures lowerBound(k_0, index_insert(k, v, I_1)) == true;
{
    switch (I_1) {
        case Nil:
        case Cons(e_1, I_2):
            switch (e_1) {
                case Pair(k_1, v_1):
                    if (k_1 < k) {
                        index_insert_preserves_lowerBound(k, v, k_1, v_1, I_2);
                        loosen_lowerBound(k_0, k_1, index_insert(k, v, I_2));
                    }
            }
    }
}
```

Description

I describe insertion in two stages. I omit the linked-list description that I used to introduce VeriFast in the search section. I first describe insertion into a BST, which is a basic high-school algorithm. I then describe how the LLRB insertion algorithm is an extension of this.

Insertion into a BST

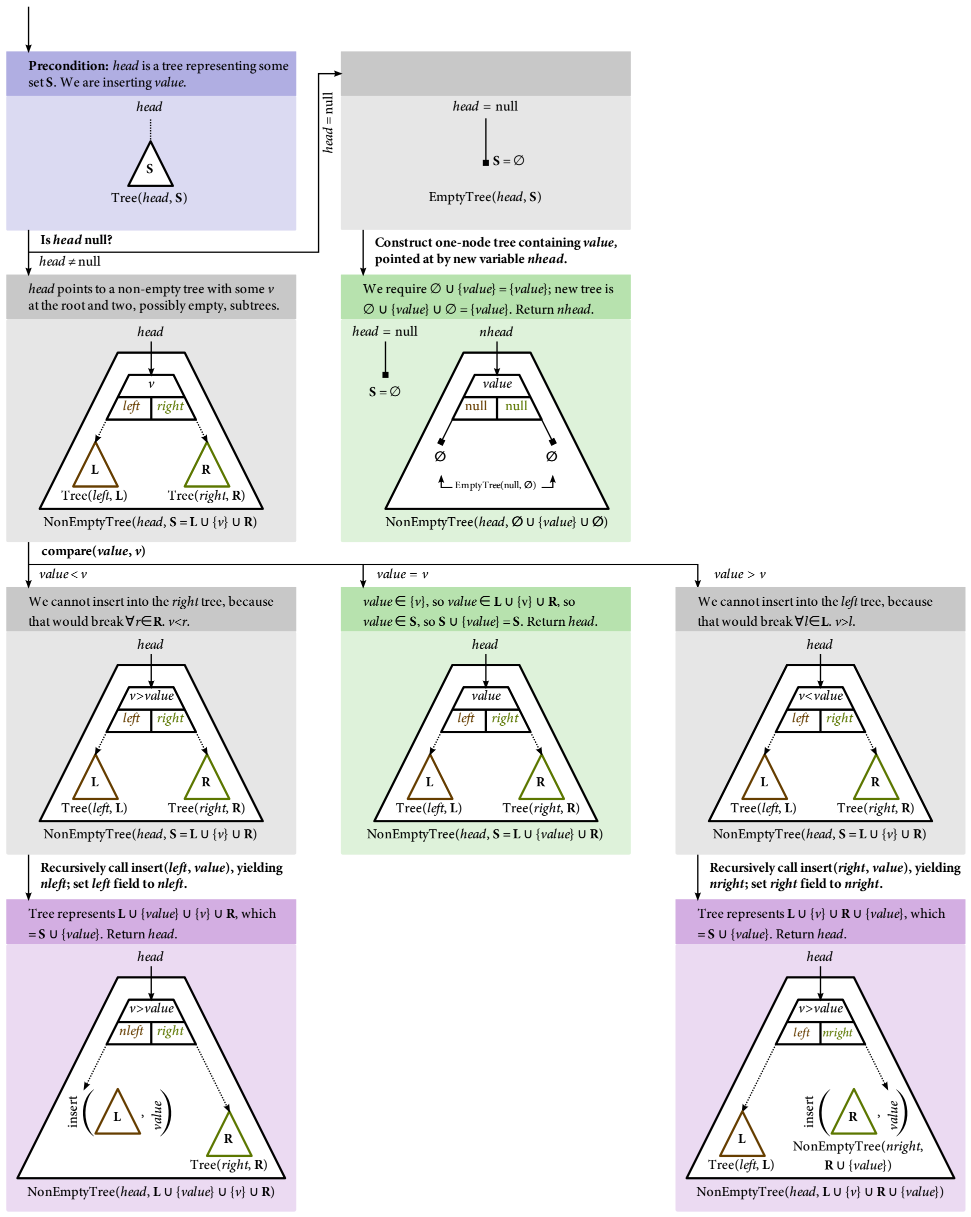
In the section on search, we saw how to search our simple index structure, and then how binary search can be said to implement this simpler algorithm. Analogously, we have just seen how to insert on our index structure, and we shall now see how insertion on BSTs relates to this.

One way to describe this is with a VeriFast fixpoint function. What is the type of this function? It takes a tree to insert into: that is, a *Tree<K,V>* . What does it do with this tree? VeriFast fixpoints are purely functional: mutation is not part of the semantics. The fixpoint therefore does not update in-place, as would our C implementation. Instead it returns a new tree (another *Tree<K,V>*) which is the "updated" tree. The two other arguments it takes are a key and value which comprise the key-value pair to insert. As with the BST search function, we specialize *K* to **int** because of the ordering relation that the algorithm requires. Therefore our fixpoint will have the signature: *Tree<***int***,V> bst_insert_fixpoint<V>(***int** *nk, V nv, Tree<***int***,V> T)* .

BST insertion is scarcely more complex than BST search, and follows the same pattern. If *T* is a leaf, insertion creates one node with the new key-value pair. Otherwise, *T* is some *Branch(l, k, v, c, r)* . We compare the new key *nk* to the root key *k* . If *nk* == *k* , we do what we did with list insert: replace the old value with the new value, so the new tree is *Branch(l, k, nv, c, r)* . Otherwise, either *nk* < *k* or *k* < *nk* . In BST search terms, *nk* belongs either in the left tree *l* or the right tree *r* . We then do the insert on the appropriate subtree, and the rest of the tree is unaffected: so if *nk* < *k* , the result is *Branch(bst_insert_fixpoint(nk, nv, l), k, v, c, r)* .

Here is the VeriFast fixpoint definition. As before, ignore the coloring.

```
fixpoint Tree<int,V> bst_insert_fixpoint<K,V>(int nk, V nv, Tree<int,V> t) {
    switch (t) {
        case Leaf: return Branch(Leaf, nk, nv, Red, Leaf);
        case Branch(l, k, v, c, r):
            return
                nk < k ? Branch(bst_insert_fixpoint(nk, nv, l), k, v, c, r) :
                k < nk ? Branch(l, k, v, c, bst_insert_fixpoint(nk,nv,r)) :
                                        Branch(l, k, nv, c, r)
                ;
    }
}
```



Precondition: head is a tree representing some set S. We are inserting value.
head
S
Tree(head, S)
head = null
head = null
S = ∅
EmptyTree(head, S)
Is head null?
head ≠ null
Construct one-node tree containing value, pointed at by new variable nhead.
head points to a non-empty tree with some v at the root and two, possibly empty, subtrees.
head
v
left
right
L
R
Tree(left, L)
Tree(right, R)
NonEmptyTree(head, S = L ∪ {v} ∪ R)
We require ∅ ∪ {value} = {value}; new tree is ∅ ∪ {value} ∪ ∅ = {value}. Return nhead.
head = null
nhead
S = ∅
value
null
null
∅
∅
EmptyTree(null, ∅)
NonEmptyTree(head, ∅ ∪ {value} ∪ ∅)
compare(value, v)
value < v
value = v
value > v
We cannot insert into the right tree, because that would break ∀r∈R. v<r.
head
v>value
left
right
L
R
Tree(left, L)
Tree(right, R)
NonEmptyTree(head, S = L ∪ {v} ∪ R)
value ∈ {v}, so value ∈ L ∪ {v} ∪ R, so value ∈ S, so S ∪ {value} = S. Return head.
head
value
left
right
L
R
Tree(left, L)
Tree(right, R)
NonEmptyTree(head, S = L ∪ {value} ∪ R)
We cannot insert into the left tree, because that would break ∀l∈L. v>l.
head
v<value
left
right
L
R
Tree(left, L)
Tree(right, R)
NonEmptyTree(head, S = L ∪ {v} ∪ R)
Recursively call insert(left, value), yielding nleft; set left field to nleft.
Recursively call insert(right, value), yielding nright; set right field to nright.
Tree represents L ∪ {value} ∪ {v} ∪ R, which = S ∪ {value}. Return head.
head
v>value
nleft
right
insert
L
value
R
Tree(right, R)
NonEmptyTree(head, L ∪ {value} ∪ {v} ∪ R)
Tree represents L ∪ {v} ∪ R ∪ {value}, which = S ∪ {value}. Return head.
head
v>value
left
nright
insert
R
value
L
Tree(left, L)
NonEmptyTree(nright, R ∪ {value})
NonEmptyTree(head, L ∪ {v} ∪ R ∪ {value})

We could now verify that this algorithm is correct. The proof obligation for this would follow the same scheme as with our proof of BST search: the operation over trees, when the trees are flattened, gives us the operation over lists. That is, *flatten*(*bst_insert_fixpoint*(*k*, *v*, *T*)) == *index_insert*(*k*, *v*, *flatten*(*T*)) .

However, I do not pursue this, because this fixpoint merely serves as an introduction to LLRB insert.

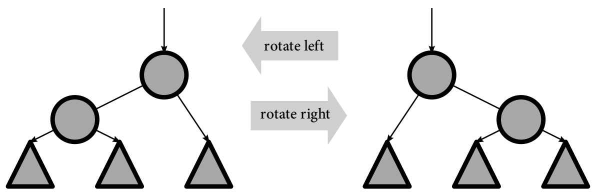
Insertion into an LLRB

Insertion into an LLRB follows a similar pattern to insertion into a standard BST. The algorithm is recursive. The first phase of the algorithm, in which we descend the tree to the position that the key must be in, is identical in the LLRB insertion algorithm. The difference is in the second phase, in which we ascend the tree to the root.

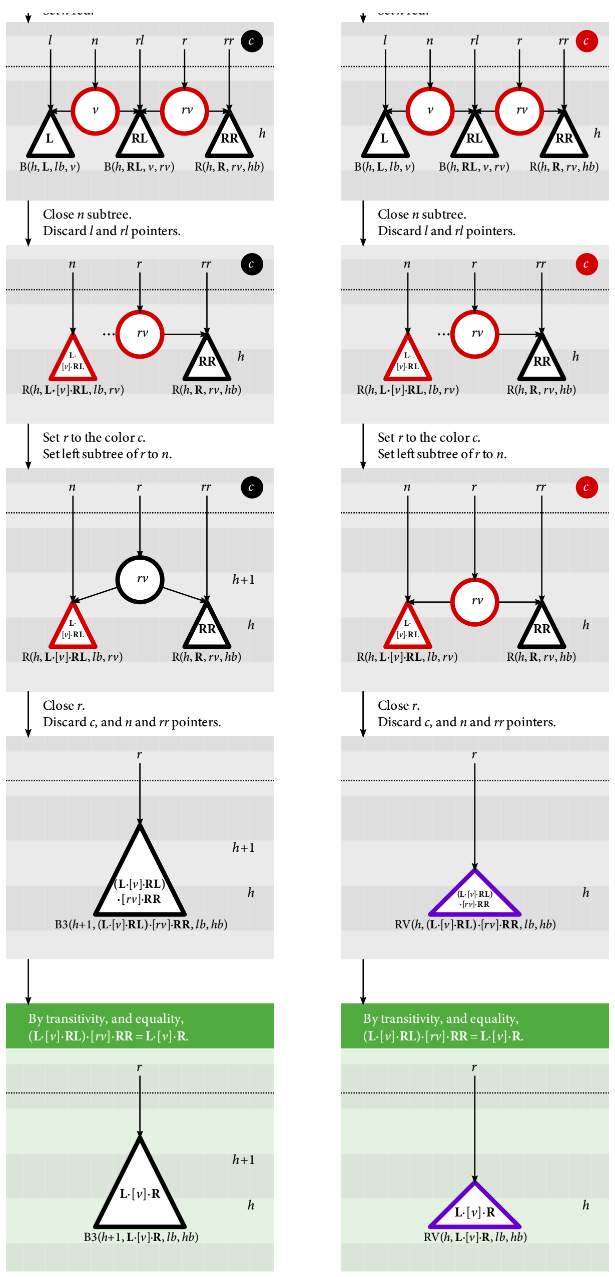
In BST insertion, this phase is trivial: we possibly create a new node with the new key and value at a leaf, and then just ascend the tree to the root and return it. In LLRB insertion, if we do not insert a new node (because we found that the key already exists), then it follows this same pattern. However, if we create a new node at a leaf, then we have increased the path from the root to that new leaf, and this potentially breaks the balancedness of the tree. Therefore, the ascension phase of the LLRB insert algorithm is concerned with correcting this unbalancedness.

Tree rotation

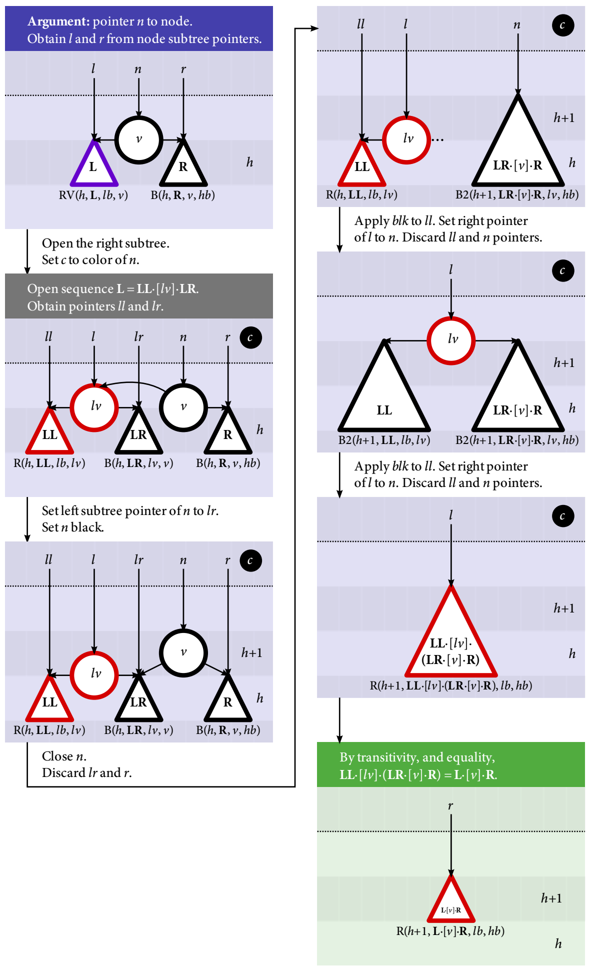
The fundamental operation that rebalances the tree structure is called a *tree rotation*. This operation is in fact extremely simple. Consider a tree that consists of two nodes at the top and three subtrees below them. There are two such trees:



The tree rotation transforms between these two tree structures. As such there are two symmetrical rotations, one which moves from the left tree to the right, and another that moves in the other direction. These are known as “left rotation” and a “right rotation”.


l n rl r rr c
v rv
L RL RR h
B(h, L, lb, v) B(h, RL, v, rv) R(h, R, rv, hb)
Close n subtree.
Discard l and rl pointers.
n r rr c
rv
L·[v]·RL RR h
R(h, L·[v]·RL, lb, rv) R(h, R, rv, hb)
Set r to the color c.
Set left subtree of r to n.
n r rr c
rv h+1
L·[v]·RL RR h
R(h, L·[v]·RL, lb, rv) R(h, R, rv, hb)
Close r.
Discard c, and n and rr pointers.
r
h+1
(L·[v]·RL)·[rv]·RR h
B3(h+1, (L·[v]·RL)·[rv]·RR, lb, hb)
RV(h, (L·[v]·RL)·[rv]·RR, lb, hb)
By transitivity, and equality,
(L·[v]·RL)·[rv]·RR = L·[v]·R.
r
h+1
L·[v]·R h
B3(h+1, L·[v]·R, lb, hb)
RV(h, L·[v]·R, lb, hb)

Left rotation


Argument: pointer n to node.
Obtain l and r from node subtree pointers.
RV(h, L, lb, v)
B(h, R, v, hb)
Open the right subtree.
Set c to color of n.
Open sequence L = LL·[lv]·LR.
Obtain pointers ll and lr.
R(h, LL, lb, lv)
B(h, LR, lv, v)
B(h, R, v, hb)
Set left subtree pointer of n to lr.
Set n black.
Close n.
Discard lr and r.
B2(h+1, LR·[v]·R, lv, hb)
Apply blk to ll. Set right pointer of l to n. Discard ll and n pointers.
B2(h+1, LL, lb, lv)
Apply blk to ll. Set right pointer of l to n. Discard ll and n pointers.
R(h+1, LL·[lv]·(LR·[v]·R), lb, hb)
By transitivity, and equality,
LL·[lv]·(LR·[v]·R) = L·[v]·R.
R(h+1, L·[v]·R, lb, hb)

Right rotation

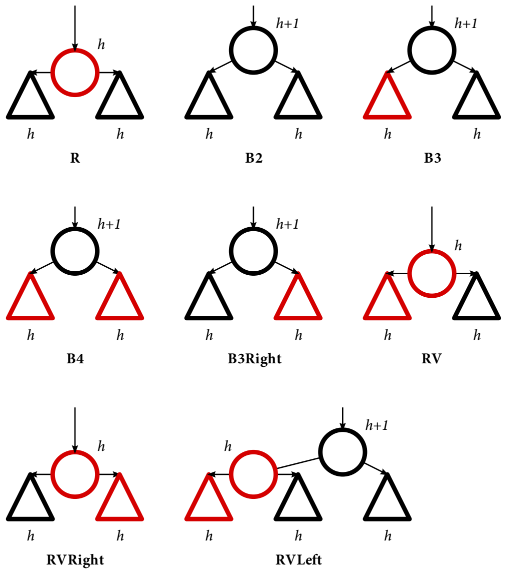
Outline of algorithm

As we ascend the tree, we keep track of whether that subtree is "broken". Brokenness in this context means that the tree has become a different type in a way that may be invalid as a subtree of its parent node. For example, if a previously black-rooted tree is now red then it is broken because it may be the child of a red node, which would break the red rule.

At each stage of ascension, we attempt to fix the tree. The goal is to make the tree the same color and black-height as it was before the insertion (I refer to this as having the same LLRB-type). Failing that, we transform the tree to a type that lends itself to being fixed further up the tree.

There are two situations in which we can stop. First, if we have a tree that is the same LLRB-type as before the insertion, we can stop, because this is the only property that the parent tree cares about in order to be a valid LLRB tree. Second, if we get to the root of the tree, we have fixed the tree all the way up, and so we can stop (and color the root black in order to maintain that invariant).

Now I describe the specific ways in which the tree may be broken, how the tree may get into these broken states, and how we can fix them. There are four kinds of broken tree in the insertion algorithm, and I give them the names *RV* , *RVRight* , *B3Right* , B_4 , and *RVLeft* .



OK trees and broken trees

An *RV* tree has a single red violation at the root, meaning a red root node has a red left subtree. An *RVRight* tree is the mirror image of this: a red root node has a red *right* subtree. A *B3Right* tree is like an *RVRight* with a black root node: it is a black root node with a red right subtree. A B_4 tree is a black root node with two red subtrees. Finally, an *RVLeft* tree is a black-rooted tree with an *RV* left subtree. All these trees have valid black-heights; it is only the red rule that they violate.

We then have fixing procedures that fix these broken trees using rotations and/or recolorings. (The details of these will be shown later; right now we just need to know what they do, not how they do it.) Two of these trees can be fully fixed, meaning after they are fixed then the whole tree is fixed. A *B3Right* can be transformed into a normal black-rooted tree of the same black-height. An *RVLeft* can be transformed into a normal red-rooted tree of the same black-height.

This just leaves us with the B_4 , *RVRight* and *RV* trees. These cannot be fully fixed at the point where they appear; instead, we defer the fixing to further up the tree. However, we can reduce the problem set. A B_4 can be transformed into a normal red-rooted tree of the same black-height. An *RVRight* tree can be transformed into an *RV* tree, by a simple left rotation.

In short, there are just two ways in which the tree can be broken after insertion. Firstly, if the tree was black-rooted before insertion, it may be red-rooted after insertion, which could cause a red violation or invalid black node subtree colors. Secondly, if the tree was red-rooted before insertion, it may be an *RV* after insertion, which is always a violation.

There are two things to note here. First, a black-rooted tree never gets transformed into an *RV* . Second, only a black-rooted tree can be empty, and in this case we return a new red node, which then requires fixing.

This description neatly gives us the contracts of two mutually recursive procedures: one that inserts into a red-rooted tree and another that inserts into a black-rooted tree. I now outline the body of each of these procedures and how they use the fixing procedures. As stated, both begin with the standard BST logic: if the tree is empty then create a new node; otherwise compare the new key with the key at the root, then either replace the value at the root node, or insert left or insert right. I have described what happens with an empty tree and what happens when the key exists. This leaves us with four cases to consider: inserting left and inserting right from red-rooted and black-rooted trees.

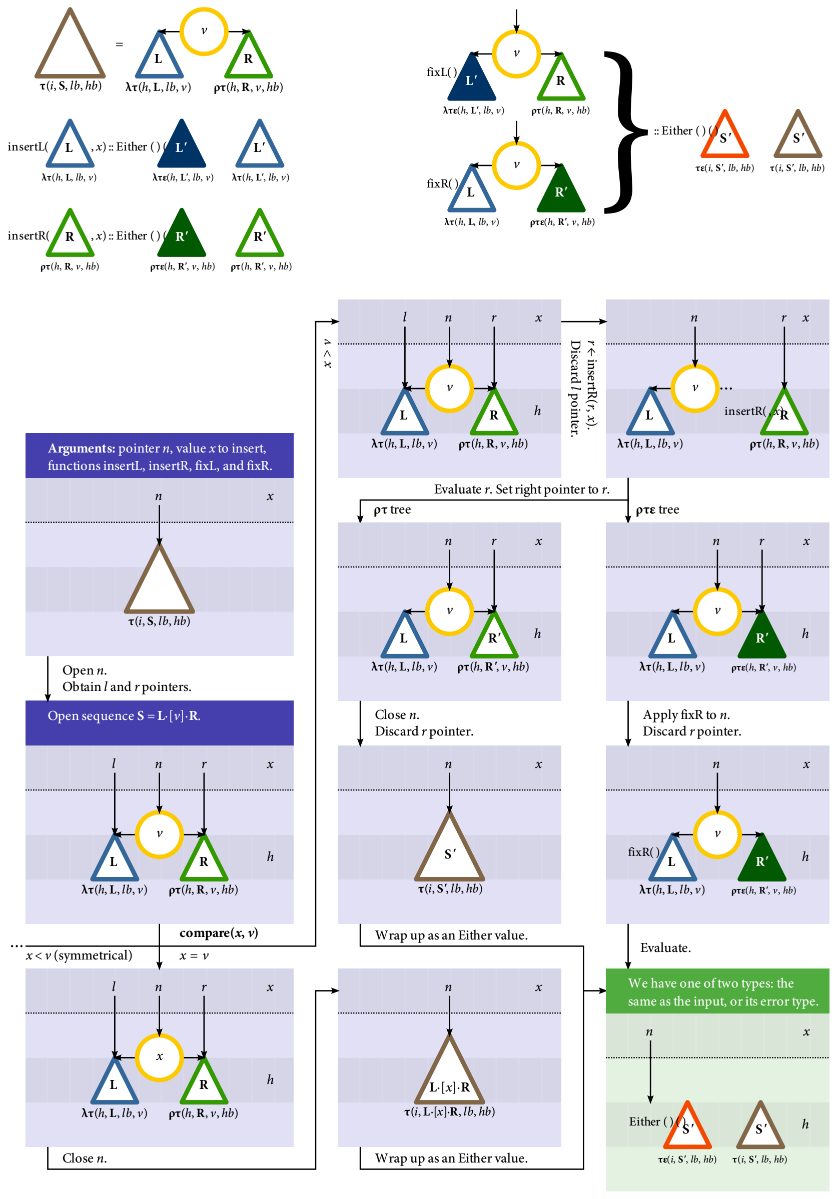
Let us first consider insertion into a red-rooted tree. Both subtrees are black-rooted, and so the new subtree returned, if it requires fixing, will be red. If we inserted left, this gives us an *RV* tree; if we inserted right, this gives us an *RVRight* tree. In the former case we simply leave the *RV* ; in the latter case, we fix the *RVRight* to give us an *RV* . This already gives us the postcondition for insertion into a red-rooted tree: either the new tree is red-rooted (where subtree insertion did not return a red-rooted tree), or it is an *RV* .

Now consider insertion into a black-rooted tree. First consider the easier case: inserting right. As the right subtree is always black, this may return a red-rooted subtree. Depending on whether the left subtree is red-rooted, this gives us either a *B3Right* tree or a B_4 tree. In the former case, we then use fixing procedure to give us a B_3 tree, and as this is a black-rooted tree, we have fully fixed the tree and can stop. In the latter case, we use the fixing procedure to give us a red-rooted tree, which we return and which will be fixed further up the tree.

Now the harder case: inserting left. The left subtree of a black node is unrestricted in color. We check its color and then use the appropriate insertion procedure. If the left subtree is black, it doesn't matter what it returns: if still black-rooted, great; if red-rooted, this is also valid; in either case, we don't need to do anything!

Finally, if the left subtree is red, it might become an *RV* . This makes the whole tree an *RVLeft* . We have a fixing procedure for that which returns a red-rooted tree. We then return that tree, and it will be fixed further up the tree.

Notice that all these manipulations maintain the black-height of the tree. The only point at which this changes is at the root of the tree. The entire tree is always black-rooted. If insertion makes it red-rooted, we then simply color the root black, which maintains the LLRB rules, increasing the black-height by one.



Verification

Tree rotation

The tree rotation algorithms have already been roughly described from a functional perspective. I first formalize this functional definition in VeriFast. I then prove some necessary lemmas concerning them. Finally, I implement them in C and show that they satisfy the fixpoint definition. Note that the two rotations, left and right, are exactly symmetrical. For this reason I only need describe one. I arbitrarily choose left rotation.

Fixpoints

I define rotation in VeriFast using fixpoint functions. I did not have to do this. I could have instead expressed the rotation directly in the C procedure contract: it takes a tree of such-and-such a form, and returns a tree of such-and-such a form. One reason I "raise" these procedures to a fixpoint is in order to give the same procedure multiple contracts. That is, the fixpoint is a minimal description of the procedure, and it can be used to fix multiple kinds of tree. If we were to just use C procedures, I would either have to implement rotation multiple times, or use an "adapter" design in which multiple procedures call a basic rotation procedure. This is undesirable because it creates unnecessary C code.

(Also, the fixpoint approach makes multiple implementations are easier: we might write an LLRB in Java and use VeriFast to verify it. If so, we would like to have as much of the proof as possible " outside" the implementation in order to minimize redundancy.)

The left rotation function is called *rotate_left_fixpoint* . This function transforms trees to trees. That is, it has one *Tree<K,V>* parameter and it returns a *Tree<K,V>* .

However, this typing does not restrict the input enough. Notice that the inputs to the rotation functions are not just any trees; they must have two or more nodes. Specifically, for a left rotation, the tree must have a root node and its right subtree must be non-empty.

This is a bit of a problem, as VeriFast only has simple algebraic types which cannot express these tree properties. We might then consider giving the fixpoint a **requires** block. However, VeriFast does not allow contracts on fixpoints: they are simple functions over simple datatypes.

We are left with no alternative but to admit the invalid tree types as input and in those cases just return dummy responses. I will then show how to deal with these dummy responses. The specific return value in these cases is not important; it just has to be correctly typed. I choose to return a *Leaf* .

Here's the fixpoint:

