Verifying concurrent indexes

Name James Fisher

Introduction

What is an index?

An index stores values, where each value has a distinct name. Many data sets fit this description. For example, a contact list stores *phone numbers*, where each number is accessed with (indexed by) a person's *name*. The Web can be modeled as a store of *pages*, where each page is indexed by a URL. More abstractly, an index defines a function from some *range* to some *domain*.

The index API

As a fundamental data type, almost every programming language has an index implementation. With it, the programmer can memoize the results of a function, model the houses on her street, or describe a directed graph. Here is pseudocode for a web cache:

```
Index<Url, Page> cache;  // `cache` stores webpages previously accessed.

Page get(Url u) {  // return the webpage at the URL `u`.
  Page p = cache.search(u);  // First look in the cache ...

  if (p != null && !p.expired())
    // if the cache has an entry for this URL and it has not expired,
    return p;  // just return that.

  p = http_get(u);  // Otherwise, fetch the page from the Web.

  if (p == null) cache.remove(u);  // If the page doesn't exist, remove our records of it,
  else           cache.insert(u, p);  // otherwise cache it for future requests.

  return p;
}
```

This small example illustrates the entire API. Notice that the `Index` is an 'abstract', or 'container' data type: we must supply the types of the keys and the values (in the above, the keys are `Url`s and the values are `Page`s). The resulting 'concrete' data type can then be instantiated. The instantiated index `cache` has three operations that can be performed on it: `search`, `insert` and `remove`. For an index `i` that maps keys of type `K` onto values of type `V`, these operations do the following:

`value = i.search(key)` If there is a value associated with `key` in `i`, `value` will be that value. Otherwise, `value` will be `null`.

`i.insert(key, value)` Subsequent calls to `i.search(key)` will return `value`, until a subsequent call to `i.insert(k, —)` or `i.remove(k)`, where `k == key`.

`i.remove(key)` Subsequent calls to `i.search(key)` will return `null`, until subsequent calls to `i.insert(k, —)` or `i.remove(k)`, where `k == key`.

Verifying linked lists

The LL data structure

The linked list (LL) is the simplest set data structure that will submit to fairly efficient insertion, deletion and search of elements from a general ordered universe. It is a good place to start to set out the principles of verification that we can then aplly to more complex algorithms.

A LL representing set **S** consists of a set of |**S**| records, called *nodes*, in memory. There is a one-to-one correspondence between elements of **S** and the set of `Nodes` in the LL representing **S**. Each node contains two fields, `value` and `tail`. This can be written in a few lines:

```
module ll.node;

struct Node {
  int value;   // The lowest value in the set
  Node* tail;  // list containing all values greater than `value`

  this(int value) {
    this.value = value;  // The tail pointer is initialized to null
  }

  this(int value, Node* tail) {
    this.value = value;
    this.tail = tail;
  }
}
```

The fields `value` and `tail` have fixed length, respectively v and t. An instance of the `Node` record thus has a fixed length in memory $l = v + t$. Therefore any node beginning at some address a spans the contiguous addresses $a \ldots a + l$. The addresses occupied by the `Nodes` are *disjoint*; *i.e.*, for any `Node`, its occupied addresses are occupied no other `Node`. We say that a record beginning at address a is 'at a'.

As there is a one-to-one correspondence between elements of **S** and `Nodes`, each `Node` contains as its `value` field one element of **S**, and for each element of **S**, there exists a `Node` containing it. The `tail` field of a record holds the address of another record in the data structure. Specifically, for a record `r` representing element s, if there are elements in **S** larger than s, the `tail` field of `r` is set to the address of the `Node` representing the next-largest element; otherwise, the `tail` field is set to `null`.

The final necessary piece of information is the address of the record representing the smallest element in **S**, which we call `head`. This address and the set of memory locations occupied by the `Nodes` comprise the LL data structure.

We can visualize the address space of memory as a line of cells, from address 0 on the left-hand-side to address_max on the right. Here is an example LL representing the set **S** = { 2, 29, 30, 77 } from the domain of natural numbers < 256. There are 216 memory locations, spanning addresses 0 to 65,535, so a memory address is two bytes long. Memory locations are one byte long. The domain of the set is representable by one byte, so the `value` field is one byte long; and the `tail` field, being an address, is two bytes long; an single record is thus three bytes in total. In the following diagram, arrows are used to highlight fields that reference other memory locations.

image/svg+xml 2 2,034 36,945 36,946 36,947 29 34,562 2,034 2,035 2,036 30 54,396 34,562 34,563 34,564 77 NULL 54,396 54,397 54,398 [0…2,033] 0

[2,037…34,561] [34,565…36,944] [36,948…54,395] [54,399…65,535] 36,945

The `Nodes` are at arbitrary memory locations; the program does not have the ability to decide on the locations given to it. The addresses cannot really contribute to the data held by the data structure, and so the above diagram therefore contains a lot of visual 'noise'. We can abstract the diagram to remove these addresses and untangle the arrows:

image/svg+xml 2 29 30 77 NULL head

It is now visually obvious why this is called a list: the structure can be seen as a connected, directed, acyclic graph, with a single path between the node distinguished as the first and that distinguished as the last, on which all nodes lie. The second thing this diagram makes visually obvious is that the linked list 'contains' smaller linked lists within it. For example, the `tail` field of the `Node` representing the element 2 is effectively the `head` address for a linked list containing the right-most three nodes. We can outline all the linked lists in the above diagram:

image/svg+xml 2 29 30 77 NULL head ∅ { **77** } { **30, 77** } { **29, 30, 77** } { **2, 29, 30, 77** }

It is now obvious that the LL is a *recursive data structure*: it contains smaller versions of the same structure. Abstracting from our example, we can show this recursion visually:

image/svg+xml *value head* **T** {***value***} ∪ **T**

Notice in our example that though $|\mathbf{S}| = 4$, there are actually *five* linked lists: the final `null` pointer conceptually points to a linked list representing ∅, which uses no memory locations. This does not fit in the scheme of the above diagram, which really only applies to non-empty sets. The linked list, then, actually uses two distinct representations of sets: one for non-empty sets as above, and another for the empty set, signaled by the use of a `null` pointer.

Let us refer to these representations as NonEmptyList and EmptyList respectively, which are both subtypes of the abstract type List. These are all *predicates*; meaning they wrap up a description of (part of) the state of the program, including the structure of memory. Each of the above predicates takes two parameters corresponding to the two necessary 'parts' of a LL: the `head` pointer into the structure, and the set that the structure represents. That is, $\mathsf{List}(\texttt{head}, \mathbf{S})$ describes a set of memory locations forming a linked list that represents set $\mathbf{S}$, with `head` being the address of the first `Node` in the list (or `null` if $\mathbf{S} = \varnothing$).

image/svg+xml *v head tail* List(*tail*, **T**) NonEmptyList(*head*, **S**),where **S** = {*v*} ∪ **T**,and ∀ *t*∈**T**. *v* < *t*. *head* EmptyList(*head*, **S**),where **S**

= ∅. List(*head*, **S**)

We now have a strong visual intuition for how to describe the LL formally. The above diagram translates cleanly into the notation of separation logic. Let's approach this top-down, and begin with the List predicate. The LL defines two representations of sets, and the general List predicate simply means that one of the two representations is being used to represent the set. We can therefore define List just as a disjunction:

$$\mathsf{List}(\texttt{head}, \mathbf{S}) \stackrel{\text{def}}{=} \mathsf{EmptyList}(\texttt{head}, \mathbf{S}) \lor \mathsf{NonEmptyList}(\texttt{head}, \mathbf{S}).$$

The empty set is represented by a `null` pointer and no allocated heap memory. Translating this to separation logic is also straightforward:

$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \stackrel{\text{def}}{=}$	$\mathtt{head} = \mathtt{null}\ \land$	
	$\mathbf{S} = \varnothing\ \land$	$\mathbf{S}$ is the empty set and
	$\mathbf{emp}$.	there is no allocated memory.

The $\mathsf{NonEmptyList}$ is only a little more complex. First, we must express the relationship between the set $\mathbf{S}$ represented by the list pointed to by `head`, the *value* in the `Node`, and the set $\mathbf{T}$ represented by the *tail* pointer. First, $\mathbf{S}$ is the element v plus the elements in $\mathbf{T}$; we write $\mathbf{S} = \{value\} \cup \mathbf{T}$. But *value* is not just an arbitrarily chosen value from $\mathbf{S}$, it is the smallest element. Another way to say this is that all the elements in $\mathbf{T}$ are all greater than v; we write $\forall t \in \mathbf{T}\ .\ value < t$. We can express this relationship with a helper predicate, $\mathsf{Compose}$, to be used by $\mathsf{NonEmptyList}$.

$$\mathsf{Compose}(value, \mathbf{T}, \mathbf{S}) \stackrel{\text{def}}{=} \{value\} \cup \mathbf{T} = \mathbf{S}\ \land$$
$$\forall t \in \mathbf{T}\ .\ value < t\ .$$

The predicate $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$ must first assert that $\mathbf{S}$ is non-empty. One way of saying this is to assert that the exists some element—let's call it *value*—in the set. We can then also assert that there exists some other set, $\mathbf{T}$, which together with $\{value\}$ forms $\mathbf{S}$. While we're at it, we can choose *value* to be the minimal element, as we will shortly be concerned with this value when describing the first `Node` in the list. Now we can use our $\mathsf{Compose}$ predicate to describe *value* and $\mathbf{T}$ in relation to $\mathbf{S}$; we write $\exists value, \mathbf{T}.\ \mathsf{Compose}(value, \mathbf{T}, \mathbf{S})$.

We must now describe the memory layout for a non-empty list. Looking at our visual definition, notice that the memory locations can be divided into two disjoint sets: one set for the first `Node`, consisting of contiguous memory locations beginning at `head`, and another set of locations for the rest of the `Node`s, which can be described by the List predicate. We can decompose the memory description into a description of the first `Node` and a description of the rest of the list, specifying that these sets are disjoint. This is exactly what the separating conjunction, $\ast$, does.

The first `Node` record begins at address `head` and consists of two fields, the first containing *value*, and the second containing some (unknown) address *tail*. We write this as $\mathtt{head} \mapsto value, tail$. Notice that the lengths of the fields is left unspecified and is in fact unimportant.

The rest of the list represents $\mathbf{T}$ and its head is at address *tail*. We already have a predicate to describe this: $\mathsf{List}(tail, \mathbf{S})$. Notice that List and $\mathsf{NonEmptyList}$ are mutually recursive. We are now in a position to define $\mathsf{NonEmptyList}$:

$$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \stackrel{\text{def}}{=} \exists value, tail, \mathbf{T}.$$
$$\mathsf{Compose}(value, \mathbf{T}, \mathbf{S})\ \land$$
$$\mathtt{head} \mapsto value, tail\ \ast\ \mathsf{List}(tail, \mathbf{T}).$$

Lemmata used in LL algorithms

$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\mathtt{head}, \mathbf{S})$

Simple proof used to obtain the postcondition of functions that require List rather than $\mathsf{NonEmptyList}$.