```
fixpoint Tree<K,V> rotate_left_fixpoint<K,V>(Tree<K,V> T)
requires has_right_branch(t);
{
    switch (T) {
        case Leaf: return Leaf; not possible
        case Branch(L, k, v, c, R):
            return switch (R) {
                case Leaf: return Leaf; not possible
                case Branch(RL, rk, rv, rc, RR):
                    return Branch(Branch(L, k, v, c, RL), rk, rv, rc, RR);
            };
    }
}
```

This may be hard to read. Just realize that the only valid input, *Branch*(*L*, *k*, *v*, *c*, *Branch*(*RL*, *rk*, *rv*, *rc*, *RR*)) , is transformed into *Branch*(*Branch*(*L*, *k*, *v*, *c*, *RL*), *rk*, *rv*, *rc*, *RR*) . The subtrees and nodes are all in the same order; it is only the bracketing that moves.

Lemmas

```
lemma void rotate_left_maintains_values<K,V>(Tree<K,V> t)
    requires has_right_branch(t) == true;
    ensures flatten(rotate_left_fixpoint(t)) == flatten(t);
{
    switch (t) {
        case Leaf: not possible
        case Branch(l,k,v,c,r):
            switch (r) {
                case Leaf: not possible
                case Branch(rl, rk, rv, rc, rr):
                    append_assoc(
                        flatten(l),
                        Cons(Pair(k,v),flatten(rl)),
                        Cons(Pair(rk,rv), flatten(rr))
                    );
            };
    }
}

lemma void rotate_left_is_branch<K,V>(Tree<K,V> t);
    requires has_right_branch(t) == true;
    ensures is_branch(rotate_left_fixpoint(t)) == true;

lemma void rotate_left_branches<K,V>(Tree<K,V> t0);
    requires Branch(t0, ?L, ?k, ?v, ?c, ?R0) ∗ Branch(R0, ?RL, ?rk, ?rv, ?rc, ?RR);
    ensures Branch(rotate_left_fixpoint(t0), ?L1, rk, rv, rc, RR) ∗ Branch(L1, L, k, v, c, RL);
```

Implementation

```
TreeNodePtr rotate_left(TreeNodePtr root);
requires IsTree(root, ?t0) ∗ has_right_branch(t0) == true;
ensures IsTree(result, ?t1) ∗ t1 == rotate_left_fixpoint(t0);

TreeNodePtr rotate_left(TreeNodePtr root)
requires IsTree(root, ?t0) ∗ has_right_branch(t0) == true;
ensures IsTree(result, ?t1) ∗ t1 == rotate_left_fixpoint(t0);
{
    open IsTree(root, _);
    TreeNodePtr rgt = root->rgt;
    open IsTree(rgt, _);
    root->rgt = rgt->lft;
    close IsTree(root, _);
    rgt->lft = root;
    close IsTree(rgt, _);

    return rgt;
}
```

Fixers

Broken predicates

In the informal description, I described various kinds of tree, both valid and invalid. The *IsLLRB* predicate is not specific enough to express specific kinds of valid tree, and it cannot represent any invalid trees. We therefore require more predicates to describe these.

First up are the valid trees. Red trees can be represented using *IsLLRB*(*T*, *Red*, *h*) , and this constrains both subtree colors. However, *IsLLRB*(*T*, *Blk*, *h*) does not constrain the color of the left subtree. I therefore create two predicates to represent these trees, called B_2 and B_3 .

```
predicate B_2<K,V>(Tree<K,V> tree, int h)
    = Branch(tree, ?l, _, _, Blk, ?r)
    ∗ IsLLRB(l, Blk, ?subh)
    ∗ IsLLRB(r, Blk, subh)
    ∗ h == subh+1
    ;

predicate B_3<K,V>(Tree<K,V> tree, int height)
    = Branch(tree, ?l, _, _, Blk, ?r)
    ∗ IsLLRB(l, Red, ?subh)
    ∗ IsLLRB(r, Blk, subh)
    ∗ height == subh+1
    ;
```

I now need to describe the kinds of broken tree that we experience in the insertion algorithm.

```
predicate B3Right<K,V>(Tree<K,V> tree, int height)
    = Branch(tree, ?l, _, _, Blk, ?r)
    ∗ IsLLRB(l, Blk, ?subh)
    ∗ IsLLRB(r, Red, subh)
    ∗ height == subh+1
    ;

predicate RV<K,V>(Tree<K,V> tree, int height)
    = Branch(tree, ?l, _, _, Red, ?r)
    ∗ IsLLRB(l, Red, height)
    ∗ IsLLRB(r, Blk, height)
    ;

predicate RVRight<K,V>(Tree<K,V> tree, int h)
    = Branch(tree, ?l, _, _, Red, ?r)
    ∗ IsLLRB(l, Blk, h)
    ∗ IsLLRB(r, Red, h)
    ;

predicate RVLeft<K,V>(Tree<K,V> tree, int h)
    = Branch(tree, ?l, _, _, Blk, ?r)
    ∗ RV(l, ?subh)
    ∗ IsLLRB(r, Blk, subh)
    ∗ h == subh+1
    ;

predicate B_4<K,V>(Tree<K,V> tree, int h)
```

```
        = Branch(tree, ?l, _, _, Blk, ?r)
        ∗ IsLLRB(l, Red, ?subh)
        ∗ IsLLRB(r, Red, subh)
        ∗ h == subh+1
        ;
```

These predicates can now be used as pre- and post-conditions to our various fixing procedures.

I take the same approach with many of the fixing functions as I did with the rotation functions: specify them using fixpoints and then verify that the C procedures implement those fixpoints. Note that I now consider this a mistake. There are seductive reasons to take this approach, but it causes more problems than it solves. This is discussed further in the evaluation section of this report.

Has black root?

This should be considered a warm-up: the *has_black_root* procedure does not mutate the tree, it only tells us what color the tree is. This follows our earlier definition of tree color: if the tree is a leaf, the procedure returns true, else it returns the color of the root node.

```
fixpoint bool has_black_root_fixpoint<K,V>(Tree<K,V> t) {
    switch (t) {
        case Leaf: return true;
        case Branch(l,k,v,c,r): return c == Blk;
    }
}
```

The C procedure then “ implements” the fixpoint.

```
bool has_black_root(TreeNodePtr root);
requires IsTree(root, ?T);
ensures IsTree(root, T) ∗ result == has_black_root_fixpoint(T);
```

The procedure can be used on any tree, but it is going to be useful where the tree is an LLRB. Let's show that it can be used in this situation and that the returned boolean corresponds to the color of the tree as defined by the predicate:

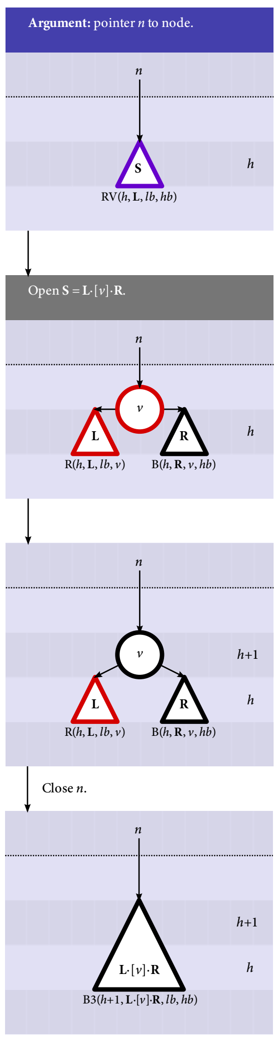
```
lemma void has_black_root_LLRB<K,V>(Tree<K,V> t)
    requires IsLLRB(t, ?c, ?h);
    ensures IsLLRB(t, has_black_root_fixpoint(t) ? Blk : Red, h);
{
    open IsLLRB(t, c, h);
    close IsLLRB(t, c, h);
}
```

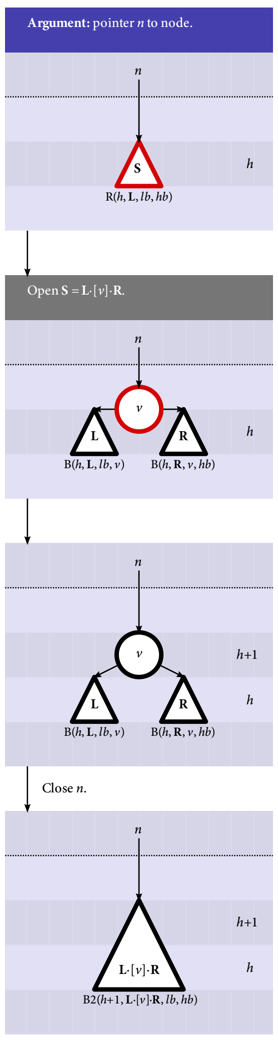
Finally, I write the actual procedure body. It is short and requires no real verification within the body, as we have moved the complexity on to the fixpoint:

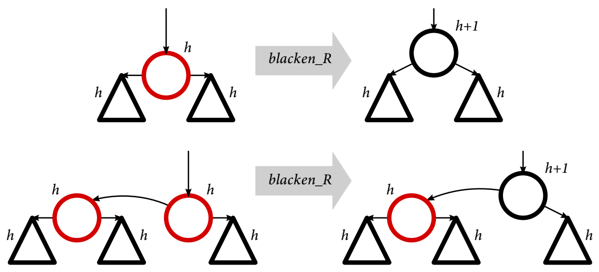
```
bool has_black_root(TreeNodePtr root)
requires IsTree(root, ?t);
ensures IsTree(root, t) ∗ result == has_black_root_fixpoint(t);
```

```
{
    open IsTree(root, t);
    if (root == 0) {
        close IsTree(root, t);
        return true;
    }
    else {
        bool result = root->color == Blk;
        close IsTree(root, t);
        return result;
    }
}
```

blacken_R





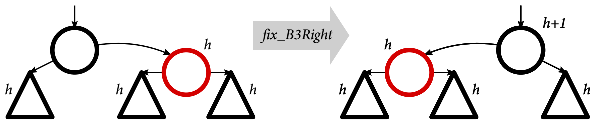


This couldn't be much simpler a procedure: we just color the red root black. This gets a little interesting when we consider that this procedure can be used in a couple of ways: to blacken a red-rooted tree, raising the height, or to blacken a red-violated tree, raising the height and fixing the violation.

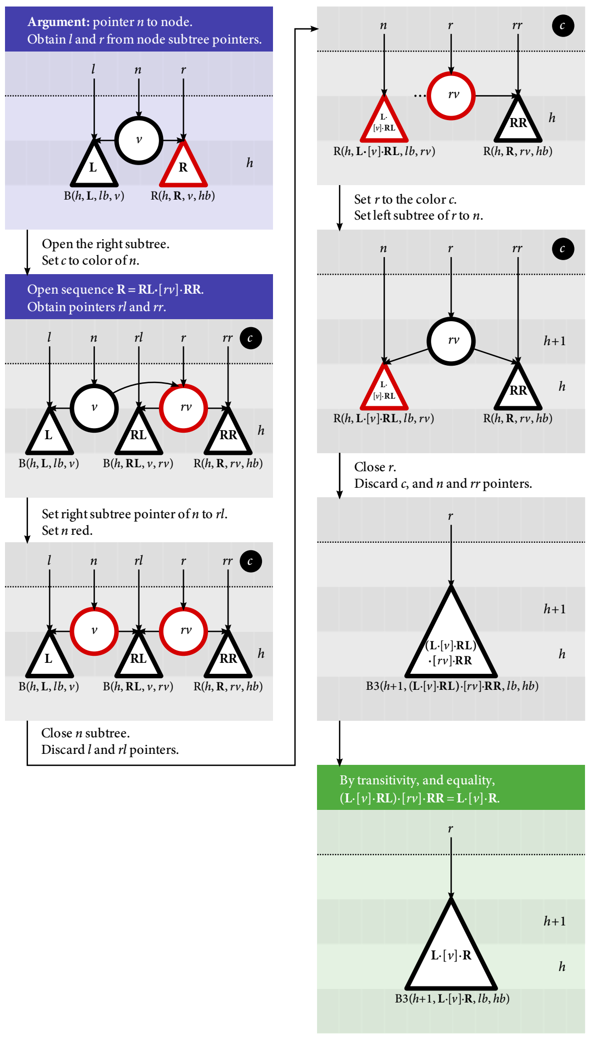
This dual use calls for multiple contracts to be placed on the procedure. However, VeriFast does not support this. I take another approach to avoid the duplication of the same C procedure in order to provide different contracts. I describe the operation of *blacken_R* using a fixpoint. Then I show that this fixpoint can be given those multiple “contracts”.

```
fixpoint Tree<K,V> blacken_R_fixpoint<K,V>(Tree<K,V> t)
requires IsLLRB(t, Red, ?h);
{
    return switch (t) {
        case Leaf: return Leaf; not possible
        case Branch(l,k,v,c,r): c == Red
            return Branch(l,k,v,Blk,r);
    };
}

lemma void blacken_R_applied_to_R<K,V>(Tree<K,V> T)
    requires IsLLRB(T, Red, ?h);
    ensures IsLLRB(blacken_R_fixpoint(T), Blk, h+1) ∗ flatten(blacken_R_fixpoint(T)) ==
    flatten(T);
{
    IsLLRB_dup(T);
    open IsLLRB(T, Red, h);
    switch (T) {
        case Leaf: not possible
        case Branch(L,k,v,c,R): c == Red
            close IsLLRB(blacken_R_fixpoint(T), Blk, h+1);
    };
}

lemma void blacken_R_applied_to_RV<K,V>(Tree<K,V> T)
    requires RV(T, ?h);
    ensures IsLLRB(blacken_R_fixpoint(T), Blk, h+1) ∗ flatten(blacken_R_fixpoint(T)) ==
    flatten(T);
{
    open RV(T, h);
    open Branch(T, _, _, _, _, _);
    switch (T) {
        case Leaf: not possible
        case Branch(L,k,v,c,R): c == Red
            close IsLLRB(blacken_R_fixpoint(T), Blk, h+1);
    };
}

void blacken_R(TreeNodePtr root);
requires IsTree(root, ?t0) ∗ is_branch(t0) == true;
ensures IsTree(root, ?t1) ∗ t1 == blacken_R_fixpoint(t0);

void blacken_R(TreeNodePtr root)
requires IsTree(root, ?t0) ∗ is_branch(t0) == true;
ensures IsTree(root, ?t1) ∗ t1 == blacken_R_fixpoint(t0);
{
    open IsTree(root, t_0);
    root->color = Blk;
    close IsTree(root, _);
}
```

fix_B3Right



A functional visual description of the procedure

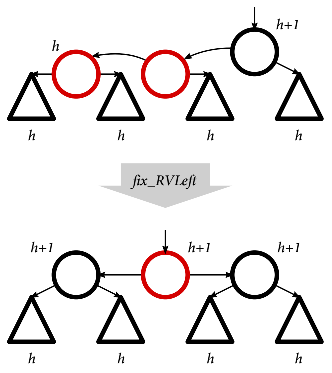

Argument: pointer n to node.
Obtain l and r from node subtree pointers.
l n r
v
L
R
h
B(h, L, lb, v)
R(h, R, v, hb)
Open the right subtree.
Set c to color of n.
Open sequence R = RL·[rv]·RR.
Obtain pointers rl and rr.
l n rl r rr c
v rv
L RL RR
h
B(h, L, lb, v)
B(h, RL, v, rv)
R(h, R, rv, hb)
Set right subtree pointer of n to rl.
Set n red.
Close n subtree.
Discard l and rl pointers.
n r rr c
L·[v]·RL
R(h, L·[v]·RL, lb, rv)
R(h, R, rv, hb)
Set r to the color c.
Set left subtree of r to n.
h+1
Close r.
Discard c, and n and rr pointers.
(L·[v]·RL)·[rv]·RR
B3(h+1, (L·[v]·RL)·[rv]·RR, lb, hb)
By transitivity, and equality,
(L·[v]·RL)·[rv]·RR = L·[v]·R.
L·[v]·R
B3(h+1, L·[v]·R, lb, hb)

The same procedure, algorithmically

The task of *fix_B3Right* is to flip the shape of the tree such that the red is on the left-hand-side. This is achieved by two recolorings and a rotation.