1.	$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$	given
2.	$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \lor \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$	given, $\lor$I
3.	$\mathsf{List}(\mathtt{head}, \mathbf{S})$	orNonEmpty, close predicate

$\underline{\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\mathtt{head}, \mathbf{S})}$

(Serves the same use as for $\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\mathtt{head}, \mathbf{S})$).

1.	$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$	given
2.	$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \lor \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$	given, $\lor$I
3.	$\mathsf{List}(\mathtt{head}, \mathbf{S})$	orEmpty, close predicate

$\underline{\mathsf{EmptyList}(\_, \mathbf{S}) \Rightarrow \mathtt{value} \notin \mathbf{S}}$

If we have an empty list, then for any given value, it is not in the set.

1.	$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$	given
2.	$\mathtt{head} = \mathtt{null} \land \mathbf{S} = \emptyset \land \mathbf{emp}$	given, open predicate
3.	$\mathbf{S} = \emptyset$	openEmptyList, $\land$E
4.	$\mathtt{value} \notin \emptyset$	Nothing in empty set
5.	$\mathtt{value} \notin \mathbf{S}$	SIsEmpty, valueNotInEmptySet, equality

$\underline{\mathsf{List}(\mathtt{head}, \mathbf{S}) \land \mathtt{head} = \mathtt{null} \Rightarrow \mathsf{EmptyList}(\mathtt{head}, \mathbf{S})}$

At the start of all LL algorithms, we test whether the `head` pointer is `null`. If it is, we can deduce that the list is empty.

1.	$\mathsf{List}(\mathtt{head}, \mathbf{S}) \land \mathtt{head} = \mathtt{null}$	given
2.	$\mathsf{List}(\mathtt{head}, \mathbf{S})$	given, $\land$E
3.	$\mathtt{head} = \mathtt{null}$	given, $\land$E
4.	$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \lor \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$	list, open predicate
5.	$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$	assume
6.	$\exists v, t, \mathbf{T}.\ \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land \mathtt{head} \mapsto v, t * \mathsf{List}(t, \mathbf{T})$	assumeNonEmpty, open predicate
7.	$\exists v, t.\ \mathtt{head} \mapsto v, t$	openNonEmpty, $\land$E, frame off $\mathsf{List}(t, \mathbf{T})$
8.	$\mathtt{head} \neq \mathtt{null}$	headPointsTo, pointer non-null
9.	$\neg\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$	assumeNonEmpty, headNull, headNotNull, RAA
10.	$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$	openList, notNotEmpty, $\lor$E

$\underline{\mathsf{List}(\mathtt{head}, \mathbf{S}) \land \mathtt{head} \neq \mathtt{null} \Rightarrow \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})}$

(Symmetrical to the converse lemma.)

$\underline{\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \Rightarrow v \in \mathbf{S}}$

Used to show that if we found the desired element at the head of the list then it is in the list.

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	given

2.	$\{v\} \cup \mathbf{T} = \mathbf{S} \land \forall\, t \in \mathbf{T}\,.\ value < t$	given, open predicate
3.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	openCompose, ∧E
4.	$\{v\} \subseteq \mathbf{S}$	??
5.	$v \in \{v\}$	Element in singleton set
6.	$v \in \mathbf{S}$	valueSubsetS, valueInValue, member of subset is member of set

<u>$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land v \neq \mathtt{value} \Rightarrow \mathtt{value} \in \mathbf{T} \leftrightarrow \mathtt{value} \in \mathbf{S}$</u>

If the value we are searching for is not at the head of the list, then it is in the set if and only if it is in the tail.

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land v \neq \mathtt{value}$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	given, ∧E
3.	$v \neq \mathtt{value}$	given, ∧E
4.	$\{\mathtt{value}\} \cup \mathbf{T} = \mathbf{S} \land \forall\, t \in \mathbf{T}\,.\ \mathtt{value} < t$	Compose, open predicate
5.	$\{\mathtt{value}\} \cup \mathbf{T} = \mathbf{S}$	openCompose, ∧E
6.	$\forall\, t \in \mathbf{T}\,.\ \mathtt{value} < t$	openCompose, ∧E
7.	$\mathbf{T} \subseteq \mathbf{S}$	valueUTIsS
8.	$\quad \mathtt{value} \in \mathbf{T}$	assume
9.	$\quad \mathtt{value} \in \mathbf{S}$	TsubsetS, assumeValueInT, member of subset is member of set
10.	$\mathtt{value} \in \mathbf{T} \to \mathtt{value} \in \mathbf{S}$	assumeValueInT, themValueInS, →I
11.	$\mathtt{value} \notin \{v\}$	noteq
12.	$\quad \mathtt{value} \in \mathbf{S}$	assume
13.	$\quad \mathtt{value} \in \{v\} \lor \mathtt{value} \in \mathbf{T}$	assumeValueInS, valueUTIsS, ??
14.	$\quad \mathtt{value} \in \mathbf{T}$	inSetOfValueOrT, valueNotInv,
15.	$\mathtt{value} \in \mathbf{S} \to \mathtt{value} \in \mathbf{T}$	assumeValueInS, valueInT, →I
16.	$\mathtt{value} \in \mathbf{T} \leftrightarrow \mathtt{value} \in \mathbf{S}$	impliesForwards, impliesBackwards, ↔I

<u>$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land \mathtt{value} < v \Rightarrow \mathtt{value} \notin \mathbf{S}$</u>

If the head of the list is greater than the value we're looking for, then the value is not in the tail either.

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land \mathtt{value} < v$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land \mathtt{value} \neq v$	given, $a < b \Rightarrow a \neq b$
3.	$\mathtt{value} \in \mathbf{T} \leftrightarrow \mathtt{value} \in \mathbf{S}$	noteq, application of previous lemma
4.	$\mathtt{value} \in \mathbf{T} \to \mathtt{value} \in \mathbf{S}$	applyLemma, ↔E
5.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	given, ∧E
6.	$\mathtt{value} < v$	given, ∧E
7.	$\{v\} \cup \mathbf{T} = \mathbf{S} \land \forall\, t \in \mathbf{T}\,.\ v < t$	Compose, open predicate
8.	$\forall\, t \in \mathbf{T}\,.\ v < t$	openCompose, ∧E

9.	$\quad \texttt{value} \in \mathbf{S}$	assume
10.	$\quad v < \texttt{value}$	allInTGreaterThanV
11.	$\texttt{value} \notin \mathbf{S}$	assumeValueInS, lt, thenVLessThanValue, RAA

$\underline{\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge v < \texttt{value} \Rightarrow \mathsf{Compose}(v, \mathbf{T} \cup \{\texttt{value}\}, \mathbf{S} \cup \{\texttt{value}\})}$

Inserting a value greater than the head of the list into the tail gives us a new list with valid order.

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge v < \texttt{value}$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	given, ∧E
3.	$v < \texttt{value}$	given, ∧E
4.	$\{v\} \cup \mathbf{T} = \mathbf{S} \wedge \forall t \in \mathbf{T} .\ v < t$	givenCompose, open predicate
5.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	openCompose, ∧E
6.	$\forall t \in \mathbf{T} .\ v < t$	openCompose, ∧E
7.	$\{v\} \cup \mathbf{T} \cup \{\texttt{value}\} = \mathbf{S} \cup \{\texttt{value}\}$	vUTIsS, $a = b \Rightarrow f(a) = f(b)$
8.	$\forall t \in \{\texttt{value}\} .\ v < t$	givenlt, ??
9.	$\forall t \in \mathbf{T} \cup \{\texttt{value}\} .\ v < t$	allInTGreaterThanv, allInvalueGreaterThanv, ??
10.	$\mathsf{Compose}(v, \mathbf{T} \cup \{\texttt{value}\}, \mathbf{S} \cup \{\texttt{value}\})$	vUTUvalueIsSUvalue, allInTUvalueGreaterThanv, close predicate

$\underline{\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge w < v \Rightarrow \mathsf{Compose}(w, \mathbf{S}, \mathbf{S} \cup \{w\})}$

Prepending a smaller value than that at the head of the list yields the desired new list in valid order.

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge w < v$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	given, ∧E
3.	$w < v$	given, ∧E
4.	$\{v\} \cup \mathbf{T} = \mathbf{S} \wedge \forall t \in \mathbf{T} .\ v < t$	givenCompose, open predicate
5.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	openCompose, ∧E
6.	$\forall t \in \mathbf{T} .\ v < t$	openCompose, ∧E
7.	$\forall t \in \mathbf{T} .\ w < t$	allInTGreaterThanv, givenlt, transitivity of < relation
8.	$\{w\} \cup \mathbf{S} = \mathbf{S} \cup \{w\}$	Commutativity of set union
9.	$\forall t \in \{v\} .\ w < t$	givenlt, ??
10.	$\forall t \in (\{v\} \cup \mathbf{T}) .\ w < t$	allInTGreaterThanvalue, allInvGreaterThanvalue, ??
11.	$\forall t \in \mathbf{S} .\ w < t$	allInvUTGreaterThanvalue, vUTIsS, substitution
12.	$\mathsf{Compose}(w, \mathbf{S}, \mathbf{S} \cup \{w\})$	valueUSIsSUvalue, allInSGreaterThanvalue, close predicate

$\underline{\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \Rightarrow \mathbf{T} = \mathbf{S} \setminus \{v\}}$

This lemma is useful in the `remove` algorithm in order to demonstrate that we can return the tail of the list if we found the element to remove at the head.

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	given
2.	$\{v\} \cup \mathbf{T} = \mathbf{S} \wedge \forall\, t \in \mathbf{T}\,.\; v < t$	given, open predicate
3.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	openCompose, ∧E
4.	$\forall\, t \in \mathbf{T}\,.\; v < t$	openCompose, ∧E
5.	$\quad v \in \mathbf{T}$	assume
6.	$\quad v < v$	allInTGreaterThanv, assumevInT
7.	$v \notin \mathbf{T}$	assumevInT, vltv, RAA
8.	$\{v\} \cap \mathbf{T} = \varnothing$	vNotInT
9.	$\mathbf{T} = \mathbf{S} \setminus \{v\}$	vUTIsS, vAndTDisjoint

$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge v \neq w \Rightarrow \mathsf{Compose}(v, \mathbf{T} \setminus \{w\}, \mathbf{S} \setminus \{w\})$

Used to show that recursive application of `remove` on the tail of the list yields a new list with the element removed.