```
fixpoint Tree<K,V> fix_B3Right_fixpoint<K,V>(Tree<K,V> T)
{
    return switch (T) {
        case Leaf: return Leaf; not possible
        case Branch(L, k, v, c, R): c == Blk
            return switch (R) {
                case Leaf: return Leaf; not possible
                case Branch(RL, rk, rv, rc, RR): rc == Red
                    return rotate_left_fixpoint(Branch(L, k, v, Red, Branch(RL, rk, rv, Blk,
                    RR)));
            };
    };
}

lemma void fix_B3Right_lemma<K,V>(Tree<K,V> t)
requires B3Right(t, ?h);
ensures B_3(fix_B3Right_fixpoint(t), h) ∗ flatten(fix_B3Right_fixpoint(t)) == flatten(t);
{
    open B3Right(t, h);
    open Branch(t, ?l, _, _, Blk, ?r);
    switch (t) {
        case Leaf:
        case Branch(l′,k,v,c,r′):
            open IsLLRB(r, Red, h-1);
            switch (r′) {
                case Leaf:
                case Branch(rl,rk,rv,rc,rr):
                    rotate_left_maintains_values(t);
                    close Branch(fix_B3Right_fixpoint(t), ?l_1, _, _, _, ?r_1);
                    close IsLLRB(l_1, Red, h-1);
                    close B_3(fix_B3Right_fixpoint(t), h);
            };
    };
}

TreeNodePtr fix_B3Right(TreeNodePtr root);
requires IsTree(root, ?t_0) ∗ B3Right(t_0, ?h);
ensures IsTree(result, fix_B3Right_fixpoint(t_0)) ∗ B3Right(t_0, h);

TreeNodePtr fix_B3Right(TreeNodePtr root)
requires IsTree(root, ?T_0) ∗ B3Right(T_0, ?h);
ensures IsTree(result, fix_B3Right_fixpoint(T_0));
{
    open B3Right(T_0, h);
    open Branch(T_0, ?L, ?k, ?v, Blk, ?R_0);

    open IsTree(root, T_0);
    red_has_branch(R_0);
    open IsTree(root->rgt, R_0);
    root->color = Red;
    root->rgt->color = Blk;
    close IsTree(root->rgt, ?R_1);
    close IsTree(root, ?T_1);

    TreeNodePtr result = rotate_left(root);
    rotate_left_is_branch(T_1);
    rotate_left_maintains_values(T_1);
    llrb_emp(L);
    llrb_emp(R0);

    return result;
}
```

fix_RVLeft

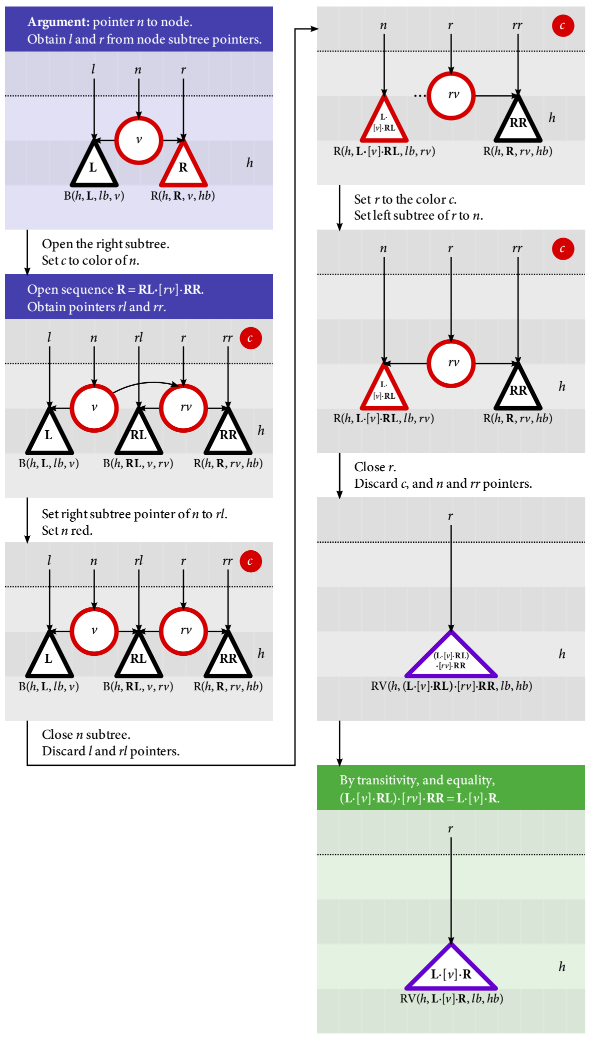


This looks the most complex of the fixers for insertion. After receiving a red-violated tree from the left subtree, we can fix it with a rotation and a recoloring.

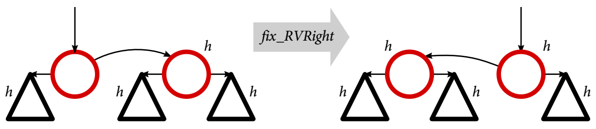
```
TreeNodePtr fix_RVLeft(TreeNodePtr root);
requires IsTree(root, ?t0) ∗ RVLeft(t0, ?h);
ensures IsTree(result, ?t1) ∗ IsLLRB(t1, Red, h) ∗ flatten(t1) == flatten(t0);

TreeNodePtr fix_RVLeft(TreeNodePtr root)
requires IsTree(root, ?t0) ∗ RVLeft(t0, ?h);
ensures IsTree(result, ?t1) ∗ IsLLRB(t1, Red, h) ∗ flatten(t1) == flatten(t0);
{
    RVLeft_has_left_branch(t0);
    open RVLeft(t0, h);
    open RV(_, h-1);
    TreeNodePtr result = rotate_right(root);
    rotate_right_branches(t0);
    rotate_right_maintains_values(t0);
    open Branch(?t1, ?l, ?dk, ?dv, Red, ?r);
    open Branch(r, _, _, _, Blk, _);
    close IsLLRB(r, Blk, _);

    open IsTree(result, t1);
    TreeNodePtr lft = result->lft;
    assert IsTree(lft, l);
    red_has_branch(l);
    blacken_R(lft);
    blacken_R_applied_to_R(l);
    assert IsTree(lft, ?newl);
    close IsLLRB(Branch(newl, dk, dv, Red, r), Red, h);
```

```
        close IsTree(result, _);
        return result;
}
```

fix_RVRight



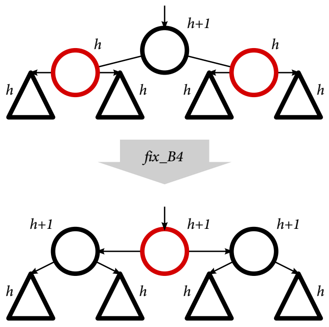
Argument: pointer n to node.
Obtain l and r from node subtree pointers.
l n r
v
L
R
h
B(h, L, lb, v)
R(h, R, v, hb)
Open the right subtree.
Set c to color of n.
Open sequence R = RL·[rv]·RR.
Obtain pointers rl and rr.
l n rl r rr c
v rv
L RL RR
h
B(h, L, lb, v)
B(h, RL, v, rv)
R(h, R, rv, hb)
Set right subtree pointer of n to rl.
Set n red.
l n rl r rr c
v rv
L RL RR
h
B(h, L, lb, v)
B(h, RL, v, rv)
R(h, R, rv, hb)
Close n subtree.
Discard l and rl pointers.
n r rr c
rv
L·[v]·RL
RR
h
R(h, L·[v]·RL, lb, rv)
R(h, R, rv, hb)
Set r to the color c.
Set left subtree of r to n.
n r rr c
rv
L·[v]·RL
RR
h
R(h, L·[v]·RL, lb, rv)
R(h, R, rv, hb)
Close r.
Discard c, and n and rr pointers.
r
(L·[v]·RL)·[rv]·RR
h
RV(h, (L·[v]·RL)·[rv]·RR, lb, hb)
By transitivity, and equality,
(L·[v]·RL)·[rv]·RR = L·[v]·R.
r
L·[v]·R
h
RV(h, L·[v]·R, lb, hb)

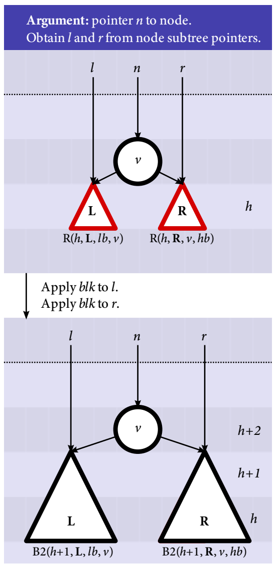


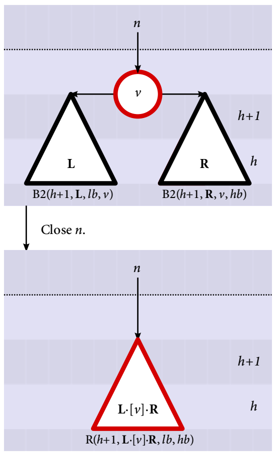
Like the *fix_B3Right* procedure, this flips the shape of the tree to place the red subtree on the left. This does not fix the tree — it is still a red-violated tree — but it is on the right side.

```
lemma void rotate_left_fixes_RVRight<K,V>(Tree<K,V> t0)
    requires RVRight(t0, ?h);
    ensures RV(rotate_left_fixpoint(t0), h);
{
    open RVRight(t0, h);
    open Branch(t0, ?l, ?k, ?v, Red, ?r);
    open IsLLRB(r, Red, h);
    close Branch(r, ?rl, _, _, _, _);
    close Branch(rotate_left_fixpoint(t0), _, _, _, Red, _);
    close IsLLRB(Branch(l, k, v, Red, rl), Red, h);
    close RV(rotate_left_fixpoint(t0), h);
    open Branch(r, rl, _, _, _, _);
}
```

fix_B4




Argument: pointer *n* to node.
Obtain *l* and *r* from node subtree pointers.
l
n
r
v
L
R
h
R(*h*, **L**, *lb*, *v*)
R(*h*, **R**, *v*, *hb*)
Apply *blk* to *l*.
Apply *blk* to *r*.
l
n
r
v
h+2
h+1
h
L
R
B2(*h*+1, **L**, *lb*, *v*)
B2(*h*+1, **R**, *v*, *hb*)


n
v
h+1
h
L
R
B2(*h*+1, **L**, *lb*, *v*)
B2(*h*+1, **R**, *v*, *hb*)
Close *n*.
n
h+1
h
L·[*v*]·R
R(*h*+1, **L**·[*v*]·**R**, *lb*, *hb*)

The B4 tree, with red on both subtrees, can be fixed by “ moving the red up” to the root. This is a pure recoloring and no rotations are necessary.

void $\mathit{fix\_B_4}(\mathit{TreeNodePtr}\ \mathit{root})$;
requires $\mathit{IsTree}(\mathit{root}, ?t_0) \ast B_4(t_0, ?h)$;
ensures $\mathit{IsTree}(\mathit{root}, ?t_1) \ast \mathit{IsLLRB}(t_1, \mathit{Red}, h) \ast \mathit{flatten}(t_1) == \mathit{flatten}(t_0)$;

void $\mathit{fix\_B_4}(\mathit{TreeNodePtr}\ \mathit{root})$
requires $\mathit{IsTree}(\mathit{root}, ?t_0) \ast B_4(t_0, ?h)$;
ensures $\mathit{IsTree}(\mathit{root}, ?t_1) \ast \mathit{IsLLRB}(t_1, \mathit{Red}, h) \ast \mathit{flatten}(t_1) == \mathit{flatten}(t_0)$;

```
{
    open B_4(t_0, h);
    open Branch(t_0, ?L, _, _, _, ?R);
    open IsTree(root, t_0);
    red_has_branch(L);
    red_has_branch(R);
    blacken_R_applied_to_R(L);
    blacken_R_applied_to_R(R);
    blacken_R(root->lft);
    blacken_R(root->rgt);
    root->color = Red;
    close IsTree(root, ?t_1);
    close IsLLRB(t_1, Red, h);
}
```

Algorithm

I have already described the insert algorithm informally. In this section, having verified the rotation and fixer procedures, I verify the mutually recursive insert algorithm.

Creating a new tree node

Before this, though, we need the ability to allocate a new node: if the key is not in the tree, then we insert a new node as a leaf.

```
TreeNodePtr newTreeNode(int key, int value)
requires emp;
ensures IsTree(result, ?t) ∗ IsLLRB(t, Red, 0) ∗ flatten(t) == Cons(Pair(key, value), Nil);
{
    TreeNodePtr result = malloc(sizeof(struct TreeNode));
    if (result == 0) { abort(); }
    result->key = key;
    result->value = value;
    result->lft = 0;
    result->rgt = 0;
    result->color = Red;
    close IsTree(result, Branch(Leaf, key, value, Red, Leaf));
    close IsLLRB(Leaf, Blk, 0);
    close IsLLRB(Leaf, Blk, 0);
    close IsLLRB(Branch(Leaf, key, value, Red, Leaf), Red, 0);
    return result;
}
```

Declarations and wrappers

The mutually recursive procedures are called *llrb_insert_R* , which inserts into a red-rooted tree, and *llrb_insert_B* , which inserts into a black-rooted tree. They have the same C signature: they take a key and value to insert, a pointer to the tree to insert into, and return a boolean, which indicates whether the tree needs fixing.

However, they have different VeriFast contracts. The contracts have a lot in common: they say that the tree pointer points into a tree and the tree is sorted, and that after insertion the tree is mutated to represent an index given by the *index_insert* fixpoint.

The difference lies in the *IsLLRB* predicate. These, together with the result boolean, describe how the tree colors can mutate. A red-rooted tree can become a red-violated tree, and a black-rooted tree can become a red-rooted tree.

```
bool llrb_insert_R(int key, int value, TreeNodePtr * rootptr);
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ sorted(flatten(t0)) == true ∗ IsLLRB(t0,
Red, ?h);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ (result ? RV(t1, h) : IsLLRB(t1, Red, h)) ∗
flatten(t1) == index_insert(key, value, flatten(t0));

bool llrb_insert_B(int key, int value, TreeNodePtr * rootptr);
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ sorted(flatten(t0)) == true ∗ IsLLRB(t0,
Blk, ?h);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ IsLLRB(t1, result ? Red : Blk, h) ∗
flatten(t1) == index_insert(key, value, flatten(t0));
```

The tree as a whole is black-rooted, meaning we start with *llrb_insert_B* . However, this contract has stuff we don't really care about, like the height of the tree. Additionally, we require a fix at the top of the tree after insertion: a returned red-rooted tree must be colored black. These two things are wrapped up in a “user-facing” procedure, *llrb_insert* :

```
void llrb_insert(int key, int value, TreeNodePtr * rootptr)
requires LLRB_BST(rootptr, ?t0, Blk, _);
ensures LLRB_BST(rootptr, ?t1, Blk, _) ∗ flatten(t1) == index_insert(key, value, flatten(t0));
{
    bool red = llrb_insert_B(key, value, rootptr);
    if (red) {
        open LLRB_BST(rootptr, ?t1, Red, ?h);
        open BST(rootptr, t1);
        red_is_branch(t1);
        blacken_R(*rootptr);
    }
}
```

The procedures proper

The procedure bodies contain quite a lot of VeriFast annotations. Most of these are just opening and closing predicates, and these are a bit tedious. The core algorithm has been described, and you should be able to follow it through the procedures.

The interesting bits are the lemma calls (the lemmas are to be defined). In short, we need to show VeriFast that inserting into a subtree (or replacing at the root) yields a new tree that represents the properly updated index. There is one lemma for each case: *index_insert_here* , *index_insert_left* , and *index_insert_right* .

```
bool llrb_insert_R(int key, int value, TreeNodePtr * rootptr)
```

```
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ sorted(flatten(t_0)) == true ∗ IsLLRB(t_0,
Red, ?h);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ (result ? RV(t_1, h) : IsLLRB(t_1, Red, h)) ∗
flatten(t_1) == index_insert(key, value, flatten(t_0));
{
    TreeNodePtr root = *rootptr;

    open IsLLRB(t_0, Red, h);
    open IsTree(root, t_0);

    int k = root->key;

    assert root->lft ↦ ?lft ∗ IsTree(lft, ?l_0) ∗ root->rgt ↦ ?rgt ∗ IsTree(rgt, ?r_0) ∗ root->value ↦
    ?v;

    sides_of_sorted_are_sorted(flatten(l_0), Cons(Pair(k, v), flatten(r_0)));
    if (key < k) {
        index_insert_left(key, value, flatten(l_0), k, v, flatten(r_0));

        bool fix = llrb_insert_B(key, value, &(root->lft));
        assert IsTree(root, ?t_1);

        if (fix) {
            close IsTree(root, t_1);
            close Branch(t_1, _, _, _, _, _);
            close RV(t_1, h);
            return true;
        }
        else {
            close IsTree(root, t_1);
            close IsLLRB(t_1, Red, h);
            return false;
        }
    }

    else if (k < key) {
        index_insert_right(key, value, flatten(l_0), k, v, flatten(r_0));
        bool fix = llrb_insert_B(key, value, &(root->rgt));
        assert IsTree(root, ?t_1);
        assert flatten(t_1) == index_insert(key, value, flatten(t_0));

        if (fix) {
            close IsTree(root, t_1);
            close Branch(t_1, _, _, _, _, _);
            close RVRight(t_1, h);
            RVRight_has_right_branch(t_1);
            rotate_left_fixes_RVRight(t_1);
            rotate_left_maintains_values(t_1);
            root = rotate_left(root);
            *rootptr = root;
```

```
                return true;
            }
            else {
                close IsTree(root, $t_1$);
                close IsLLRB($t_1$, Red, h);
                return false;
            }
        }

        else {
            root->value = value;
            index_insert_here(key, value, flatten($l_0$), k, v, flatten($r_0$));
            close IsTree(root, ?$t_1$);
            close IsLLRB($t_1$, Red, h);
            return false;
        }
}

bool llrb_insert_B(int key, int value, TreeNodePtr * rootptr)
requires pointer(rootptr, ?$root_0$) ∗ IsTree($root_0$, ?$t_0$) ∗ sorted(flatten($t_0$)) == true ∗ IsLLRB($t_0$,
Blk, ?h);
ensures pointer(rootptr, ?$root_1$) ∗ IsTree($root_1$, ?$t_1$) ∗ IsLLRB($t_1$, result ? Red : Blk, h) ∗
flatten($t_1$) == index_insert(key, value, flatten($t_0$));
{
        TreeNodePtr root = *rootptr;

        open IsLLRB($t_0$, Blk, h);
        open IsTree(root, $t_0$);

        if (root == 0) {
            root = newTreeNode(key, value);
            *rootptr = root;
            return true;
        }
        else {
            int k = root->key;

            assert root->lft ↦ ?lft ∗ IsTree(lft, ?$l_0$) ∗ root->rgt ↦ ?rgt ∗ IsTree(rgt, ?$r_0$) ∗
            root->value ↦ ?v;

            sides_of_sorted_are_sorted(flatten($l_0$), Cons(Pair(k, v), flatten($r_0$)));

            if (key == k) {
                root->value = value;
                index_insert_here(key, value, flatten($l_0$), k, v, flatten($r_0$));
                close IsTree(root, ?$t_1$);
                close IsLLRB($t_1$, Blk, h);
                return false;
            }
            else {
                has_black_root_LLRB($l_0$);
                if (has_black_root(root->lft)) {
```

```
        if (key < k) {
            index_insert_left(key, value, flatten(l_0), k, v, flatten(r_0));
            llrb_insert_B(key, value, &(root->lft));
            close IsTree(root, ?t_1);
            close IsLLRB(t_1, Blk, h);
        }
        else {
            index_insert_right(key, value, flatten(l_0), k, v, flatten(r_0));
            bool fix = llrb_insert_B(key, value, &(root->rgt));
            assert IsTree(root, ?t_1);
            if (fix) {
                close Branch(t_1, _, _, _, _, _);
                close B3Right(t_1, h);
                root = fix_B3Right(root);
                fix_B3Right_lemma(t_1);
                assert IsTree(root, ?t_2);
                open B_3(t_2, h);
                open Branch(t_2, _, _, _, _, _);
                *rootptr = root;
            }
            assert IsTree(root, ?t_3);
            close IsLLRB(t_3, Blk, h);
        }
        return false;
    }
    else {
        if (key < k) {
            index_insert_left(key, value, flatten(l_0), k, v, flatten(r_0));
            bool fix = llrb_insert_R(key, value, &(root->lft));
            assert IsTree(root, ?t_1);
            if (fix) {
                close Branch(t_1, _, _, _, _, _);
                close RVLeft(t_1, h);
                root = fix_RVLeft(root);
                *rootptr = root;
                return true;
            }
            else {
                close IsLLRB(t_1, Blk, h);
                close IsTree(root, t_1);
                return false;
            }
        }
        else {
            index_insert_right(key, value, flatten(l_0), k, v, flatten(r_0));
            bool fix = llrb_insert_B(key, value, &(root->rgt));
            assert IsTree(root, ?t_1);
            if (fix) {
                close Branch(t_1, _, _, _, _, _);
                close B_4(t_1, h);
```

```
                    fix_B4(root);
                    return true;
                }
                else {
                    close IsLLRB(t1, Blk, h);
                    close IsTree(root, t1);
                    return false;
                }
            }
        }
    }
  }
}
```

Index insertion lemmas

The three lemmas, previous introduced, follow. The proof for each takes the same structure. We recurse down the left part of the index until we reach the case where it is empty; this is our base case.

I take the first lemmas as an example for discussion. *index_insert_here* shows that replacing the root value when the root key matches yields a tree representing the index resulting from index insertion. At each point, we have to show that *left_side_of_sorted_is_bound* : that is, the key at the root is an upper bound on the sorted list that precedes it.

```
lemma void index_insert_here<V>(int nk, V nv, List<Pair<int,V> > L0, int k, V v,
List<Pair<int,V> > R0)
    requires nk == k ∧ sorted(append(L0, Cons(Pair(k,v), R0)));
    ensures index_insert(nk, nv, append(L0, Cons(Pair(k,v), R0))) == append(L0, Cons(Pair(k,nv),
    R0));
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch(l0) {
                case Pair(k0, v0):
                    left_side_of_sorted_is_bound(L0, k, v, R0);
                    index_insert_here(nk, nv, L1, k, v, R0);
            }
    }
}

lemma void index_insert_left<V>(int nk, V nv, List<Pair<int,V> > L0, int k, V v, List<Pair<int,V>
> R0)
    requires nk < k ∧ sorted(append(L0, Cons(Pair(k,v), R0)));
    ensures index_insert(nk, nv, append(L0, Cons(Pair(k,v), R0))) == append(index_insert(nk, nv,
    L0), Cons(Pair(k,v), R0));
{
    switch (L0) {
        case Nil:
```

```
        case Cons(l_0, L_1):
            switch (l_0) {
                case Pair(k_0, v_0):
                    if (k_0 < nk) index_insert_left(nk, nv, L_1, k, v, R_0);
            }
    }
}

lemma void index_insert_right<V>(int nk, V nv, List<Pair<int,V> > L_0, int k, V v,
List<Pair<int,V> > R_0)
    requires k < nk ∧ sorted(append(L_0, Cons(Pair(k,v), R_0)));
    ensures index_insert(nk, nv, append(L_0, Cons(Pair(k,v), R_0))) == append(L_0, Cons(Pair(k,v),
    index_insert(nk, nv, R_0)));
{
    switch (L_0) {
        case Nil:
        case Cons(l_0, L_1):
            switch (l_0) {
                case Pair(k_0, v_0):
                    left_side_of_sorted_is_bound(L_0, k, v, R_0);
                    if (k_0 < nk) index_insert_right(nk, nv, L_1, k, v, R_0);
            }
    }
}
```

Where the BST insert algorithm was directly recursive, I describe the LLRB insert algorithm as mutually recursive. It consists of two procedures, *llrb_insert_R* which inserts into a red-rooted tree and *llrb_insert_B* which inserts into a black-rooted tree.

The pieces of the contract could be wrapped up in neater predicates. However, this attempt to reduce redundancy is illusory: much more verbiage is then introduced by opening and closing those predicates when calls to those procedures are made. A better approach is to give the neater contract to an auxiliary procedure which then just calls our recursive procedures.

Remove the minimum

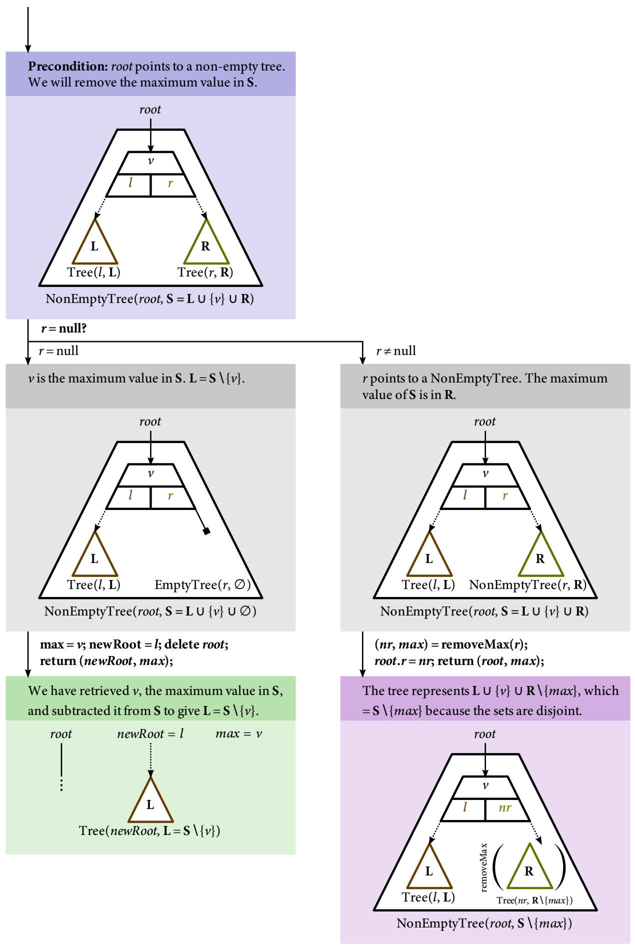
Description

The “remove the minimum”, or removeMin, operation takes a non-empty index to a tuple of two things: the key-value pair that has the minimal key in the index, and an index containing every key-value pair except that with the minimal key.

Unlike search, insert, and remove, I do not describe removeMin with a fixpoint. This is for two reasons. First, unlike those operations, it is restricted to non-empty indexes, which cannot be expressed in the VeriFast type system. Second, the operation is so simple that it does not really need a fixpoint to express it.

Remove the minimum from a BST

As before, I start out with a description of removeMin on BSTs. Removing the minimum from a leaf is an error. Consider removing the minimum from a branch. Where is the minimal key? It is certainly not in the right subtree, as all those keys are greater than the key at the root, by the definition of a BST. It is therefore either at the root or in the left subtree. If the left subtree is empty, then the minimum is at the root. In this case, we return the root key-value pair and the empty tree. Otherwise, it is in the left subtree. We then remove the minimum from that subtree, obtaining the minimum and a new subtree. As the minimum of the left subtree is the minimum of the whole tree, we simply return that as the minimum. The new tree consists of the new left subtree, the old root values, and the old right subtree.


Precondition: root points to a non-empty tree. We will remove the maximum value in S.
root
v
l
r
L
R
Tree(l, L)
Tree(r, R)
NonEmptyTree(root, S = L ∪ {v} ∪ R)
r = null?
r = null
r ≠ null
v is the maximum value in S. L = S \ {v}.
root
v
l
r
L
Tree(l, L)
EmptyTree(r, ∅)
NonEmptyTree(root, S = L ∪ {v} ∪ ∅)
r points to a NonEmptyTree. The maximum value of S is in R.
root
v
l
r
L
R
Tree(l, L)
NonEmptyTree(r, R)
NonEmptyTree(root, S = L ∪ {v} ∪ R)
max = v; newRoot = l; delete root; return (newRoot, max);
(nr, max) = removeMax(r); root.r = nr; return (root, max);
We have retrieved v, the maximum value in S, and subtracted it from S to give L = S \ {v}.
root
newRoot = l
max = v
L
Tree(newRoot, L = S \ {v})
The tree represents L ∪ {v} ∪ R \ {max}, which = S \ {max} because the sets are disjoint.
root
v
l
nr
L
R
removeMax
Tree(l, L)
Tree(nr, R \ {max})
NonEmptyTree(root, S \ {max})

Does this work for the LLRB? Notice that the minimum key is the final key on the left-hand spine of the tree. Removing it therefore decreases the length of that spine by one. If that node is colored red, we're OK: the black-height along that path remains the same, and no color violations are invoked by replacing it with a black leaf. However, the node may be colored black, in which case removing it violates the black-height. Therefore the BST algorithm is not sufficient for the LLRB.

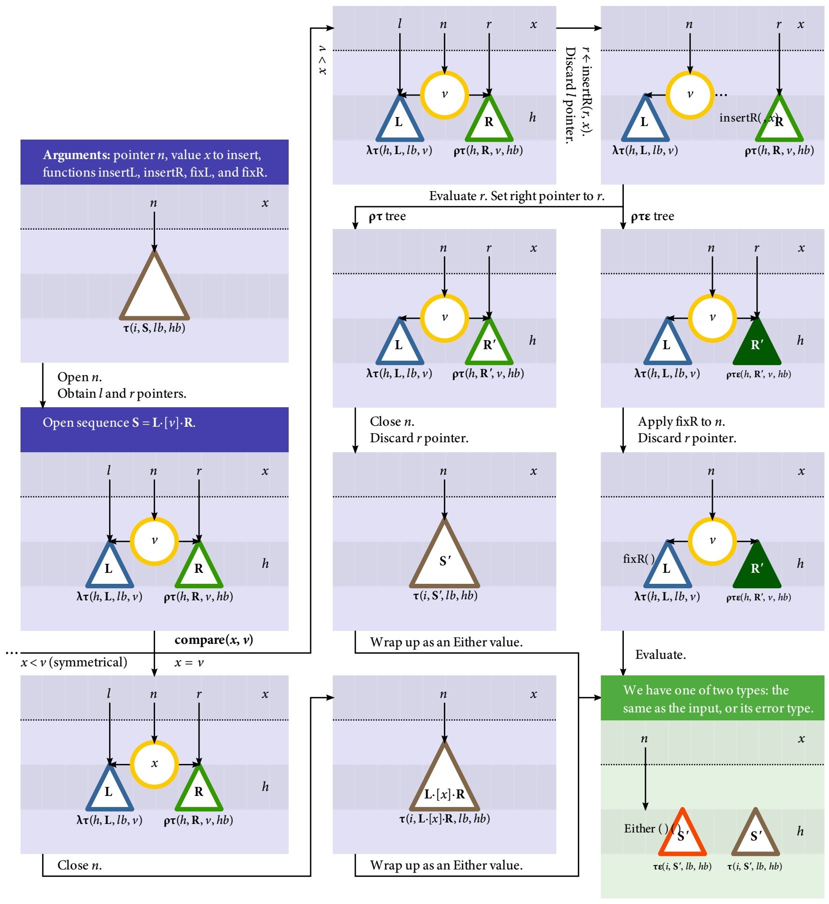
For removeMin on the LLRB, we take the same approach that we did with insert. We follow the BST algorithm until a point where the tree has to be changed. From there, we fix the tree using rotations and recolorings.

Let's think about the types of tree that removeMin must return. It is not possible for it to return the same tree type in all cases. The case of removing the minimum from a tree consisting of a single black node shows this. As with insert, notice how the brokenness of the tree is determined by the root color: removing from a red-rooted tree seems fine, but removing from a black-rooted tree breaks things. As before, we have different contracts for removal from red-rooted and black-rooted trees.

Notice a difference with the broken trees in insert: there, it was the red rule that was violated, while the black rule was maintained; here, the black rule is violated, and the red rule appears to be maintained. The goal is to fix a left subtree that is short by one black-height. This short left must be black, because if it were red, we could trivially fix it before ascending the tree, just by coloring the red node black. Therefore, all returned left subtrees that are broken are black-rooted with a black-height of one less than the input black-height.

As we now know the color and height of the broken left subtree, and we know the color and height of the right subtree because of the original LLRB invariants, we have only two cases to consider: whether the root is red or black. Whether we can fix the tree locally or instead defer to the parent depends on whether we can find a red node to sacrifice: the red gets colored black to fill in the missing black-height.

If the tree is red-rooted, we obviously have a red node to fix the tree; some rotation and recoloring fixes the tree and we're done. If if it is black-rooted, we continue the search. The right subtree is not a leaf, because it has a height greater than the broken left subtree. We can then look at its left subtree, and switch on whether it is red or black. If it is red, we color it black, do some rotation, and we're done. However, if it is black, we stop the search, do a rotation, and defer the fix.

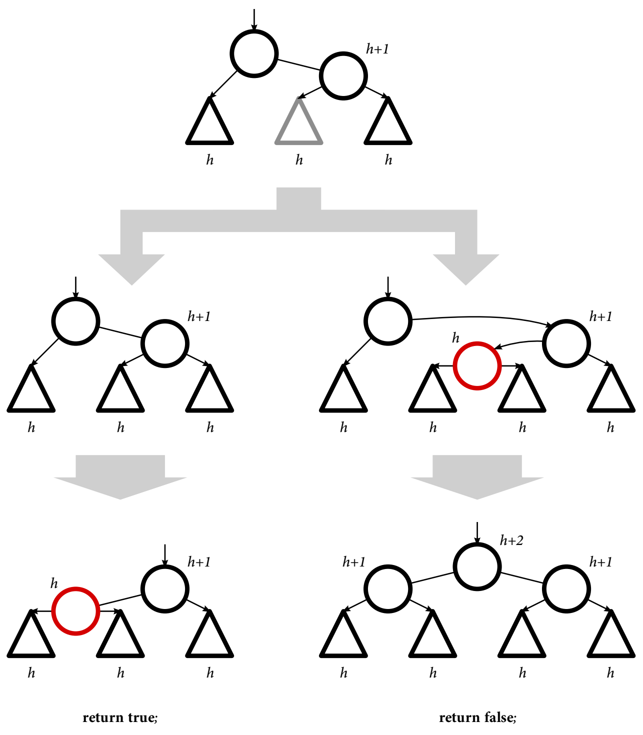


Verification

Verification of remove the minimum is a little easier than the other operations in one respect. The operation on indexes is so simple that we don't really need a section specifying it: the resulting index is simply the tail of the original index.

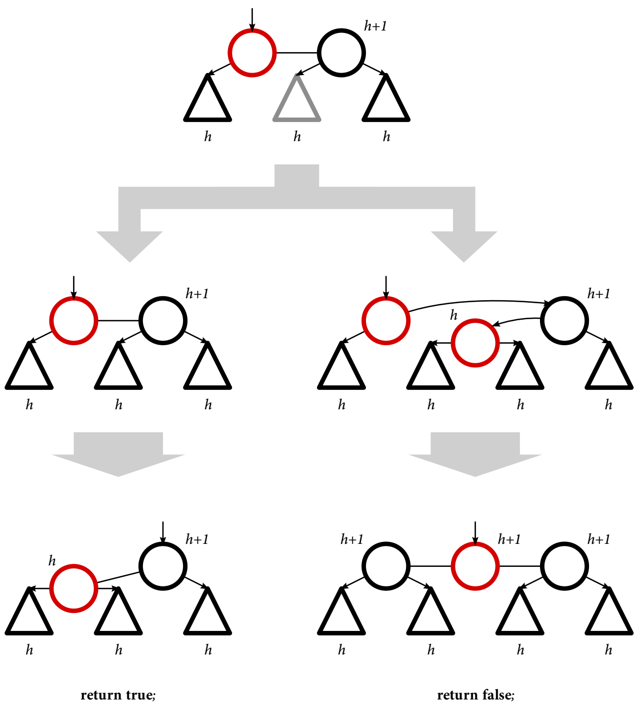
Fixers

Fixing a short left subtree (black)



```
bool fix_short_left_B(TreeNodePtr * rootptr);
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ Branch(t_0, ?l_0, ?k, ?v, Blk, ?r_0) ∗
IsLLRB(l_0, Blk, ?h_0) ∗ IsLLRB(r_0, Blk, h_0+1);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, Blk, (result ? h_0+1 : h_0+2)) ∗
flatten(t_1) == flatten(t_0);

bool fix_short_left_B(TreeNodePtr * rootptr)
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ Branch(t_0, ?l_0, ?k, ?v, Blk, ?r_0) ∗
IsLLRB(l_0, Blk, ?h_0) ∗ IsLLRB(r_0, Blk, h_0+1);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, Blk, (result ? h_0+1 : h_0+2)) ∗
flatten(t_1) == flatten(t_0);
{
    TreeNodePtr root = *rootptr;
    open Branch(t_0, l_0, k, v, Blk, r_0);
    open IsTree(root_0, t_0);

    TreeNodePtr rgt = root->rgt;

    height_gte_0(l_0);
    open IsLLRB(r_0, Blk, h_0+1);
    open IsTree(rgt, r_0);

    TreeNodePtr rgtlft = rgt->lft;
    assert IsTree(rgtlft, ?rl_0);

    has_black_root_LLRB(rl_0);
    if (has_black_root(rgtlft)) {
            root->color = Red;
            assert IsTree(root, ?t0_5);
            close Branch(t0_5, l_0, _, _, Red, r_0);
            close Branch(r_0, _, _, _, Blk, _);
            rotate_left_branches(t0_5);
            rotate_left_maintains_values(t0_5);
            root = rotate_left(root);
            assert IsTree(root, ?t_1);
            open Branch(t_1, ?l_1, _, _, Blk, _);
            open Branch(l_1, l_0, _, _, Red, _);

            close IsLLRB(l_1, Red, h_0);
            close IsLLRB(t_1, Blk, h_0+1);

            *rootptr = root;
            return true;
        }
        else {
            open IsTree(rgtlft, rl_0);
            open IsLLRB(rl_0, Red, h_0);
            assert IsTree(root, ?t_1);

            close Branch(t_1, l_0, k, v, Blk, r_0);
            close Branch(r_0, _, _, _, _, _);
            assert Branch(r_0, rl_0, _, _, Blk, _);
            close Branch(rl_0, _, _, _, Red, _);
            rotate_dbl_left_branches(t_1);
            rotate_dbl_left_maintains_values(t_1);
            root = rotate_dbl_left(root);
            assert IsTree(root, ?t_2);
            open Branch(t_2, ?l_2, _, _, Red, ?r_2);
            open Branch(l_2, l_0, _, _, Blk, _);
            open Branch(r_2, _, _, _, Blk, _);

            open IsTree(root, t_2);
            root->color = Blk;
            close IsTree(root, _);
            assert IsTree(root, ?t_3);

            close IsLLRB(l_2, Blk, h_0+1);
            close IsLLRB(r_2, Blk, h_0+1);
            close IsLLRB(t_3, Blk, h_0+2);
            *rootptr = root;
            return false;
        }
    }
```

Fixing a short left subtree (red)



```
bool fix_short_left_R(TreeNodePtr * rootptr);
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ Branch(t0, ?l0, ?k, ?v, Red, ?r0) ∗
IsLLRB(l0, Blk, ?h0) ∗ IsLLRB(r0, Blk, h0+1);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ IsLLRB(t1, (result ? Blk : Red), h0+1) ∗
flatten(t1) == flatten(t0);

bool fix_short_left_R(TreeNodePtr * rootptr)
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ Branch(t0, ?l0, ?k, ?v, Red, ?r0) ∗
IsLLRB(l0, Blk, ?h0) ∗ IsLLRB(r0, Blk, h0+1);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ IsLLRB(t1, (result ? Blk : Red), h0+1) ∗
flatten(t1) == flatten(t0);
{
    TreeNodePtr root = *rootptr;
    open Branch(t0, l0, k, v, Red, r0);
    open IsTree(root0, t0);

    TreeNodePtr rgt = root->rgt;

    height_gte_0(l0);
    open IsLLRB(r0, Blk, h0+1);
    open IsTree(rgt, r0);

    TreeNodePtr rgtlft = rgt->lft;
    assert IsTree(rgtlft, ?rl0);

    has_black_root_LLRB(rl0);
    if (has_black_root(rgtlft)) {
        close Branch(t_0, l_0, _, _, Red, r_0);
        close Branch(r_0, _, _, _, Blk, _);
        root = rotate_left(root);
        rotate_left_branches(t_0);
        rotate_left_maintains_values(t_0);
        assert IsTree(root, ?t_1);
        open Branch(t_1, ?l_1, _, _, Blk, _);
        open Branch(l_1, l_0, _, _, Red, _);

        close IsLLRB(l_1, Red, h_0);
        close IsLLRB(t_1, Blk, h_0+1);