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge v \neq w$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	given, ∧E
3.	$v \neq w$	given, ∧E
4.	$\{v\} \cup \mathbf{T} = \mathbf{S} \wedge \forall\, t \in \mathbf{T}\,.\; v < t$	givenCompose, open predicate
5.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	openCompose, ∧E
6.	$\forall\, t \in \mathbf{T}\,.\; v < t$	openCompose, ∧E
7.	$(\{v\} \cup \mathbf{T}) \setminus \{w\} = \mathbf{S} \setminus \{w\}$	vUTIsS, equal operations
8.	$(\{v\} \setminus \{w\}) \cup (\mathbf{T} \setminus \{w\}) = \mathbf{S} \setminus \{w\}$	(vUT)minuswIsSminusw, distributivity of set minus over set union
9.	$\{v\} \setminus \{w\} = \{v\}$	vNotw, ??
10.	$\{v\} \cup (\mathbf{T} \setminus \{w\}) = (\mathbf{S} \setminus \{w\})$	vminuswUTminuswIsSminusw, vminuswIsv, substitution
11.	$\forall\, t \in (\mathbf{T} \setminus \{w\})\,.\; v < t$	allInTGreaterThanv, for all in set then for all in subset
12.	$\{v\} \cup (\mathbf{T} \setminus \{w\}) = (\mathbf{S} \setminus \{w\}) \wedge \forall\, t \in (\mathbf{T} \setminus \{w\})\,.\; v < t$	vUTminuswIsSminusw, AllInTminuswGreaterThanv, ∧I
13.	$\mathsf{Compose}(v, \mathbf{T} \setminus \{w\}, \mathbf{S} \setminus \{w\})$	composeParts, close predicate

Recursive LL algorithms

A recursive LL `search` algorithm

The purpose of a `search` algorithm is to determine whether a given element from the domain is a member of the set represented by a particular LL. Our first task is to formally encode this purpose in a *specification*. Our basic tool here is the concept of pre- and post-conditions: given that A (some description of the program state) is true before the `search`, B will be true afterwards. Our task then is to establish first what is necessary before execution of `search` in order for that execution to be meaningful, and secondly what (given that this pre-condition is satisfied) the `search` function guarantees will be true afterwards.

The `search` function takes two parameters: `head`, a pointer into a LL, and `value`, the element we are searching for. The value of the `value` parameter is arbitrary, and so we need not concern ourselves with it in the precondition: all values are valid. The `head` variable, on the other hand, is not arbitrary: it must point to a valid LL. We can express that: List(`head`, **S**).

The execution of `search` will return a boolean value, so the function will be called like so: `o = search(head, value)`. The value `o` will express whether *value* $\in$ **S**; *i.e.*, `o` $\leftrightarrow$ *value* $\in$ **S**. There is an additional post-condition: `search` leaves the LL intact, and so List(`head`, **S**) continues to be true after execution. (Notice this specification in fact allows the algorithm to alter memory as long as it leaves it in a state that represents the abstract set **S**. For our purposes this makes the specification simpler.)

Our specification for `search` is:

{ List(`tail`, **S**) } { `o = search(tail, value)` } { List(`tail`, **S**) $\land$ `o` $\leftrightarrow$ `value` $\in$ **S** }

The recursive method to search a list closely follows the recursive data structure that we have defined. Given a List(`head`, **S**), either it is empty or it is not. If it is empty (as indicated by `head = null`), then it does not contain `value`, so we can return `false`. Otherwise, we look at the first value in the list, v, and compare it to `value`. Three relations are possible: `value` $< v$, or `value` $= v$, or $v <$ `value`. Most trivially, if `value` $= v$ then `value` is in **S**, so we can return `true`. Next, if `value` $< v$, then `value` is not in **S**, because the list is in ascending order. Finally, if $v <$ `value`, then `value` could be in **S** if and only if it is in **T**, so we call `search(tail, value)`, and return that value. Diagrammatically, our recursive search algorithm works as follows:

image/svg+xml NonEmptyList(*head*, S) The list is non-empty. The values *v* and *tail* areunknown. **S** = {*v*} ∪ **T** and ∀ *t*∈**T**. *v* < *t*. *v* *head*

tail List(*tail*, **T**) **Precondition:** *head* points to a list representingset S. We are searching for *value*. *head* List(*head*, **S**) The list is empty, and

represents the set ∅.*value* ∉ ∅. Return false. *head* EmptyList(*head*, **S**) **Is the pointer *head* null?** *head* is not null *head* is null

compare(*value*, *v*) *value* < *v* *value* = *v* *value* > *v* NonEmptyList(*head*, S) *value* ∉ {*v*}. All in **T** are > *v*, so *value* ∉ **T**. So*value* ∉ {*v*} ∪ **T**, so

value ∉ **S**. Return false. *v*> *value* *head* *tail* List(*tail*, **T**) NonEmptyList(*head*, S) *value* ∈ {*v*}, so *value* ∈ {*v*} ∪ **T**, so *value* ∈ **S**.Return true.

value *head* *tail* List(*tail*, **T**) NonEmptyList(*head*, S) *value* ∉ {*v*}, so (*value* ∈ S) ↔ (*value* ∈ **T**).Recursively apply search to *tail*. *v*< *value*

head *tail* List(*tail*, **T**)

A full annotation of our recursive search algorithm follows.

```
module ll.search.recursive;

import ll.node;

bool search(Node* head, int value) {
```

List(head, **S**)

```
  bool o;
  if (head == null) {
```

Assert if-condition.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{head} = \mathtt{null}$

Lemma: $\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{head} = \mathtt{null} \Rightarrow \mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$.