        *rootptr = root;
        return true;
    }
    else {
        open IsTree(rgtlft, rl_0);
        open IsLLRB(rl_0, Red, h_0);
        root->color = Blk;
        assert IsTree(root, ?t_1);

        close Branch(t_1, l_0, k, v, Blk, r_0);
        close Branch(r_0, _, _, _, _, _);
        assert Branch(r_0, rl_0, _, _, Blk, _);
        close Branch(rl_0, _, _, _, Red, _);
        root = rotate_dbl_left(root);
        rotate_dbl_left_branches(t_1);
        rotate_dbl_left_maintains_values(t_1);
        assert IsTree(root, ?t_2);
        open Branch(t_2, ?l_2, _, _, Red, ?r_2);
        open Branch(l_2, l_0, _, _, Blk, _);
        open Branch(r_2, _, _, _, Blk, _);

        close IsLLRB(l_2, Blk, h_0+1);
        close IsLLRB(r_2, Blk, h_0+1);
        close IsLLRB(t_2, Red, h_0+1);
        *rootptr = root;
        return false;
    }
}
```

Freeing memory

Just as insertion had to allocate a new node, so removal possibly has to deallocate a node. In C, deallocation is done with the *free*() procedure call.

VeriFast implicitly gives *free*() one contract for each **struct** it knows about. One that it knows about is **struct** *TreeNode* . Recall the *malloc_block_TreeNode* predicate that we were given when we allocated the node? We require it here for deallocation: the predicate represents a permission to deallocate the memory.

I wrap up the call to *free*() in a procedure with an appropriate contract for the *TreeNode* type:

void *freeTreeNode*(*TreeNodePtr node*)
requires *node->key* ↦ _ ∗ *node->value* ↦ _ ∗ *node->color* ↦ _ ∗ *node->lft* ↦ _ ∗ *node->rgt* ↦ _ ∗ *malloc_block_TreeNode*(*node*);
ensures emp;
{
    free(*node*);
}

Recursive procedures

The C procedures, *removeMinR* and *removeMinB* , take three parameters and return a boolean. The first parameter is a pointer to the tree to remove the minimum from. The latter two parameters are pointers to allocated space for a key and a value; the procedure should put the minimal key-value pair in these slots. The return value of the procedure indicates whether there is fixing that needs doing; the meaning of this is different for the two procedures.

The precondition for *removeMin* needs to express that the index is non-empty, else removing the minimum is undefined. I do this with the *Cons* predicate I defined earlier, which says that a list uses the cons constructor. The postcondition must express that the values in the two slots represent the key-value pair, that the mutated tree represents the rest of the index, and finally must express the meaning of the returned boolean.

For removal from a red-rooted tree, the returned boolean indicates whether the new tree has become black-rooted. This is in fact not something that requires fixing: replacing a red tree with a black tree of the same height does not break either of the LLRB rules.

For removal from a black-rooted tree, the returned boolean indicates whether the new tree has dropped in height. This *is* something that needs fixing (and this is done with the two fixers discussed).

The contracts follow:

bool *removeMinR*(*TreeNodePtr* * *rootptr*, **int** * *key*, **int** * *value*);
requires pointer(*rootptr*, ?$root_0$) ∗ *IsTree*($root_0$, ?t_0) ∗ *IsLLRB*(t_0, *Red*, ?h_0) ∗ *Cons*(*flatten*(t_0), ?*kv*, ?*rest*) ∗ *Pair*(*kv*, ?*k*, ?*v*) ∗ **integer**(*key*, _) ∗ **integer**(*value*, _);
ensures pointer(*rootptr*, ?$root_1$) ∗ *IsTree*($root_1$, ?t_1) ∗ *IsLLRB*(t_1, (*result* ? *Blk* : *Red*), h_0) ∗ *flatten*(t_1) == *rest* ∗ **integer**(*key*, *k*) ∗ **integer**(*value*, *v*);

bool *removeMinB*(*TreeNodePtr* * *rootptr*, **int** * *key*, **int** * *value*);
requires pointer(*rootptr*, ?$root_0$) ∗ *IsTree*($root_0$, ?t_0) ∗ *IsLLRB*(t_0, *Blk*, ?h_0) ∗ $0 < h_0$ ∗ *Cons*(*flatten*(t_0), ?*kv*, ?*rest*) ∗ *Pair*(*kv*, ?*k*, ?*v*) ∗ **integer**(*key*, _) ∗ **integer**(*value*, _);
ensures pointer(*rootptr*, ?$root_1$) ∗ *IsTree*($root_1$, ?t_1) ∗ *IsLLRB*(t_1, *Blk*, (*result* ? h_0-1 : h_0)) ∗ *flatten*(t_1) == *rest* ∗ **integer**(*key*, *k*) ∗ **integer**(*value*, *v*);

bool *removeMinR*(*TreeNodePtr* * *rootptr*, **int** * *key*, **int** * *value*)
requires pointer(*rootptr*, ?$root_0$) ∗ *IsTree*($root_0$, ?t_0) ∗ *IsLLRB*(t_0, *Red*, ?h_0) ∗ *Cons*(*flatten*(t_0), ?*kv*, ?*rest*) ∗ *Pair*(*kv*, ?*k*, ?*v*) ∗ **integer**(*key*, _) ∗ **integer**(*value*, _);
ensures pointer(*rootptr*, ?$root_1$) ∗ *IsTree*($root_1$, ?t_1) ∗ *IsLLRB*(t_1, (*result* ? *Blk* : *Red*), h_0) ∗

```
flatten(t_1) == rest ∗ integer(key, k) ∗ integer(value, v);
{
    TreeNodePtr root = *rootptr;

    open IsLLRB(t_0, Red, h_0);
    open IsTree(root, t_0);
    open Cons(flatten(t_0), kv, rest);
    open Pair(kv, k, v);

    assert root->lft ↦ ?lft ∗ IsTree(lft, ?l_0) ∗ root->rgt ↦ ?rgt ∗ IsTree(rgt, ?r_0);

    if (root->lft == 0) {
        *key = root->key;
        *value = root->value;

        null_ptr_is_black_Leaf(root->lft);

        black_0_has_no_elements(l_0);
        black_0_has_no_elements(r_0);

        dispose_null_ptr(root->lft);
        dispose_black__0(root->rgt);
        freeTreeNode(root);
        *rootptr = 0;
        close IsTree(0, Leaf);
        close IsLLRB(Leaf, Blk, 0);

        return true;
    }
    else {
        black_non_null_has_height(root->lft);
        black_with_height_has_elements(l_0);

        close Cons(flatten(t_0), kv, rest);
        close Pair(kv, k, v);
        leftmost_of_left_is_leftmost_of_whole(flatten(t_0), flatten(l_0), root->key, root->value,
        flatten(r_0));
        removeMin_left_is_removeMin_root(flatten(t_0), flatten(l_0), root->key, root->value,
        flatten(r_0));

        bool fix = removeMinB(&(root->lft), key, value);

        assert IsTree(root, ?t_1);
        if (fix) {
            close Branch(t_1, _, _, _, Red, _);
            return fix_short_left_R(rootptr);
        }
        else {
            close IsTree(root, t_1);
            close IsLLRB(t_1, Red, h_0);
            return false;
        }
    }
}


bool removeMinB(TreeNodePtr * rootptr, int * key, int * value)
requires rootptr ↦ ?root_0 ∗ IsTree(root_0, ?t_0) ∗ IsLLRB(t_0, Blk, ?h_0) ∗ 0 < h_0 ∗ Cons(flatten(t_0),
?kv, ?rest) ∗ Pair(kv, ?k, ?v) ∗ integer(key, _) ∗ integer(value, _);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, Blk, (result ? h_0-1 : h_0)) ∗
flatten(t_1) == rest ∗ integer(key, k) ∗ integer(value, v);
{
    TreeNodePtr root = *rootptr;

    open IsLLRB(t_0, Blk, h_0);
    open IsTree(root, t_0);
    open Cons(flatten(t_0), kv, rest);
    open Pair(kv, k, v);

    assert root->lft ↦ ?lft ∗ IsTree(lft, ?l_0) ∗ root->rgt ↦ ?rgt ∗ IsTree(rgt, ?r_0);

    if (root->lft == 0) {
        *key = root->key;
        *value = root->value;

        null_ptr_is_black_Leaf(root->lft);

        black_0_has_no_elements(l_0);
        black_0_has_no_elements(r_0);

        dispose_null_ptr(root->lft);
        dispose_black_0(root->rgt);
        freeTreeNode(root);
        *rootptr = 0;
        close IsTree(0, Leaf);
        close IsLLRB(Leaf, Blk, 0);

        return true;
    }
    else {
        has_black_root_LLRB(l_0);
        if (has_black_root(root->lft)) {
            black_non_null_has_height(root->lft);
            black_with_height_has_elements(l_0);

            close Cons(flatten(t_0), kv, rest);
            close Pair(kv, k, v);

            leftmost_of_left_is_leftmost_of_whole(flatten(t_0), flatten(l_0), root->key, root->value,
            flatten(r_0));
            removeMin_left_is_removeMin_root(flatten(t_0), flatten(l_0), root->key, root->value,
            flatten(r_0));
            bool fix = removeMinB(&(root->lft), key, value);
            assert IsTree(root, ?t_1);
            if (fix) {
                close Branch(t_1, _, _, _, Blk, _);
                return fix_short_left_B(rootptr);
            }
            else {
                close IsLLRB(t_1, Blk, h_0);
                return false;
            }
        }
        else {
            red_has_elements(l_0);
            close Cons(flatten(t_0), kv, rest);
            close Pair(kv, k, v);

            leftmost_of_left_is_leftmost_of_whole(flatten(t_0), flatten(l_0), root->key, root->value,
            flatten(r_0));
            removeMin_left_is_removeMin_root(flatten(t_0), flatten(l_0), root->key, root->value,
            flatten(r_0));
            bool fix = removeMinR(&(root->lft), key, value);
            close IsTree(root, ?t_1);
            close Branch(t_1, _, _, _, _, _);
            b3_or_b_2(t_1, fix);
            return false;
        }
    }
}


lemma void leftmost_of_left_is_leftmost_of_whole<V>(List<Pair<int,V> > I_0, List<Pair<int,V> >
L_0, int k, V v, List<Pair<int,V> > R_0)
    requires I_0 == append(L_0, Cons(Pair(k,v), R_0)) ∗ Cons(I_0, ?e_0, ?I_1) ∗ Cons(L_0, ?l_0, ?L_1);
    ensures l_0 == e_0 ∗ Cons(I_0, e_0, I_1) ∗ Cons(L_0, l_0, L_1);
{
    open Cons(I_0, e_0, I_1);
    open Cons(L_0, l_0, L_1);
    close Cons(I_0, e_0, I_1);
    close Cons(L_0, l_0, L_1);
}

lemma void removeMin_left_is_removeMin_root<V>(List<Pair<int,V> > I_0, List<Pair<int,V> >
L_0, int k, V v, List<Pair<int,V> > R_0)
    requires Cons(I_0, ?e_0, ?I_1) ∗ I_0 == append(L_0, Cons(Pair(k,v), R_0)) ∗ Cons(L_0, e_0, ?L_1);
    ensures I_1 == append(L_1, Cons(Pair(k,v), R_0)) ∗ Cons(L_0, e_0, L_1);
{
    open Cons(I_0, e_0, I_1);
    open Cons(L_0, e_0, L_1);
    close Cons(L_0, e_0, L_1);
}
```

Remove

After *insert* , the second basic mutation function is *remove* . Together with *search* , these three functions form a traditional triple. (Remove the minimum is typically not considered as basic as these three — perhaps because it relies on an ordering, where the others, at the mathematical level, do not.)

Specification

Semantics of removal

As with insertion, I define the semantics of removal in terms of how it changes the behaviour of search. Let I_1 == *index_remove*(k, I_0) . Now consider searching for some *kn* in the new index I_1 . If *kn* == *k* , we just removed from the index at *k* , so *index_search*(*kn*, I_1) == *Nothing* . Otherwise, *kn* ≠ *k* . As with insertion, removal only touches on the index at the one key *k* , and so searching for *kn* should be unaffected. That is, *index_search*(*kn*, I_1) == *index_search*(*kn*, I_0) . Here then is our lemma that a *index_remove* algorithm must satisfy:

```
lemma void index_index_remove_proof<V>(int k, List<Pair<int,V> > L0, int kn);
    requires true;
    ensures index_search(kn, index_remove(k, L0)) == (kn == k ? Nothing : index_search(kn, L0));
```

A list removal algorithm

Our simple *index_remove* algorithm works like this. If the list is empty, removing anything from it also yields the empty list. Otherwise, the list consists of a pair (k_0,v_0) and the rest of the list I_1 . Let I_1' == *index_remove*(k, I_1) . If k == k_0 , *index_remove*(k, I_0) == I_1' . Else $k \neq k_0$.

```
fixpoint List<Pair<int,V> > index_remove<V>(int k, List<Pair<int,V> > I0) {
    switch (I0) {
        case Nil: return Nil;
        case Cons(e0, I1):
            return switch (e0) {
                case Pair(k0, v0):
                    return k == k0 ? index_remove(k, I1) : Cons(e0, index_remove(k, I1));
            };
    }
}
```

Verification of *index_remove*

This *index_remove* algorithm satisfies our remove semantics.

```
lemma void index_remove_proof<V>(int k, List<Pair<int,V> > L0, int kn)
    requires true;
    ensures index_search(kn, index_remove(k, L0)) == (kn == k ? Nothing : index_search(kn, L0));
```

```
{
    if (kn == k) {
        index_search_after_remove_is_Nothing(k, L_0);
    }
    else {
        index_search_unaffected_by_remove_non_search_key(k, L_0, kn);
    }
}


lemma void index_search_unaffected_by_remove_non_search_key<V>(int k, List<Pair<int,V> >
L_0, int kn)
    requires kn ≠ k;
    ensures index_search(kn, index_remove(k, L_0)) == index_search(kn, L_0);
{
    switch (L_0) {
        case Nil:
        case Cons(l_0, L_1):
            switch (l_0) {
                case Pair(k_0, v_0): index_search_unaffected_by_remove_non_search_key(k, L_1,
                kn);
            }
    }
}

lemma void index_search_after_remove_is_Nothing<V>(int k, List<Pair<int,V> > L_0)
    requires true;
    ensures index_search(k, index_remove(k, L_0)) == Nothing;
{
    switch (L_0) {
        case Nil:
        case Cons(l_0, L_1):
            switch (l_0) {
                case Pair(k_0, v_0): index_search_after_remove_is_Nothing(k, L_1);
            }
    }
}
```

Description

Removal from a BST

Removal, as with the other algorithms, has a descent phase followed by an ascent phase. The descent phase is the same as the search and insert descent phases: attempt to find the node with the key argument. If we don't find the key, we have no work to do: the key is removed. The ascent phase therefore does no work. If we do find the key, we have to remove it. The ascent phase then swaps out the old subtree for a new subtree with the key removed.

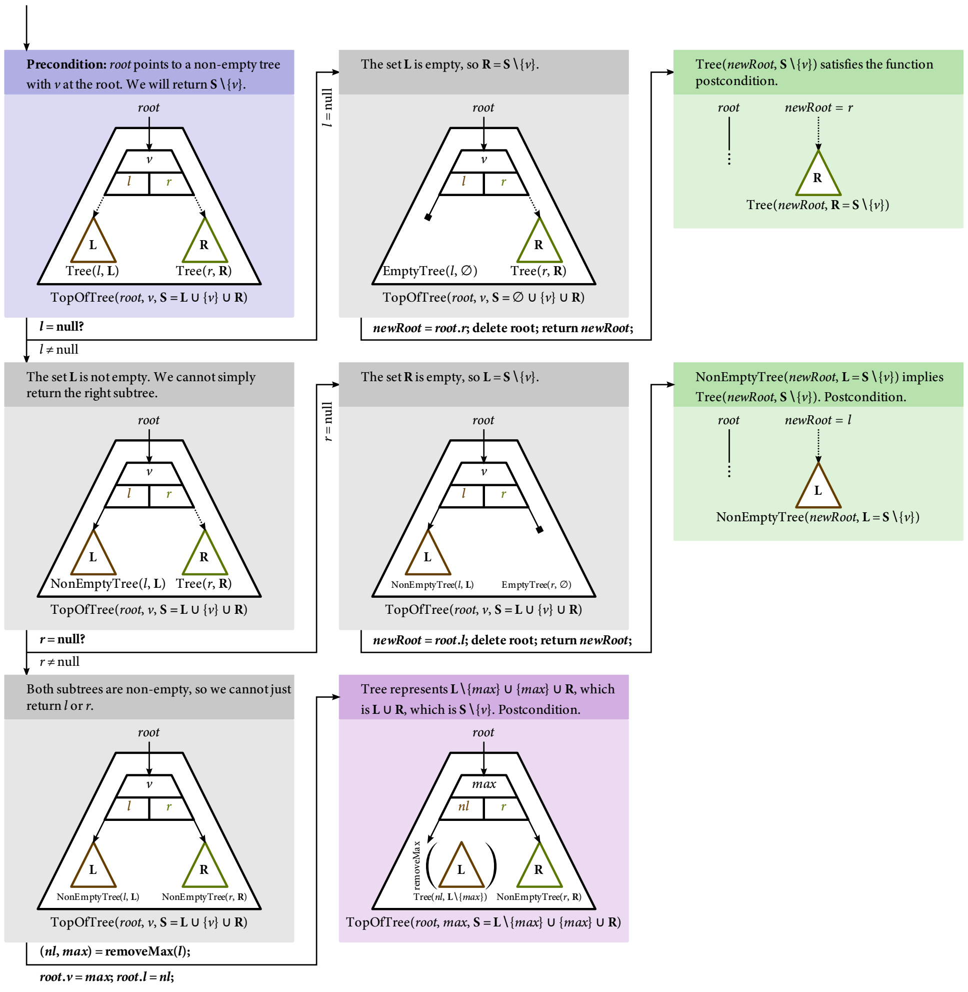
There is, however, an important algorithmic difference between insertion and removal. The insert algorithm, if it changes the tree, always descends to a leaf. Here, the base case of the algorithm, creating a new one-node subtree, is simple.

With removal, the opposite is true: if it changes the tree, the node it must remove is “in the middle” of the tree. This is more tricky: how do we stitch together the two subtrees without the root node?

If one of the subtrees is empty, we're in luck: the other subtree contains all the non-removed key-value pairs, so we just return that subtree.

If both are non-empty, it's harder. The fix *has* to be non-local, for the following reason. n nodes can only ever connect exactly n+1 subtrees: zero nodes can connect one subtree, and if you add another subtree, you need one more node to join them together. But we start with zero nodes and two subtrees. What is more, every time we open a non-empty subtree, we get one more node and one more subtree. We're always trying to use n nodes to stitch together n+2 subtrees. Therefore we have to keep opening subtrees until we get to an empty subtree, which reduces the number of subtrees by one and gives us no more nodes, leaving us with n nodes and n+1 subtrees. This is non-local.

The BST removal algorithm uses the following non-local fix: remove the minimum from the right subtree, then use this minimum to join the left subtree and the new right subtree.


Precondition: *root* points to a non-empty tree with v at the root. We will return $\mathbf{S} \setminus \{v\}$.

The set **L** is empty, so $\mathbf{R} = \mathbf{S} \setminus \{v\}$.

Tree(*newRoot*, $\mathbf{S} \setminus \{v\}$) satisfies the function postcondition.

The set **L** is not empty. We cannot simply return the right subtree.

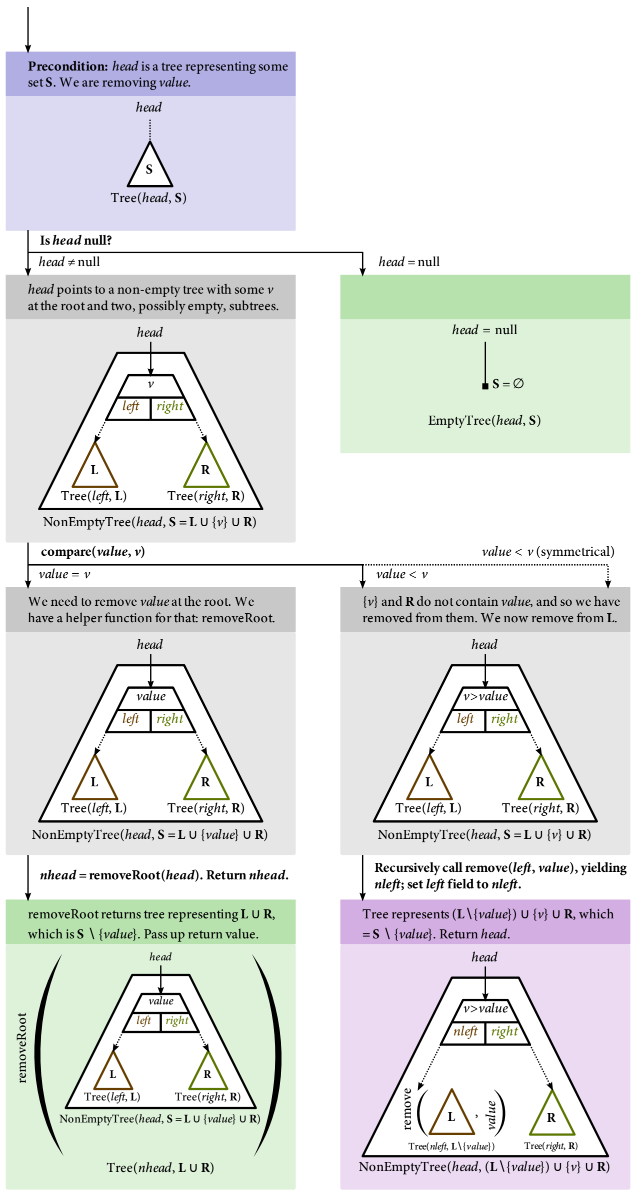
The set **R** is empty, so $\mathbf{L} = \mathbf{S} \setminus \{v\}$.

NonEmptyTree(*newRoot*, $\mathbf{L} = \mathbf{S} \setminus \{v\}$) implies Tree(*newRoot*, $\mathbf{S} \setminus \{v\}$). Postcondition.

Both subtrees are non-empty, so we cannot just return l or r.

Tree represents $\mathbf{L} \setminus \{max\} \cup \{max\} \cup \mathbf{R}$, which is $\mathbf{L} \cup \mathbf{R}$, which is $\mathbf{S} \setminus \{v\}$. Postcondition.

root
v
l
r
L
R
Tree(l, L)
Tree(r, R)
TopOfTree(root, v, S = L ∪ {v} ∪ R)
l = null?
l ≠ null
l = null
EmptyTree(l, ∅)
TopOfTree(root, v, S = ∅ ∪ {v} ∪ R)
newRoot = root.r; delete root; return newRoot;
newRoot = r
Tree(newRoot, R = S \ {v})
NonEmptyTree(l, L)
r = null?
r ≠ null
r = null
EmptyTree(r, ∅)
newRoot = root.l; delete root; return newRoot;
newRoot = l
NonEmptyTree(newRoot, L = S \ {v})
NonEmptyTree(r, R)
(nl, max) = removeMax(l);
root.v = max; root.l = nl;
max
nl
removeMax
Tree(nl, L \ {max})
TopOfTree(root, max, S = L \ {max} ∪ {max} ∪ R)


Precondition: *head* is a tree representing some set **S**. We are removing *value*.
head
S
Tree(*head*, **S**)
Is *head* null?
head ≠ null
head = null
head points to a non-empty tree with some *v* at the root and two, possibly empty, subtrees.
head
v
left
right
L
R
Tree(*left*, **L**)
Tree(*right*, **R**)
NonEmptyTree(*head*, **S** = **L** ∪ {*v*} ∪ **R**)
head = null
S = ∅
EmptyTree(*head*, **S**)
compare(*value*, *v*)
value < *v* (symmetrical)
value = *v*
value < *v*
We need to remove *value* at the root. We have a helper function for that: removeRoot.
head
value
left
right
L
R
Tree(*left*, **L**)
Tree(*right*, **R**)
NonEmptyTree(*head*, **S** = **L** ∪ {*value*} ∪ **R**)
{*v*} and **R** do not contain *value*, and so we have removed from them. We now remove from **L**.
head
v>*value*
left
right
L
R
Tree(*left*, **L**)
Tree(*right*, **R**)
NonEmptyTree(*head*, **S** = **L** ∪ {*v*} ∪ **R**)
nhead = removeRoot(*head*). Return *nhead*.
Recursively call remove(*left*, *value*), yielding *nleft*; set *left* field to *nleft*.
removeRoot returns tree representing **L** ∪ **R**, which is **S** \ {*value*}. Pass up return value.
removeRoot
head
value
left
right
L
R
Tree(*left*, **L**)
Tree(*right*, **R**)
NonEmptyTree(*head*, **S** = **L** ∪ {*value*} ∪ **R**)
Tree(*nhead*, **L** ∪ **R**)
Tree represents (**L** \ {*value*}) ∪ {*v*} ∪ **R**, which = **S** \ {*value*}. Return *head*.
head
v>*value*
nleft
right
remove
L
value
R
Tree(*nleft*, **L** \ {*value*})
Tree(*right*, **R**)
NonEmptyTree(*head*, (**L** \ {*value*}) ∪ {*v*} ∪ **R**)

Removal from an LLRB

As is familiar by now, the LLRB algorithm is the BST algorithm with an ascent phase that fixes the tree.

If you follow the introduction to removeMin, you will find that most of it applies to remove too. The brokenness of the returned tree is exactly the same: removing from a black-rooted tree might result in a black-rooted tree with one less black height.

This implies the fixing algorithms are going to be similar, too. When removing from the left subtree, this is indeed the case: as the postcondition of removing from the left subtree is the same as in removeMin, and the postcondition we have to satisfy for removal is the same too, we can follow exactly the same routine.

However, unlike removeMin, remove can be applied to a right subtree. In this case, we have exactly analogous fixers: *fix_short_right_R* and *fix_short_right_B* . The way they work isn't symmetrical to the short left fixers, due to the asymmetry of the tree constraints, but the contract is symmetrical.

Verification

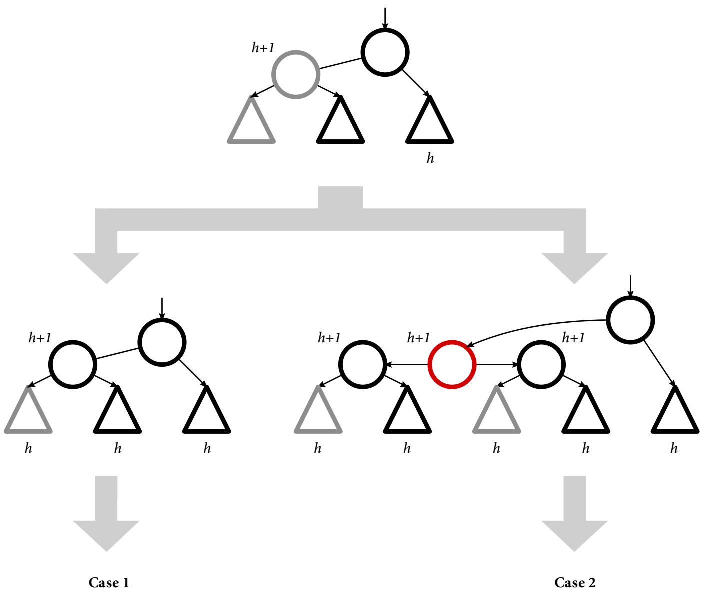
Fixers

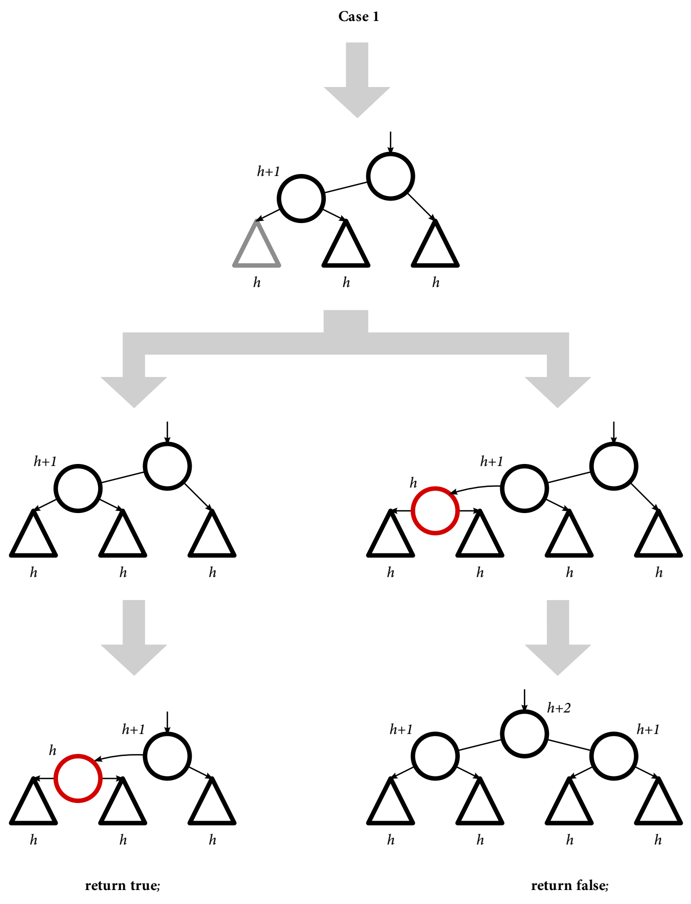
Fixing a short right subtree (red)

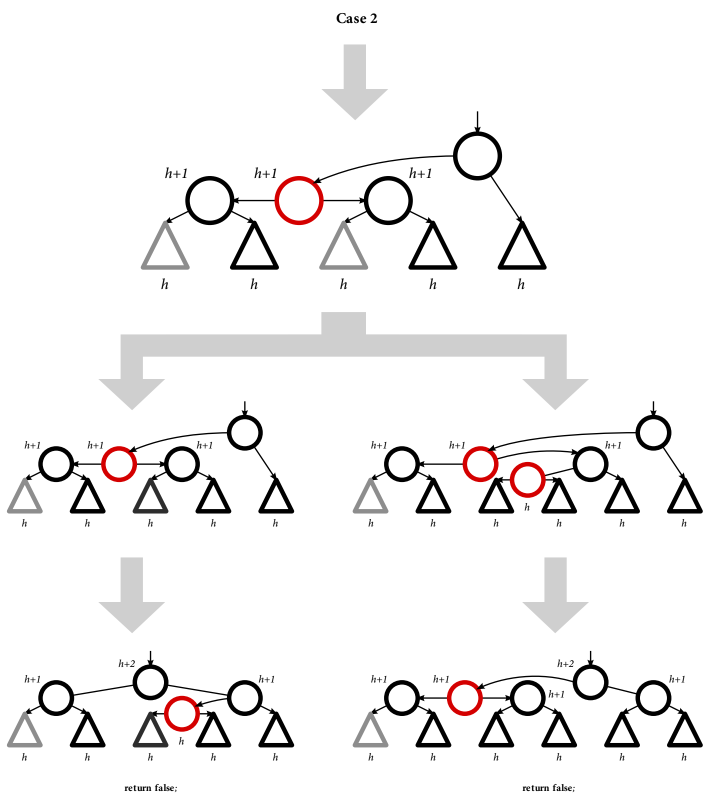
```
bool fix_short_right_R(TreeNodePtr * rootptr);
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ Branch(t_0, ?l_0, ?k, ?v, Red, ?r_0) ∗
IsLLRB(r_0, Blk, ?h_0) ∗ IsLLRB(l_0, Blk, h_0+1);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, (result ? Blk : Red), h_0+1) ∗
flatten(t_1) == flatten(t_0);