$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$

Lemma: $\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathtt{value} \notin \mathbf{S}$.

$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{value} \notin \mathbf{S}$

Weakening lemma: $\mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\mathtt{head}, \mathbf{S})$.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{value} \notin \mathbf{S}$

```
    o = false;
```

Assignment.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{value} \notin \mathbf{S} \wedge \mathtt{o} = \mathtt{false}$

false ↔ false.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{o} \leftrightarrow \mathtt{value} \in \mathbf{S}$

```
}
else {
```

Deny if-condition.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{head} \neq \mathtt{null}$

Lemma: $\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{head} \neq \mathtt{null} \Rightarrow \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$.

$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$

Open $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \mathtt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T})$

```
    if (head.value == value) {
```

Assert if-condition: substitute value for v.

$\exists tail, \mathbf{T}.$
$\quad \mathsf{Compose}(\mathtt{value}, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \mathtt{head} \mapsto \mathtt{value}, tail \ast \mathsf{List}(tail, \mathbf{T})$

$\mathsf{Compose}(\mathtt{value}, \mathbf{T}, \mathbf{S}) \Rightarrow \mathtt{value} \in \mathbf{S}$.

$\exists tail, \mathbf{T}.$
$\quad \mathsf{Compose}(\mathtt{value}, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \mathtt{head} \mapsto \mathtt{value}, tail \ast \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad \mathtt{value} \in \mathbf{S}$

Close $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$.

$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{value} \in \mathbf{S}$

Weakening lemma: $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\mathtt{head}, \mathbf{S})$.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \wedge \mathtt{value} \in \mathbf{S}$

```
        o = true;
```

Assignment.

$\text{List}(\texttt{head}, \mathbf{S}) \wedge \texttt{value} \in \mathbf{S} \wedge \texttt{o} = \texttt{true}$

true ↔ true.

$\text{List}(\texttt{head}, \mathbf{S}) \wedge \texttt{o} \leftrightarrow \texttt{value} \in \mathbf{S}$

```
}
else {
```

Deny if-condition.

$\exists v, tail, \mathbf{T}.$
$\quad \text{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \text{List}(tail, \mathbf{T}) \wedge$
$\quad v \neq \texttt{value}$

```
  if (head.value < value) {
```

Assert if-condition.

$\exists v, tail, \mathbf{T}.$
$\quad \text{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \text{List}(tail, \mathbf{T}) \wedge$
$\quad v < \texttt{value}$

Lemma: $\text{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge v < \texttt{value} \Rightarrow \texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S}$.

$\exists v, tail, \mathbf{T}.$
$\quad \text{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \text{List}(tail, \mathbf{T}) \wedge$
$\quad v < \texttt{value} \wedge$
$\quad \texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S}$

```
    o = search(head.tail, value);
```

Inductive use of specification. Note $|\mathbf{T}| < |\mathbf{S}|$.

$\exists v, tail, \mathbf{T}.$
$\quad \text{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \text{List}(tail, \mathbf{T}) \wedge$
$\quad v < \texttt{value} \wedge$
$\quad \texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S} \wedge$
$\quad \texttt{o} \leftrightarrow \texttt{value} \in \mathbf{T}$

Transitivity of double implication. Discard unrequired assertions.

$\exists v, tail, \mathbf{T}.$
$\quad \text{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \text{List}(tail, \mathbf{T}) \wedge$
$\quad \texttt{o} \leftrightarrow \texttt{value} \in \mathbf{S}$

Close $\text{NonEmptyList}(\texttt{head}, \mathbf{S})$.

$\text{NonEmptyList}(\texttt{head}, \mathbf{S}) \wedge \texttt{o} \leftrightarrow \texttt{value} \in \mathbf{S}$

Weakening: $\text{NonEmptyList}(\texttt{head}, \mathbf{S}) \Rightarrow \text{List}(\texttt{head}, \mathbf{S})$.

$\text{List}(\texttt{head}, \mathbf{S}) \wedge \texttt{o} \leftrightarrow \texttt{value} \in \mathbf{S}$

```
  }
  else {
```

Deny if-condition. Use $\neg(v < \texttt{value}) \Rightarrow \texttt{value} \leq v$.

$\exists v, tail, \mathbf{T}.$

```
            Compose(v, T, S) ∧
            head ↦ v, tail ∗ List(tail, T) ∧
            value ≠ v ∧ value ≤ v

        (a ≠ b ∧ a ≤ b) ⇒ b < a.

        ∃v, tail, T.
            Compose(v, T, S) ∧
            head ↦ v, tail ∗ List(tail, T) ∧
            value < v

        Lemma: (Compose(v, T, S) ∧ value < v) ⇒ value ∉ S.

        ∃v, tail, T.
            Compose(v, T, S) ∧
            head ↦ v, tail ∗ List(tail, T) ∧
            value ∉ S

        Close List(head, S).

        List(head, S) ∧ value ∉ S
       o = false;

        Assignment.

        List(head, S) ∧ value ∉ S ∧ o = false

        false ↔ false.

        List(head, S) ∧ o ↔ value ∈ S
      }

      If-rule.

      List(head, S) ∧ o ↔ value ∈ S
    }

    If-rule.

    List(head, S) ∧ o ↔ value ∈ S
  }

  If-rule.

  List(head, S) ∧ o ↔ value ∈ S
  return o;
}
```