bool fix_short_right_R(TreeNodePtr * rootptr)
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ Branch(t_0, ?l_0, ?k, ?v, Red, ?r_0) ∗
IsLLRB(r_0, Blk, ?h_0) ∗ IsLLRB(l_0, Blk, h_0+1);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, (result ? Blk : Red), h_0+1) ∗
flatten(t_1) == flatten(t_0);
{
    TreeNodePtr root = *rootptr;
    open Branch(t_0, l_0, k, v, Red, r_0);
    open IsTree(root, t_0);

    height_gte_0(r_0);
    open IsLLRB(l_0, Blk, h_0+1);

    TreeNodePtr lft = root->lft;
    open IsTree(lft, l_0);

    TreeNodePtr lftlft = lft->lft;
    assert IsTree(lftlft, ?ll_0);

    has_black_root_LLRB(ll_0);
    if (has_black_root(lftlft)) {
        root->color = Blk;
        lft->color = Red;
        close IsTree(lft, _);
        assert IsTree(lft, ?l_1);
        close IsLLRB(l_1, Red, h_0);
        close IsTree(root, _);
        assert IsTree(root, ?t_1);
        close IsLLRB(t_1, Blk, h_0+1);
        return true;
    }
    else {
        open IsTree(lftlft, ll_0);
        open IsLLRB(ll_0, Red, h_0);
        close Branch(ll_0, _, _, _, Red, _);
        close Branch(l_0, ll_0, _, _, Blk, _);
        close Branch(t_0, l_0, _, _, Red, _);
        rotate_right_branches(t_0);
        rotate_right_maintains_values(t_0);
        root = rotate_right(root);
        assert IsTree(root, ?t_1);
        assert Branch(t_1, ?l_1, _, _, Blk, ?r_1);
        open Branch(l_1, _, _, _, Red, _);
        open Branch(r_1, _, _, _, Red, _);
        close IsLLRB(l_1, Red, h_0);
        close IsLLRB(r_1, Red, h_0);
        close B_4(t_1, h_0+1);
        fix_B_4(root);

        *rootptr = root;
        return false;
    }
}
```

Fixing a short right subtree (black)

Without a doubt, this is the hairiest part of the LLRB algorithms: it examines significantly more of the tree than all the other fixers we have seen do.




Case 1
h+1
h
h
h
h+1
h
h
h
h+1
h
h
h
h
h
h+1
h
h
h
h
h+2
h+1
h+1
h
h
h
h
return true;
return false;



```
bool fix_short_right_B(TreeNodePtr * rootptr);
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ Branch(t0, ?l0, ?k, ?v, Blk, ?r0) ∗
IsLLRB(l0, ?lc, ?h0) ∗ IsLLRB(r0, Blk, h0-1);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ IsLLRB(t1, Blk, (result ? h0 : h0+1)) ∗
flatten(t1) == flatten(t0);

bool fix_short_right_B(TreeNodePtr * rootptr)
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ Branch(t0, ?l0, ?k, ?v, Blk, ?r0) ∗
IsLLRB(l0, ?lc, ?h0) ∗ IsLLRB(r0, Blk, h0-1);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ IsLLRB(t1, Blk, (result ? h0 : h0+1)) ∗
flatten(t1) == flatten(t0);
{
    TreeNodePtr root = *rootptr;

    height_gte_0(r0);

    open Branch(t0, l0, k, v, Blk, r0);
    open IsTree(root, t0);
    TreeNodePtr lft = root->lft;

    has_black_root_LLRB(l0);
    if (has_black_root(lft)) {
        open IsLLRB(l0, Blk, h0);
        open IsTree(lft, l0);
        TreeNodePtr lftlft = lft->lft;
        assert IsTree(lftlft, ?ll_1);
        has_black_root_LLRB(ll_1);
        if (has_black_root(lftlft)) {
            lft->color = Red;
            close IsTree(lft, _);
            assert IsTree(lft, ?l_1);
            close IsLLRB(l_1, Red, h_0-1);
            close IsTree(root, _);
            assert IsTree(root, ?t_1);
            close IsLLRB(t_1, Blk, h_0);
            return true;
        }
        else {
            red_has_branch(ll_1);
            blacken_R_applied_to_R(ll_1);
            blacken_R(lftlft);
            close IsTree(lft, ?l_1);
            close IsTree(root, ?t_1);
            close Branch(l_1, _, _, _, Blk, _);
            close Branch(t_1, l_1, k, v, Blk, r_0);
            rotate_right_branches(t_1);
            rotate_right_maintains_values(t_1);
            root = rotate_right(root);
            assert IsTree(root, ?t_2);
            open Branch(t_2, _, _, _, Blk, ?r_2);
            open Branch(r_2, _, _, _, Blk, r_0);
            close IsLLRB(r_2, Blk, h_0);
            close IsLLRB(t_2, Blk, h_0+1);
            *rootptr = root;
            return false;
        }
    }
    else {
        open IsLLRB(l_0, Red, h_0);
        open IsTree(lft, l_0);
        TreeNodePtr lftrgt = lft->rgt;
        open IsTree(lftrgt, ?lr_0);
        open IsLLRB(lr_0, Blk, h_0);
        TreeNodePtr lftrgtlft = lftrgt->lft;
        assert IsTree(lftrgtlft, ?lrl_1);
        has_black_root_LLRB(lrl_1);
        if (has_black_root(lftrgtlft)) {
            lftrgt->color = Red;
            close IsTree(lftrgt, ?lr_1);
            close IsLLRB(lr_1, Red, h_0-1);
            lft->color = Blk;
            close IsTree(lft, ?l_1);
            close Branch(l_1, _, _, _, Blk, lr_1);
```

```
                close $IsTree(root, ?t_1)$;
                close $Branch(t_1, l_1, k, v, Blk, r_0)$;
                $rotate\_right\_branches(t_1)$;
                $rotate\_right\_maintains\_values(t_1)$;
                $root = rotate\_right(root)$;
                assert $IsTree(root, ?t_2)$;
                open $Branch(t_2, \_, \_, \_, Blk, ?r_2)$;
                open $Branch(r_2, lr_1, \_, \_, Blk, r_0)$;
                close $IsLLRB(r_2, Blk, h_0)$;
                close $IsLLRB(t_2, Blk, h_0+1)$;
                $*rootptr = root$;
                return false;
            }
            else {
                $red\_has\_branch(lrl_1)$;
                $blacken\_R\_applied\_to\_R(lrl_1)$;
                $blacken\_R(lftrgtlft)$;
                close $IsTree(lftrgt, ?lr_1)$;
                close $IsTree(lft, ?l_1)$;
                close $IsTree(root, ?t_1)$;
                close $Branch(lr_1, \_, \_, \_, Blk, \_)$;
                close $Branch(l_1, \_, \_, \_, Red, lr_1)$;
                close $Branch(t_1, l_1, \_, \_, Blk, r_0)$;
                $rotate\_dbl\_right\_branches(t_1)$;
                $rotate\_dbl\_right\_maintains\_values(t_1)$;
                $root = rotate\_dbl\_right(root)$;
                assert $IsTree(root, ?t_2)$;
                open $Branch(t_2, ?l_2, \_, \_, Blk, ?r_2)$;
                open $Branch(l_2, \_, \_, \_, Red, \_)$;
                open $Branch(r_2, \_, \_, \_, Blk, \_)$;
                close $IsLLRB(l_2, Red, h_0)$;
                close $IsLLRB(r_2, Blk, h_0)$;
                close $IsLLRB(t_2, Blk, h_0+1)$;
                $*rootptr = root$;
                return false;
            }
        }
}
```

Remove the root

Before getting to the remove function proper, I define another helper algorithm: remove the root. This abstracts away the "trick" of replacing the root with the minimum of the right subtree. It also takes complexity out of the remove function, which is large enough as it is.

bool *llrb_removeRoot_R*(*TreeNodePtr* * *rootptr*);
requires pointer$(rootptr, ?root_0) * IsTree(root_0, ?t_0) * IsLLRB(t_0, Red, ?h_0) * Branch(t_0, ?l_0, \_, \_, Red, ?r_0)$;
ensures pointer$(rootptr, ?root_1) * IsTree(root_1, ?t_1) * IsLLRB(t_1, (result\ ?\ Blk : Red), h_0) *$

```
flatten(t_1) == append(flatten(l_0), flatten(r_0));

bool llrb_removeRoot_R(TreeNodePtr * rootptr)
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ IsLLRB(t_0, Red, ?h_0) ∗ Branch(t_0, ?l_0, _, _,
Red, ?r_0);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, (result ? Blk : Red), h_0) ∗
flatten(t_1) == append(flatten(l_0), flatten(r_0));
{
    TreeNodePtr root = *rootptr;

    open IsLLRB(t_0, Red, h_0);
    open IsTree(root, t_0);
    open Branch(t_0, l_0, _, _, Red, r_0);

    if (root->rgt == 0) {
        null_ptr_is_black_Leaf(root->rgt);
        black_0_has_no_elements(r_0);

        dispose_null_ptr(root->rgt);

        TreeNodePtr lft = root->lft;
        *rootptr = root->lft;

        freeTreeNode(root);

        return true;
    }
    else {
        black_non_null_has_height(root->rgt);
        black_with_height_has_elements(r_0);
        open Cons(flatten(r_0), ?rmin, ?rrest);
        close Cons(flatten(r_0), rmin, rrest);
        close Pair(rmin, _, _);
        bool fix = removeMinB(&(root->rgt), &(root->key), &(root->value));

        assert IsTree(root, ?t_1);

        if (fix) {
            close Branch(t_1, _, _, _, Red, _);
            return fix_short_right_R(rootptr);
        }
        else {
            close IsLLRB(t_1, Red, h_0);
            return false;
        }
    }
}

bool llrb_removeRoot_B(TreeNodePtr * rootptr);
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ IsLLRB(t_0, Blk, ?h_0) ∗ 0 < h_0 ∗
Branch(t_0, ?l_0, _, _, Blk, ?r_0);
```

```
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, Blk, (result ? h_0-1 : h_0)) ∗
flatten(t_1) == append(flatten(l_0), flatten(r_0));

bool llrb_removeRoot_B(TreeNodePtr * rootptr)
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ IsLLRB(t_0, Blk, ?h_0) ∗ 0 < h_0 ∗
Branch(t_0, ?l_0, _, _, Blk, ?r_0);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, Blk, (result ? h_0-1 : h_0)) ∗
flatten(t_1) == append(flatten(l_0), flatten(r_0));
{
    TreeNodePtr root = *rootptr;

    open IsLLRB(t_0, Blk, h_0);
    open IsTree(root, t_0);
    open Branch(t_0, l_0, _, _, Blk, r_0);

    if (root->rgt == 0) {
        null_ptr_is_black_Leaf(root->rgt);
        black_0_has_no_elements(r_0);

        dispose_null_ptr(root->rgt);

        TreeNodePtr lft = root->lft;
        *rootptr = root->lft;

        freeTreeNode(root);

        /@ assert IsTree(lft, ?l0);
        has_black_root_LLRB(l_0);
        if (has_black_root(lft)) {
            return true;
        }
        else {
            red_has_branch(l_0);
            blacken_R_applied_to_R(l_0);
            blacken_R(lft);
            return false;
        }
    }
    else {
        black_non_null_has_height(root->rgt);
        black_with_height_has_elements(r_0);
        open Cons(flatten(r_0), ?rmin, ?rrest);
        close Cons(flatten(r_0), rmin, rrest);
        close Pair(rmin, _, _);
        bool fix = removeMinB(&(root->rgt), &(root->key), &(root->value));

        assert IsTree(root, ?t_1);

        if (fix) {
            close Branch(t_1, _, _, _, Blk, _);
            return fix_short_right_B(rootptr);
        }
        else {
            close IsLLRB(t_1, Blk, h_0);
            return false;
        }
    }
}
```

Notice that we treated stack variables like heap variables!

Remove