A recursive LL `insert` algorithm

The purpose of an `insert` algorithm is to mutate the heap representing some set **S** such that it represents the set **S** ∪ {`value`} for some given parameter `value`. A function implementing `insert` returns a pointer into the new heap. A specification for our LL data structure falls naturally out of this description:

{ List(head, **S**) } { `nhead = insert(head, value);` } { NonEmptyList(nhead, **S** ∪ {value}) }

The recursive `insert` algorithm works as follows. We first determine whether the list is empty by testing for `value` = `null`. If the list is empty, it represents ∅, and we wish to mutate the state to represent ∅ ∪ {`value`}, which is simply {`value`}. The representation of this is a one-node list with its *tail* field set to `null`; we create this and return it. If the list is not empty, we compare *v*, the first value in the list, with `value`. If *v* = `value`, then `value` ∈ **S**, so $\emptyset \cup$ {`value`} = **S**. The `head` variable already points into a set representing **S**, so we just return `head`. If $v >$ `value`, then `value` $\notin$ **S**, and we can construct a valid LL representing **S** $\cup$ {`value`} by appending a node containing `value` at the start. Finally, if $v <$ `value`, we call `insert(tail, value)` to obtain a pointer to a list representing **T** $\cup$ {`value`}; by replacing the *tail* field of the first node with this new pointer, we obtain a list representing $\{v\}$ $\cup$ **T** $\cup$ {`value`}, which is equal to **S** $\cup$ {`value`}. We then return the original `head` pointer.

image/svg+xml NonEmptyList(*head*, S) The list is non-empty. The values *v* and *tail* areunknown. **S** = {*v*} ∪ **T** and ∀ *t*∈**T**. *v* < *t*. *v* *head*

tail List(*tail*, **T**) **Precondition:** *head* points to a list representingset **S**. We are inserting *value*. *head* List(*head*, **S**) The list is empty, and

represents the set ∅.{*value*} ∪ ∅ = {value}. Return one-node list. *head* EmptyList(*head*, **S**) **Is the pointer *head* null?** *head* is not null *head* is

null NonEmptyList(*head*, **S**) ∀ *s*∈**S**. *value* < *s*, so we can insert *value* at thehead of the list. *v*> *value* *head* *tail* List(*tail*, **T**)

NonEmptyList(*head*, {*value*} ∪ **S**) *value* ∈ **S**, so {*value*} ∪ **S** = **S**. Return *head*,the same linked list. *value* *head* *tail* List(*tail*, **T**)

NonEmptyList(*head*, S) The head of the list is consistent with *value* ∈ **S**.We must ensure *value* is in the tail. *v*< *value* *head* *tail* List(*tail*, **T**)

{*value*} ∪ ∅ = {value}. We are done.Return *nhead*. NonEmptyList(*nhead*, {*value*}) *value* *nhead* null EmptyList(null, ∅) **Construct one-**

node list containing *value*,pointed at by new variable *nhead*. compare(*value*, *v*) *value* < *v* *value* = *v* *value* > *v* NonEmptyList(*nhead*,

{*value*} ∪ **S**) NonEmptyList(*head*, **S**) NonEmptyList(*nhead*, {*value*} ∪ **S**), so *nhead* satisfies the postcondition.Return *nhead*. *v*> *value* *tail*

List(*tail*, **T**) *value* *nhead* *head* **Construct a new node containing *value* and*head*, pointed at by new variable *nhead*.** NonEmptyList(*head*,

{*value*} ∪ **S**) *head* now points to a list representing {*v*} ∪ {*value*} ∪ **T**, which bycommutativity and substitution is {*value*} ∪ **S**. Return *head*.

v< *value* *head* *ntail* List(*tail*, **T**) insert , *value* NonEmptyList(*ntail*, {*value*} ∪ **T**) **Recursively apply insert to *tail*, yielding newpointer**

***ntail*. Set *tail* field of first node to *ntail*.**

The annotated code for recursive `insert` follows.

```
module ll.insert.recursive;

import ll.node;

Node* insert(Node* head, int value) {
  Node* o;
```

Precondition.

$\mathsf{List}(\mathtt{head}, \mathbf{S})$

```
if (head == null) {
```

Assert if-condition.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \land \mathtt{head} = \mathtt{null}$

Lemma: $\mathsf{List}(\mathtt{head}, \mathbf{S}) \land \mathtt{head} = \mathtt{null} \Rightarrow \mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$

$\mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$

Open $\mathsf{EmptyList}(\mathtt{head}, \mathbf{S})$. Discard $\mathtt{head} = \mathtt{null}$.

$\mathbf{S} = \varnothing \land \mathbf{emp}$

```
  o = new Node(value);
```

Specification for `new Node(value)`.

$\mathbf{S} = \varnothing \land \mathbf{emp} \ast \mathsf{NonEmptyList}(\mathtt{o}, \{\mathtt{value}\})$

$\mathbf{emp} \ast \mathsf{X} = \mathsf{X}$.

$\mathbf{S} = \varnothing \land \mathsf{NonEmptyList}(\mathtt{o}, \{\mathtt{value}\})$

$\mathbf{X} = \varnothing \cup \mathbf{X}$.

$\mathbf{S} = \varnothing \land \mathsf{NonEmptyList}(\mathtt{o}, \varnothing \cup \{\mathtt{value}\})$

Substitution. Discard $\mathbf{S} = \varnothing$.

$\mathsf{NonEmptyList}(\mathtt{o}, \mathbf{S} \cup \{\mathtt{value}\})$

```
}
else {
```

Deny if-condition.