```
bool llrb_remove_R(TreeNodePtr * rootptr, int key);
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ sorted(flatten(t_0)) == true ∗ IsLLRB(t_0,
Red, ?h_0);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ sorted(flatten(t_1)) == true ∗ IsLLRB(t_1,
(result ? Blk : Red), h_0) ∗ flatten(t_1) == index_remove(key, flatten(t_0));

bool maybe_fix_left(TreeNodePtr * rootptr, bool fix)
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ Branch(t_0, ?l_0, _, _, Red, ?r_0) ∗ IsLLRB(l_0,
Blk, ?lh) ∗ IsLLRB(r_0, Blk, ?rh) ∗ (fix ? rh == lh + 1 : rh == lh);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ IsLLRB(t_1, (result ? Blk : Red), rh) ∗
flatten(t_1) == flatten(t_0);
{
    TreeNodePtr root = *rootptr;
    if (fix) {
        return fix_short_left_R(rootptr);
    }
    else {
        open Branch(t_0, l_0, _, _, Red, r_0);
        close IsLLRB(t_0, Red, rh);
        return false;
    }
}

bool llrb_remove_R(TreeNodePtr * rootptr, int key)
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ sorted(flatten(t_0)) == true ∗ IsLLRB(t_0,
Red, ?h_0);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ sorted(flatten(t_1)) == true ∗ IsLLRB(t_1,
(result ? Blk : Red), h_0) ∗ flatten(t_1) == index_remove(key, flatten(t_0));
{
    TreeNodePtr root = *rootptr;

    open IsTree(root, t_0);
    open IsLLRB(t_0, Red, h_0);

    int k = root->key;

    assert root->lft ↦ ?lft ∗ IsTree(lft, ?l_0) ∗ root->rgt ↦ ?rgt ∗ IsTree(rgt, ?r_0) ∗ root->value ↦
    ?v;
    sides_of_sorted_are_sorted(flatten(l0), Cons(Pair(k,v), flatten(r0)));
    remove_preserves_sorted(key, flatten(t0));

    if (key < k) {
        remove_left(key, flatten(l0), k, v, flatten(r0));
        bool fix = llrb_remove_B(&(root->lft), key);
        assert IsTree(root, ?t1);
        close Branch(t1, ?l1, k, _, Red, r0);

        return maybe_fix_left(rootptr, fix);
    }

    else if (k < key) {
        remove_right(key, flatten(l0), k, v, flatten(r0));
        bool fix = llrb_remove_B(&(root->rgt), key);
        assert IsTree(root, ?t1);
        if (fix) {
            close Branch(t1, _, _, _, _, _);
            return fix_short_right_R(rootptr);
        }
        else {
            close IsLLRB(t1, Red, h0);
            close IsTree(root, t1);
            return false;
        }
    }

    else {
        close IsLLRB(t0, Red, h0);
        close Branch(t0, l0, k, v, Red, r0);
        remove_root(flatten(l0), k, v, flatten(r0));
        return llrb_removeRoot_R(rootptr);
    }
}

bool llrb_remove_B(TreeNodePtr * rootptr, int key);
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ sorted(flatten(t0)) == true ∗ IsLLRB(t0,
Blk, ?h0);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ sorted(flatten(t1)) == true ∗ IsLLRB(t1,
Blk, (result ? h0-1 : h0)) ∗ flatten(t1) == index_remove(key, flatten(t0));

bool llrb_remove_B(TreeNodePtr * rootptr, int key)
requires pointer(rootptr, ?root0) ∗ IsTree(root0, ?t0) ∗ sorted(flatten(t0)) == true ∗ IsLLRB(t0,
Blk, ?h0);
ensures pointer(rootptr, ?root1) ∗ IsTree(root1, ?t1) ∗ sorted(flatten(t1)) == true ∗ IsLLRB(t1,
Blk, (result ? h0-1 : h0)) ∗ flatten(t1) == index_remove(key, flatten(t0));
{
    TreeNodePtr root = *rootptr;
    if (root == 0) {
        null_ptr_is_black_Leaf(root);
        black_0_has_no_elements(t_0);
        return false;
}
else {
        open IsTree(root, t_0);
        int k = root->key;

        assert root->lft ↦ ?lft ∗ IsTree(lft, ?l_0) ∗ root->rgt ↦ ?rgt ∗ IsTree(rgt, ?r_0) ∗
        root->value ↦ ?v;

        sides_of_sorted_are_sorted(flatten(l_0), Cons(Pair(k,v), flatten(r_0)));
        remove_preserves_sorted(key, flatten(t_0));

        open IsLLRB(t_0, Blk, h_0);
        if (key < k) {
                has_black_root_LLRB(l_0);
                if (has_black_root(root->lft)) {
                        remove_left(key, flatten(l_0), k, v, flatten(r_0));
                        bool fix = llrb_remove_B(&(root->lft), key);

                        assert IsTree(root, ?t_1);
                        if (fix) {
                                close Branch(t_1, _, _, _, Blk, _);
                                return fix_short_left_B(rootptr);
                        }
                        else {
                                close IsLLRB(t_1, Blk, h_0);
                                return false;
                        }
                }
                else {
                        remove_left(key, flatten(l_0), k, v, flatten(r_0));
                        bool fix = llrb_remove_R(&(root->lft), key);
                        close IsTree(root, ?t_1);
                        close Branch(t_1, _, _, _, _, _);
                        b3_or_b2(t_1, fix);
                        return false;
                }
        }

        else if (key > k) {
                remove_right(key, flatten(l_0), k, v, flatten(r_0));
                bool fix = llrb_remove_B(&(root->rgt), key);
                close IsTree(root, ?t_1);
                if (fix) {
                        close Branch(t_1, _, _, _, _, _);
                        return fix_short_right_B(rootptr);
                }
                else {
                        close IsLLRB(t_1, Blk, h_0);
                        return false;
                }
            }

            else {
                close IsLLRB($t_0$, Blk, $h_0$);
                black_non_null_has_height(root);
                close Branch($t_0$, _, _, _, _, _);
                remove_root(flatten($l_0$), k, v, flatten($r_0$));
                return llrb_removeRoot_B(rootptr);
            }
        }
}
```

Iteration

A final operation that I discuss is *iteration*. One advantage of an index that is not possible with a standard function is that the keys are known.

This is substantially different in flavour to the previous operations. The obvious difference is that it is *higher-order*: the procedure takes a pointer to a procedure as a parameter. The pointed-to procedure takes a key and value as arguments and returns nothing. The iteration function is supposed to walk through the elements of the index in order and for each, pass the key and value to the pointed-to procedure.

Specification

Full disclosure: this specification is closely modelled on a linked list map specification in the VeriFast examples. [20]

The following VeriFast pieces are fairly exotic. A **predicate_family** is effectively a function from C procedures to predicates. That is, called with a pointer to a C procedure, it evaluates to a predicate that behaves like any other predicate.

```
predicate_family iterator_post(void* func)(Pair<int,int> pair);

predicate_ctor iterator_post_ctor(iterator * f)(Pair<int,int> pair) = iterator_post(f)(pair);

typedef void iterator(int k, int v);
requires true;
ensures iterator_post(this)(Pair(k, v));
```

Verification

```
void llrb_iterate(iterator * it, TreeNodePtr root);
requires IsTree(root, ?T0) ∗ is_iterator(it) == true;
ensures IsTree(root, T0) ∗ foreach(flatten(T0), iterator_post_ctor(it));

void llrb_iterate_tree(iterator * it, TreeNodePtr root)
requires IsTree(root, ?T0) ∗ is_iterator(it) == true;
ensures IsTree(root, T0) ∗ foreach_Tree(T0, iterator_post_ctor(it));
{
    open IsTree(root, T0);

    if (root ≠ 0) {
        llrb_iterate_tree(it, root->lft);
        it(root->key, root->value);
        close iterator_post_ctor(it)(Pair(root->key, root->value));
        llrb_iterate_tree(it, root->rgt);
    }

    close IsTree(root, T0);
    close foreach_Tree(T0, iterator_post_ctor(it));
}

void llrb_iterate(iterator * it, TreeNodePtr root)
requires IsTree(root, ?T0) ∗ is_iterator(it) == true;
ensures IsTree(root, T0) ∗ foreach(flatten(T0), iterator_post_ctor(it));
{
    llrb_iterate_tree(it, root);
    foreach_tree_2_index(T0, iterator_post_ctor(it));
}

predicate foreach_Tree<K,V>(Tree<K,V> T, predicate(Pair<K,V>) p) =
    switch (T) {
        case Leaf: return true;
        case Branch(L,k,v,c,R): return foreach_Tree(L, p) ∗ p(Pair(k,v)) ∗ foreach_Tree(R, p);
    };

lemma void foreach_tree_2_index<K,V>(Tree<K,V> T0, predicate(Pair<K,V>) p)
    requires foreach_Tree(T0, p);
    ensures foreach(flatten(T0), p);
{
    open foreach_Tree(T0, p);
    switch (T0) {
        case Leaf:
            close foreach(flatten(T0), p);
        case Branch(L0,k,v,c,R0):
            foreach_tree_2_index(L0, p);
            foreach_tree_2_index(R0, p);
            close foreach(Cons(Pair(k,v), flatten(R0)), p);
            foreach_append(flatten(L0), Cons(Pair(k,v), flatten(R0)));
    }
}
```

Evaluation

With the exception of items in the appendix, the core contributions of my project have now been presented. I now evaluate the project. I do so in two stages. First, an evaluation of how well VeriFast performs as a tool for practical verification. Second, an evaluation of my case study, its successes, and its failures.

Evaluation of VeriFast

What cannot be verified?

The VeriFast specification language allows one to make much stronger assertions about a program than can, say, the C type system. However, there are limitations on what it can express.

Totality

VeriFast only checks the *partial* correctness of C procedures. This means that it does not check whether the procedure terminates. Rather, if VeriFast reports no errors, this means that *if* the procedure terminates *then* it terminates with a state satisfying the postcondition. We can write trivially non-terminating procedures like the following, with which VeriFast finds no error:

```
bool id(bool b)
requires emp;
ensures result == b;
{
    while (true)
    invariant emp;
    {
    }

    return b;
}
```

Similarly, we can write recursive procedures with no base case, with which VeriFast also finds no error:

```
bool list_search_nonterminating(int needle, int * value, ListTreeNode * listTreeNode)
requires IsList(listTreeNode, ?L) ∗ integer(value, ?v0);
ensures IsList(listTreeNode, L) ∗ integer(value, ?v1) ∗ IsMaybe(result, v1, index_search(needle,
L));
{
    return list_search_nonterminating(needle, value, listTreeNode);
}
```

How much of a drawback is this? I have found it surprisingly hard to write "interestingly" non-terminating procedures that satisfy non-trivial contracts. Yes, in general, we can, for example, insert a non-terminating loop anywhere, the invariant of which simply maintains the program state, or recurse without a base case.

More generally, VeriFast cannot express how long it takes for a procedure to terminate. We cannot, for example, have specify for *list_search* that *if* it terminates *then* it terminates in time linearly proportional to the length of the list.

Memory use

Nor does VeriFast check all correctness with respect to memory use. Analogous to the non-terminating procedure, which requires infinite time to produce a result, we might write a procedure that requires infinite memory in order to calculate its result.

More seriously, procedures that need to allocate new memory often do not have a defined behaviour in the event that they are unable to allocate new memory. What is the intended behaviour of *insert* where no memory is available for the new element? The typical C signature given to procedures like this (e.g., the insert procedures in Linux Red-Black tree implementations) do not make room for the procedure to indicate an error. Verifying the correctness of almost all real-world software in the presence of finite available memory would classify this as a bug. Moreover, this is a hard "bug" to correct. This is because memory failure is infectious: a procedure making use of a possibly-failing procedure is itself a possibly-failing procedure, except in the unusual case that it can fall back to a method of fulfilling its contract without allocating new memory.

We do not specify that a procedure does not modify heap chunks it has access to. We can only specify that, when the procedure terminates, it yields a heap chunk that is equivalent to that which it consumed, where equivalence is defined by the predicates and other constraints placed upon it in the contract. For example, our specification of `search` says that the procedure consumes an *IsTree*(*root*, T_0) and yields an *IsTree*(*root*, T_0) . This does not say that the heap chunks are equivalent when considered as a map from heap addresses to heap values. The procedure could, for example, destroy and rebuild the entire tree.

Purity

Many programming languages, famously Haskell, make the distinction between "pure" and "impure" functions: an impure function may do arbitrary input/output before returning its result, but a pure function is precisely that: a mathematical function of its parameters. This distinction is at the core of the language type system, and the usefulness of it has been exhaustively argued elsewhere. However, VeriFast offers no way to make such a basic assertion about C procedures. The assertions we can make are concerned with memory layout.

The C function `printf` should consume and produce a *permission* to print. This permission is a predicate. It can be *split*, meaning a lemma exists that requires the permission and produces two.

Limitations of fixpoints

Fixpoints cannot match on integers. This means that such a simple function as factorial cannot be written as a VeriFast fixpoint. VeriFast could have an algebraic definition of numbers that can be substituted for machine integers.

There are many functions that terminate that VeriFast cannot identify as terminating. For instance, the *index_union_fixpoint* fixpoint cannot be determined to terminate because an "inductive argument of a recursive call must be a switch clause pattern variable." That is, VeriFast's normal termination check, that there exists a switch clause pattern variable in the body which is used for all recursive calls, does not apply in this case because the function uses multiple variables for the recursive calls. One correct inductive parameter would be the sum of the lengths of the two parameters, but VeriFast cannot see this, and nor does it provide a mechanism to make the inductive parameter explicit.

Dependent types

VeriFast's **inductive** data types are "simple". They cannot be parameterized by anything but other types; we therefore cannot have indexed types. For example, we cannot express *balanced* trees as a VeriFast inductive data type, as we can, for example, with Haskell and GADTs. Instead we had to enforce balancedness manually: I chose to do this with a predicate, *IsLLRB* , which was included in the precondition to anything requiring the tree to be balanced. A particular ugliness caused by this is when defining fixpoints. In this project, I often had to define functions over balanced trees. As balanced was not expressed by the type, the domain of the function is larger than desired. For all these inputs, one has to return a dummy value. It is then necessary to use a lemma function to reduce the function's domain.

Fixpoint non-primitive termination

This makes it impractical to write many useful functions.

No switching on function calls.

Correspondence

Many times in the course of using VeriFast for this project, I had to contact Bart Jacobs, the principal developer of the program. I uncovered a number of (mostly trivial) bugs in it. I contacted Bart Jacobs frequently about these, and most were fixed quickly. I also had to contact Bart for help on undocumented or unclear features.

- **"Wrong ELF class" error in shared objects.** This bug is exhibited on 64-bit machines running the 32-bit VeriFast distribution. No fix has been made: VeriFast is tied to 32-bit because of its dependence on the 32-bit Z3 compiler binary.
- **Heap chunk pointer syntax is supported only for struct fields.** This syntax could also be used to express memory chunks that are not struct fields, like array elements or pointers to integers. As far as I am aware, this has not yet been extended.
- **No multi-level patterns for inductive datatype constructors.** Multi-level pattern-matching is apparently one of the most commonly requested features. I am told that this is in the works.
- **Multiple contracts for single procedure.** Some procedures in my project would have benefited from being able to assign multiple contracts to them: the workaround I chose was to use fixpoints instead and then assign "contracts" to the fixpoints using lemmas. Again, I am told this is in the works.
- **Lack of natural number data type.** It frustrated me having to use **int** in places where a *nat* type would have been more appropriate. In addition, VeriFast does not support induction over integers or naturals (though it would if given a Peano definition of the naturals.)
- **Displaying annotations in TeX.** This report required extensive inclusion of C with VeriFast annotations. Many of the extant papers on VeriFast did also, so I suspected that a lexer/parser was available that generated, say, TeX. This is not the case and all papers had hand-generated their code snippets. I instead wrote a lexer/parser in order to include my code in this report.
- **Existentials not introduced when closing a predicate.** Previously, introduced existentials in closing predicates, like **close** $Branch(t_1, As, b, bc, ?r_1)$, were ignored. This bug was fixed.
- **preprocessor interprets relative `#include` paths from directory of open file, not directory of preprocessed file.** This is a bug with respect to the C preprocessor semantics. It was fixed

promptly.

- **“ Body of fixpoint function must be switch statement or return statement” when appending semicolon to switch statement in fixpoint function.** This happens with things like

```
fixpoint bool foo(List<int> l) {
    switch (l) {
        case Nil: return true;
        case Cons(x,xs): return true;
    };
}
```

- **Request: call to boolean fixpoint in place of predicate.** As discussed, fixpoints and predicates are often interchangeable. One annoying syntactic disadvantage of the fixpoint is that without the “ == **true** suffix, a call to a fixpoint is indistinguishable from, and gets interpreted as, a predicate. ”
- **Request: top-level asserts.** Currently one can only have assertions inside procedures. There are some kinds of assertions (ones that are not concerned with memory) that make sense outside of this scope.
- **“ internal error: Not_found” when using array initializations without declaring the length explicitly.** I don't believe this has been fixed.
- **What does lemma_auto do? This was a feature I saw in the examples but which was not documented anywhere. (The keyword introduces a lemma that is automatically called by VeriFast. Chiefly for didactic reasons, I do not use them in this project.)**
- **C++.** I noted that VeriFast verifies subsets of C and Java. A subset of C++ would be valuable. Bart confirmed that this is a project that is going to be undertaken.
- **Internal error when applying function pointer in body of fixpoint.** I had accidentally attempted to use a C procedure inside a VeriFast fixpoint. This was fixed.
- **Stack overflow caused by cyclical `#includes`.** Where cycles exist in the `#include` graph, VeriFast overflows its stack. I do not believe this is fixed.
- **`#include` of same file using different filepaths causes duplicated inclusion.** Instead of using the header guards in the header file to determine the identity of a file, VeriFast uses the filepath. In general a file can be addressed in an infinite number of ways. VeriFast does not detect where a file is included twice by different paths and so it includes it twice, causing an error. I don't think this is fixed.
- **Parse error on header file with no contents.** This is not yet fixed.
- **`#include` treats absolute file paths as relative paths.** VeriFast was interpreting all paths as relative paths.
- **Internal error when including file two levels up the directory tree.** VeriFast was using an incorrect path canonicalization that cancels path segments of the form [e, “..”] to [], which cannot be applied where e=“..” or e=“.”.
- **`typedef` inductive types.** I wanted to write

```
typedef Index<K,V> = List<Pair<K,V> >;
```

  but this is not accepted by VeriFast.
- **VeriFast source.** VeriFast is closed-source, and I wanted access to the repository in order to fix various bugs. Bart gave me access (though I did not have the time or expertise to fix anything).
- **Broken “functions.c” example.** One of the standard examples that is distributed with VeriFast was broken; I fixed it.
- **Unproven `main` procedure precondition.** I thought VeriFast was allowing anything to be given as the precondition to the C `main` procedure. However, the precondition is checked at link time. This was found while trying to hack together a permissions system to verify procedure purity with respect to input/output.
- **Segmentation fault on attempt to verify.** This apparently is a 64-bit bug.

- **Self-include causes stack overflow.** This is really the same bug as the cyclical `#includes`, just with a tight cycle.
- **A "fact" construct which is essentially a predicate that represents emp .** I suggested making a distinction between predicates that hold heap chunks and those that don't. Not having this means that I have to write tedious lemmas and invoke them all the time: *e.g.* for $BalancedTree(T)$, I might have a lemma *BalancedTree_duplicate* which takes $BalancedTree(T)$ " to $BalancedTree(T) * BalancedTree(T)$, and a lemma *BalancedTree_emp* which takes a $BalancedTree(T)$ to **emp** . Bart tells me that this capability exists, but I did not have the time to investigate it.
- **VeriFast becomes extremely CPU-hungry after being used for a long time.** This is possibly a garbage collection issue with the IDE.

Annoyances

Here I document a few things that I have found annoying when using VeriFast. This is intended as constructive criticism based on an extensive case study!

No distinction between representation of heap chunks and representation of facts

Predicates in VeriFast have multiple uses. Among these are representation of heap chunks and representation of facts. The conflation of these two causes some ugliness. We often have to prove that a predicate used to express a fact does not contain any heap chunks: this is tedious. Lemmas that consume fact-predicates have to manually reconstruct them in order to produce them again; failure to do so means that users of the lemma lose access to the fact that makes the lemma precondition.

The language could make the distinction between *predicates* and *chunks*, where predicates can only refer to predicates, but chunks can refer to both predicates and chunks. Then we can:

- leak predicates
- duplicate predicates

No trouble-free way of representing facts in VeriFast

We have two options for representing facts in VeriFast: predicates and fixpoints. Both have serious deficiencies for practical use. Predicates are not distinguished from heap chunks and so suffer serious problems of practicality because they do not behave like facts. Fixpoints are difficult to work with because they do not operate over all necessary data types, the type system is not strong enough to enforce desirable constraints, they lack of pattern-matching, and we cannot use existential quantification in a fixpoint expression. VeriFast's error-reporting abilities with fixpoints is poor: one is frequently left with the unhelpful error, " Assertion might not hold: (= [large expression] [large expression])". Moreover, they do not interact well together.

This has been a primary source of agitation when using VeriFast, and I lost many days to it.

(At a late stage in this project, Bart Jacobs informed me that there are ways of representing heapless predicates, but I have not experimented with them.)

Unnatural C code

In some places I was forced to write C that felt un-natural or un-idiomatic. The *freeTreeNode* procedure would ordinarily just be a call to *free* , but we have to wrap it in another procedure in order to give it a sensible contract.

In iterative algorithms, I had to declare variables that I would not have to if I were just writing C. (See the iterative search procedure.)

No multi-level pattern-matching

A significant proportion of the code necessary in lemmas was due to the lack of multi-level pattern-matching. For instance, instead of writing:

```
lemma void lb_not_mem<V>(int lb, List<Pair<int,V> > I0)
    requires lowerBound(lb, I0) == true;
    ensures index_search(lb, I0) == Nothing;
{
    switch (I0) {
        case Nil:
        case Cons(e0, I1):
            switch (e0) {
                case Pair(k0, v0): lb_not_mem(lb, I1);
            }
    }
}
```

we could instead write:

```
lemma void lb_not_mem<V>(int lb, List<Pair<int,V> > I0)
    requires lowerBound(lb, I0) == true;
    ensures index_search(lb, I0) == Nothing;
{
    switch (I0) {
        case Nil:
        case Cons(Pair(k0, v0), I1): lb_not_mem(lb, I1);
    }
}
```

Preconditions on fixpoints

My rotation fixpoints, for instance, only make sense when the tree they are rotating is of a certain shape. I had to express this separately in a lemma.

Fixpoint constraints on data types

The index specification represented an index as a list of pairs. Part of this representation is that the list is sorted, and I represented this fact with a fixpoint. However, the constraint " all indexes are sorted" had to be carried around separately. It would be nice if, somehow, it could be attached to the data type.

Vacuuous truths

The following verifies with no errors:

```
int returnsFive()
requires 1 == 2;
ensures result == 5;
{
    return 4;
}
```

This might seem surprising. The reason is that VeriFast does not find any consistent paths through the procedure — in this case, because the precondition contains an obvious falsity.

Lack of documentation

As an evolving project, the documentation and tutorial for VeriFast are significantly behind the current version. Therefore, I had to contact Bart Jacobs about features where the documentations was lacking. While fine for this project, this is not how equivalent work in industry could proceed. This said, the subset of VeriFast that is documented is documented well.

Documentation

The documentation and tutorial for VeriFast are significantly behind the current version. Therefore, I had to contact Bart Jacobs about features where the documentations was lacking.

Closed-source

VeriFast is closed-source. This runs contrary to the spirit of verification projects, which work closely with source code. More critically, the soundness of a verification tool is crucial, and this can only be determined by detailed examination of its source. In other words, a verification tool must itself be verified.

Evaluation of the project

I have successfully verified the core index operations on a balanced tree, within the bounds of what VeriFast allows one to verify. The project, however, was not without mistakes, and there are avenues for correction and expansion.

Mistakes

While some mistakes are evident in the report, inevitably, it is a cleaned-up and artificial description of how I went about coding and verifying the LLRB. This means that it does not make clear the avenues I explored but which failed. Nor does it make clear the methodology one should use when approaching a similar task.

Putting everything in one predicate

This report presented the pieces of the verification puzzle in many chunks: Trees, fixpoints, predicates. When I first started on this project, I had one large predicate that expressed the memory layout, the LLRB structure, and the sortedness.

For a start, this is unwieldy. It does not allow one to unfold only the necessary pieces of information; it is all-or-nothing. It prohibits us from reasoning about structures like trees independent of memory.

I gradually tried to move towards a style where the procedure was divorced from the algorithm, which was instead represented functionally. While "going the whole hog" with this approach was also a mistake, it is a step in the right direction.

Doing everything at once

Similarly, a key mistake was that I tried to verify too much at once: memory allocation correctness, key ordering correctness, red-rule correctness, black-rule correctness, and so on.

A better approach would have been to work from the least to most challenging. A basic first verification might just show that the algorithms do not access unallocated memory and do not leak memory. I might then have looked at the red rule. This rule can be enforced directly using VeriFast's ADTs (see the Further Work section). I would then have attempted the black rule, then the sortedness of the tree, and only finally the compliance with the index search specification.

I originally worked "bottom-up" on the verification in that I tried to start with the most fundamental results, for example, the lemmas on appending sorted lists. This was the wrong approach. Intuition works much better top-down. I should have started with the result to prove, just writing the lemmas it depends on, and only approaching those once I had finished.

(I did actually encounter a single lemma that I spent a long time attempting to prove, but was in fact incorrect. I was attempting to show that, given two lists A and B such that A ++ B is sorted, a lower bound b on the list A must also be a lower bound on the list B . The problem with this is that A can be empty, in which case its lower bound is unrestricted!)

This is how verification would proceed in a practical environment. The low-hanging fruit is picked first. VeriFast is a good tool in that it lets one do that.

Excessive use of fixpoints

My attempt to use fixpoints to describe as much as possible of the behaviour of the algorithms was not a success. While this approach works, it results in much extra verbiage, and is much more difficult to verify.

Many things that in idiomatic C would just be a single command must be raised to a whole procedure for practical verification. The *blacken_R* procedure is one example.

Multiple equivalent definitions

When I defined BSTs, I used a "naturalistic" definition of sortedness: that all the keys in the left tree are less than the root, and all the keys in the right tree are greater than the root, and the subtrees are sorted too. However, I also used another definition of sortedness when treating insert and remove: that, when the tree is flattened to a list, all the keys are sorted.

These are equivalent definitions. By using both, I had to show VeriFast that they are equivalent. Other than being an interesting exercise, this was not worth it: the verification is enough work as it is without adding extra definitions and lemmas to prove their equivalence. If I was to do this again, I would avoid the first definition in favour of simply saying that the flattened tree is sorted.

Mis-specifying iterate

The specification for the *llrb_iterate* procedure is limiting. Typically, one would like to iterate over an index and do arbitrary things with the elements in the iterator. For example, we might iterate through the elements, filtering them, and inserting the filtered elements into a new index. Such behaviour is prohibited by the specification as written, because the iterator function does not have sufficient resource.

Further work

Here I offer a number of ways in which this project could be extended.

Iterative implementations

For search, I showed how an iterative implementation of BST search could be verified in VeriFast. All other operations in this report used recursive procedures.

Writing and verifying iterative procedures for these would be interesting. This is somewhat different to the search procedure because the iterative algorithms for insert, remove and so on must be markedly different: they have a second phase in which they ascend the tree.

Where recursion takes advantage of the stack to keep a record of the path travelled to the node, an iterative algorithm must keep an explicit stack. There are two standard ways of doing this: firstly, allocating a stack (e.g. a linked list of pointers into nodes), and secondly, reversing the pointers in the tree as one descends it so that they can be followed back up.

The second technique is more interesting. In effect, the tree context shape is realized with pointers. A similar tree context predicate should be definable in which the pointers go up the path rather than down.

Express the red rule in the algebraic type

The red rule is the simpler of the two LLRB rules to verify. Trees that obey this rule can be defined using the standard ADTs available in standard functional languages. VeriFast is no different.

```
inductive LLRBTree<K,V> =
    | Leaf
    | Black1(RedTree<K,V>, K, V, LLRBTree<K,V>)
    | Black2(LLRBTree<K,V>, K, V, LLRBTree<K,V>)
    ;

inductive RedTree<K,V> =
    | Red(LLRBTree<K,V>, K, V, LLRBTree<K,V>)
    ;

predicate BlackHeightB<K,V>(LLRBTree<K,V> t, int h) =
    switch (t) {
        case Leaf: return h == 0;
        case Black1(l,k,v,r): return BlackHeightR(l, h-1) ∗ BlackHeightB(r, h-1);
        case Black2(l,k,v,r): return BlackHeightB(l, h-1) ∗ BlackHeightB(r, h-1);
    };
```

```
predicate BlackHeightR<K,V>(RedTree<K,V> t, int h) =
    switch (t) {
        case Red(l,k,v,r): return BlackHeightB(l, h) * BlackHeightB(r, h);
    };
```

This definition *may* have saved considerable complexity. It also may not have: for example, I would have to define separate predicates, *IsLLRBTree* and *IsRedTree* , one for each type.

More tree operations

I have a number of operations on the index specification which were not implemented for LLRBs. Finding, implementing and verifying algorithms for these over LLRBs is an obvious candidate for further work. These operations include:

```
fixpoint List<Pair<K,V2> > index_mapValues_fixpoint<K,V1,V2>(fixpoint(V1,V2) f,
List<Pair<K,V1> > I0) {
    switch (I0) {
        case Nil: return Nil;
        case Cons(e0, I1):
            return switch (e0) {
                case Pair(k0,v0): return Cons(Pair(k0, f(v0)), index_mapValues_fixpoint(f, I1));
            };
    }
}

lemma void index_mapValues_proof<K,V1,V2>(K k, List<Pair<K,V1> > I0, fixpoint(V1,V2) f)
    requires true;
    ensures index_search(k, index_mapValues_fixpoint(f, I0)) == maybe_map(f, index_search(k,
    I0));
{
    switch (I0) {
        case Nil:
        case Cons(e0, I1):
            switch (e0) {
                case Pair(k0,v0): index_mapValues_proof(k, I1, f);
            }
    }
}

fixpoint List<Pair<int,V> > index_union_fixpoint<V>(List<Pair<int,V> > IA, List<Pair<int,V> >
IB)
{
    switch (IA) {
        case Nil: return IB;
        case Cons(akv0, IA1):
            return switch (akv0) {
                case Pair(ak0, av0):
                    return switch (IB) {
                        case Nil: return Nil;
                        case Cons(bkv0, IB1):
```

```
                    return switch (bkv0) {
                        case Pair(bk0, bv0):
                            return ak0 ≤ bk0
                                ? Cons(akv0, index_union_fixpoint(IA1, IB))
                                : Cons(bkv0, index_union_fixpoint(IB1, IA))
                                ;
                    };
                };
            };
    }
}

fixpoint Maybe<T> either<T>(Maybe<T> m0, Maybe<T> m1)
{
    switch (m0) {
        case Just(t): return Just(t);
        case Nothing: return m1;
    }
}

lemma void index_union_proof<K,V>(K kn, List<Pair<K,V> > I0, List<Pair<K,V> > I1)
    requires true;
    ensures index_search(kn, index_union_fixpoint(I0, I1)) == either(index_search(kn, I0),
    index_search(kn, I1));
{
}
```

Similar data structures

Other data structures exist that are similar in spirit to the LLRB. The most obvious is the classic Red-Black tree. Others include the AA tree and, to a lesser extent, the AVL tree.

It would be interesting to see how much these have in common with the LLRB when verifying them. Can most of the lemmas be re-used? Are any different techniques required to verify them?

Conclusion

This project was motivated by a desire to assess the practicability of machine-assisted program verification. This assessment was to be made by a case study using a particular proof assistant on a particular implementation of a particular data structure for a particular abstract data type.

Each of these particulars was well-chosen. Indexes were appropriate as a well-understood and fundamental data type. The LLRB was chosen for its performance characteristics that make it suitable for an industry index implementation and for its level of complexity representative of most index algorithms. My implementation of an LLRB was written for clarity and verifiability without sacrificing idiomatic style. VeriFast was the front-runner in the particular style of verification appropriate for this test case.

The initial goal — to verify my implementation to the limits of what VeriFast can verify — was broadly successful. Those limits are quite loose, yet there are things about my program that I should like to verify about it but which VeriFast does not currently do. This includes I/O purity, termination-checking, and time and space complexity. There were also failures and mistakes in my attempts to use VeriFast to its full. As a case study, these are useful: they provide a catalog of errors made by someone new to verification. There is considerable potential for further work on this project, both correction and expansion.

End notes

1. Wikipedia; *Lookup table* (2012). From http://en.wikipedia.org/wiki/Lookup_table.
2. Wikipedia; *Splay Tree* (2012). From http://en.wikipedia.org/wiki/Splay_tree.
3. Adrian Hey; *Data.Tree.AVL* (2004,2008). From http://hackage.haskell.org/packages/archive/AvlTree/4.2/doc/html/src/Data-Tree-AVL.html. , the most-documented and -commented AVL tree source I know of. Wikipedia; *AVL Tree* (2012). From http://en.wikipedia.org/wiki/AVL_Tree. .
4. Wikipedia; *K-ary Tree* (2012). From http://en.wikipedia.org/wiki/K-ary_tree.
5. GNU authors; *stl_tree.h* (2005). From http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl__tree_8h-source.html. Algorithms are based on those in CLR.
6. Oracle; *TreeMap.java* (2011). From http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html. Algorithms are based on those in CLR.
7. Raja R Harinath; *RBTree.cs* (2007). From https://github.com/mosa/Mono-Class-Libraries/blob/master/mcs/class/System/System.Collections.Generic/RBTree.cs.
8. Andrea Arcangeli, David Woodhouse; *rbtree.c* (1999, 2002). From https://github.com/mirrors/linux/blob/master/lib/rbtree.c.
9. Unknown author; *dictobject.c* (unknown date). From http://svn.python.org/projects/python/trunk/Objects/dictobject.c. In the same directory one will also find `dictnotes.txt`, an in-depth discussion on optimization of the Python dictionary structure.
10. Based on Stephen Adams, "Efficient sets: a balancing act", Journal of Functional Programming 3(4):553-562, October 1993, http://www.swiss.ai.mit.edu/~adams/BB. J. Nievergelt and E.M. Reingold, "Binary search trees of bounded balance", SIAM journal of computing 2(1), March 1973. http://www.haskell.org/ghc/docs/latest/html/libraries/containers/src/Data-Map.html#Map
11. Symmetric binary B-Trees: Data structure and maintenance algorithms, Rudolf Bayer
12. Robert Sedgewick; *Left-leaning Red-Black Trees* (2008). From http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf.
13. UNIX man page for grep
14. Multiple authors; *atoi(3) - Linux man page* (2012). From http://linux.die.net/man/3/atoi.
15. Multiple authors; *Agda* (2012). From http://wiki.portal.chalmers.se/agda/pmwiki.php.
16. Bart Jacobs; *VeriFast* (2012). From http://people.cs.kuleuven.be/~bart.jacobs/verifast/.
17. Wikipedia; *Hoare logic* (2012). From http://en.wikipedia.org/wiki/Hoare_logic.
18. John C. Reynolds; *Separation Logic: A Logic for Shared Mutable Data Structures* (2002). From http://www.cs.cmu.edu/~jcr/seplogic.pdf.
19. Bart Jacobs, Jan Smans, Frank Piessens; *The VeriFast Program Verifier: A Tutorial* (November 17, 2010). From http://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf.
20. `functions.c`, in `examples/functions` in the VeriFast distribution.

Appendix: miscellaneous C, fixpoints and lemmas

```
lemma_auto void in_left_in_append<V>(int needle, V v, List<Pair<int,V> > L0, List<Pair<int,V>
> R0)
    requires index_search(needle, L0) == Just(v);
    ensures index_search(needle, append(L0, R0)) == Just(v);
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch (l0) {
                case Pair(k0, v0): if (needle ≠ k0) in_left_in_append(needle, v, L1, R0);
            }
    }
}

lemma void in_right_list_hb_left_list<V>(int needle, V v, List<Pair<int,V> > L0, List<Pair<int,V>
> R0)
    requires sorted(append(L0, R0)) == true ∧ index_search(needle, R0) == Just(v);
    ensures higherBound(L0, needle) == true;
{
    switch (L0) {
        case Nil:
        case Cons(l0, L1):
            switch (l0) {
                case Pair(k0, v0):
                    in_right_list_hb_left_list(needle, v, L1, R0);
                    sides_of_sorted_are_sorted<V>(L0, R0);
                    switch (R0) {
                        case Nil: not possible
                        case Cons(r0, R1):
                            switch (r0) {
                                case Pair(rk0, rv0):
                                    left_side_of_sorted_is_bound(L0, rk0, rv0, R1);
                                    mem_gte_leftmost(rk0, rv0, R1, needle, v);
                                    loosen_higherBound(L0, rk0, needle);
                            }
                    }
            }
    }
}

predicate IsCircularList(ListNode * head; List<Pair<int,int> > L) =
    head == 0 ?
        L == Nil
    :
        IsListNode(head, ?key, ?value, ?tail) ∗
        IsListSeg(tail, head, ?$L_1$) ∗
        L == Cons(Pair(key, value), $L_1$)
    ;

predicate IsTreeNode(TreeNodePtr node; TreeNodePtr l, int k, int v, Color c, TreeNodePtr r)
    = node ≠ 0
    ∗ node->lft ↦ l
    ∗ node->rgt ↦ r
    ∗ node->key ↦ k
    ∗ node->value ↦ v
    ∗ node->color ↦ c
    ∗ malloc_block_TreeNode(node)
    ;

lemma void leftmost_is_lower_bound_of_left_list<V>(int $k_0$, V $v_0$, List<Pair<int,V> > $L_1$,
List<Pair<int,V> > $R_0$)
    requires sorted(append(Cons(Pair($k_0$, $v_0$), $L_1$), $R_0$)) == true;
    ensures lowerBound($k_0$, $L_1$) == true;
{
    switch ($L_1$) {
        case Nil:
        case Cons($l_1$, $L_2$):
            switch ($l_1$) {
                case Pair($k_1$, $v_1$): leftmost_is_lower_bound_of_left_list($k_0$, $v_0$, $L_2$, $R_0$);
            }
    }
}

lemma void leftmost_less_than_other_member<V>(int $k_0$, V $v_0$, List<Pair<int,V> > $L_0$, int k, V v,
List<Pair<int,V> > $R_0$)
    requires sorted(append(Cons(Pair($k_0$,$v_0$), $L_0$), Cons(Pair(k,v), $R_0$))) == true;
    ensures $k_0$ < k;
{
    switch ($L_0$) {
        case Nil:
        case Cons($l_0$, $L_1$):
            switch ($l_0$) {
                case Pair($k_1$, $v_1$): leftmost_less_than_other_member($k_1$, $v_1$, $L_1$, k, v, $R_0$);
            }
    }
}

bool list_search_iterative(int needle, int * value, ListNode * listNode)
requires IsList(listNode, ?I) ∗ integer(value, ?$v_0$);
ensures IsList(listNode, I) ∗ integer(value, ?$v_1$) ∗ IsMaybe(result, $v_1$, index_search(needle, I));
{
    ListNode * i = listNode;

    while (i ≠ 0)
    invariant IsListSeg(listNode, i, ?$I_0$) ∗ integer(value, $v_0$) ∗ IsList(i, ?$I_1$) ∗ I == append($I_0$, $I_1$)
    ∗ index_search(needle, $I_0$) == Nothing;
    {
        open IsListSeg(listNode, i, I_0);
        if (listNode->key == needle) {
            *value = listNode->value;
            close IsMaybe(true, listNode->value, index_search(needle, I));
            return true;
        }
        else {
            i = i->tail;
        }
    }

    return false;
}

void llrb_insert(int key, int value, TreeNodePtr * rootptr);
requires pointer(rootptr, ?root_0) ∗ IsTree(root_0, ?t_0) ∗ sorted(flatten(t_0)) == true ∗ IsLLRB(t_0,
Blk, _);
ensures pointer(rootptr, ?root_1) ∗ IsTree(root_1, ?t_1) ∗ sorted(flatten(t_1)) == true ∗ IsLLRB(t_1,
Blk, _) ∗ flatten(t_1) == index_insert(key, value, flatten(t_0));

fixpoint int longest_path<K,V>(Tree<K,V> t)
{
    switch (t) {
        case Leaf: return 0;
        case Branch(l, k, v, c, r):
            return max(longest_path(l), longest_path(r))+1;
    }
}

fixpoint int shortest_path<K,V>(Tree<K,V> t)
{
    switch (t) {
        case Leaf: return 0;
        case Branch(l, k, v, c, r):
            return min(shortest_path(l), shortest_path(r))+1;
    }
}

lemma void lowerBound_is_lowerBound_right<V>(int lb, List<Pair<int,V> > L_0,
List<Pair<int,V> > R_0)
    requires lowerBound(lb, append(L_0, R_0)) == true;
    ensures lowerBound(lb, R_0) == true;
{
    switch (L_0) {
        case Nil:
        case Cons(l_0, L_1):
            switch (l_0) {
                case Pair(k_0,v_0): lowerBound_is_lowerBound_right(lb, L_1, R_0);
            }
    }
}
```

```
fixpoint int max(int a, int b)
{
    return a < b ? b : a;
}

fixpoint Maybe<B> maybe_map<A,B>(fixpoint(A,B) f, Maybe<A> m) {
    switch (m) {
        case Nothing: return Nothing;
        case Just(a): return Just(f(a));
    }
}

lemma void mem_gte_leftmost<V>(int k0, V v0, List<Pair<int,V> > l1, int kn, V vn)
    requires sorted(Cons(Pair(k0,v0), l1)) == true ∗ index_search(kn, Cons(Pair(k0,v0), l1)) ==
    Just(vn);
    ensures k0 ≤ kn;
{
    switch (l1) {
        case Nil:
        case Cons(e1, l2):
            switch (e1) {
                case Pair(k1, v1):
                    if (k0 ≠ kn) {
                        mem_gte_leftmost(k1, v1, l2, kn, vn);
                    }
            }
    }
}

fixpoint int min(int a, int b)
{
    return a < b ? a : b;
}

predicate NearlyBalanced<K,V>(Tree<K,V> t) =
    longest_path(t) ≤ 2 * shortest_path(t);

TreeNodePtr newTreeNode(int key, int value);
requires emp;
ensures IsTree(result, ?t) ∗ IsLLRB(t, Red, 0) ∗ flatten(t) == Cons(Pair(key, value), Nil);

lemma void right_side_of_sorted_is_bound<V>(List<Pair<int,V> > L0, int k, V v,
List<Pair<int,V> > R0)
    requires sorted(append(L0, Cons(Pair(k,v), R0))) == true;
    ensures lowerBound(k, R0) == true;
{
    sides_of_sorted_are_sorted(L0, Cons(Pair(k,v), R0));
}

lemma void root_is_mem<V>(List<Pair<int,V> > L0, int k, V v, List<Pair<int,V> > R0)
    requires sorted(append(L0, Cons(Pair(k,v), R0))) == true;
    ensures index_search(k, append(L_0, Cons(Pair(k,v), R_0))) == Just(v);
{
    in_right_in_append(k, v, L_0, Cons(Pair(k,v), R_0));
}

lemma void RVRight_has_right_branch<K,V>(Tree<K,V> t_0)
    requires RVRight(t_0, ?h);
    ensures RVRight(t_0, h) * has_right_branch(t_0) == true;
{
    open RVRight(t_0, h);
    open Branch(t_0, _, _, _, _, _);

    switch (t_0) {
        case Leaf: not possible
        case Branch(l, k, v, c, r):
            red_has_branch(r);
            assert is_branch(r) == true;
            close Branch(t_0, _, _, _, _, _);
            close RVRight(t_0, h);
    }
}

lemma void tree_search_is_list_search<V>(int needle, Tree<int,V> t);
    requires bst(t) == true;
    ensures tree_search(needle, t) == index_search(needle, flatten(t));
```