$\mathsf{List}(\mathtt{head}, \mathbf{S}) \land \mathtt{head} \neq \mathtt{null}$

Lemma: $\mathsf{List}(\mathtt{head}, \mathbf{S}) \land \mathtt{head} \neq \mathtt{null} \Rightarrow \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$.

$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$

Open $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
$\quad \mathtt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T})$

```
  if (head.value == value) {
```

Assert if-condition: substitute `value` for v.

$\exists tail, \mathbf{T}.$
$\quad \mathsf{Compose}(\mathtt{value}, \mathbf{T}, \mathbf{S}) \land$
$\quad \mathtt{head} \mapsto \mathtt{value}, tail \ast \mathsf{List}(tail, \mathbf{T})$

$\mathsf{Compose}(\mathtt{value}, \mathbf{T}, \mathbf{S}) \Rightarrow \mathtt{value} \in \mathbf{S}$.

$\exists tail, \mathbf{T}.$
$\quad \mathsf{Compose}(\mathtt{value}, \mathbf{T}, \mathbf{S}) \land$
$\quad \mathtt{head} \mapsto \mathtt{value}, tail \ast \mathsf{List}(tail, \mathbf{T}) \land$
$\quad \mathtt{value} \in \mathbf{S}$

Close $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$.

$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \land \mathtt{value} \in \mathbf{S}$

$a \in \mathbf{B} \Rightarrow \{a\} \subseteq \mathbf{B}$

$\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S}) \wedge \{\texttt{value}\} \subseteq \mathbf{S}$

$\mathbf{A} \subseteq \mathbf{B} \Rightarrow \mathbf{B} \cup \mathbf{A} = \mathbf{B}$

$\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S}) \wedge \{\texttt{value}\} \cup \mathbf{S} = \mathbf{S}$

Substitution. Discard $\{\texttt{value}\} \cup \mathbf{S} = \mathbf{S}$.

$\mathsf{NonEmptyList}(\ \texttt{head}, \{\texttt{value}\} \cup \mathbf{S}\ )$

```
    o = head;
```

Assignment.

$\mathsf{NonEmptyList}(\ \texttt{o}, \{\texttt{value}\} \cup \mathbf{S}\ )$

```
}
else {
```

Deny if-condition.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad v \neq \texttt{value}$

```
    if (head.value < value) {
```

Assert if-condition.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad v < \texttt{value}$

```
      Node* ntail = insert(head.tail, value);
```

Use specification for `insert`.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(\texttt{ntail}, \mathbf{T} \cup \{\texttt{value}\}) \wedge$
$\quad v < \texttt{value}$

```
      head.tail = ntail;
```

Assignment. Reintroduce existential quantification.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T} \cup \{\texttt{value}\}) \wedge$
$\quad v < \texttt{value}$

Lemma: $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge v < \texttt{value} \Rightarrow \mathsf{Compose}(\ v, \mathbf{T} \cup \{\texttt{value}\}, \mathbf{S} \cup \{\texttt{value}\}\ )$

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T} \cup \{\texttt{value}\}, \mathbf{S} \cup \{\texttt{value}\}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T} \cup \{\texttt{value}\})$

Introduce existential quantification on $\mathbf{T} \cup \{\texttt{value}\}$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S} \cup \{\texttt{value}\}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T})$

Close $\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S} \cup \{\texttt{value}\})$.

$\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S} \cup \{\texttt{value}\})$

```
o = head;
```

Assignment.

$\mathsf{NonEmptyList}(\texttt{o}, \mathbf{S} \cup \{\texttt{value}\})$

```
}
else {
```

Deny if-condition. Use $\neg(v < \texttt{value}) \Rightarrow \texttt{value} \leq v$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad \texttt{value} \neq v \wedge \texttt{value} \leq v$

$(a \neq b \wedge a \leq b) \Rightarrow b < a$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad \texttt{value} < v$

```
nhead = new Node(value, head);
```

Specification for `new Node(value, head)`.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad \texttt{value} < v$

Close $\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S})$.

$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge \texttt{value} < v \wedge$
$\texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \mathsf{NonEmptyList}(\texttt{head}, \mathbf{S})$

Lemma: $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge \texttt{value} < v \Rightarrow \mathsf{Compose}(\texttt{value}, \mathbf{S}, \mathbf{S} \cup \{\texttt{value}\})$

$\mathsf{Compose}(\texttt{value}, \mathbf{S}, \mathbf{S} \cup \{\texttt{value}\}) \wedge$
$\texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \mathsf{NonEmptyList}(\texttt{head}, \mathbf{S})$

Weakening lemma: $\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\texttt{head}, \mathbf{S})$.

$\mathsf{Compose}(\texttt{value}, \mathbf{S}, \mathbf{S} \cup \{\texttt{value}\}) \wedge$
$\texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \mathsf{List}(\texttt{head}, \mathbf{S})$

$\exists$I on $\mathbf{S}$ as $\mathbf{T}$, `head` as *tail*, and `value` as v.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S} \cup \{\texttt{value}\}) \wedge$
$\quad \texttt{nhead} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T})$

Close $\mathsf{NonEmptyList}(\texttt{nhead}, \mathbf{S} \cup \{\texttt{value}\})$.

$\mathsf{NonEmptyList}(\texttt{nhead}, \mathbf{S} \cup \{\texttt{value}\})$

```
o = nhead;
```

Assignment.

$\mathsf{NonEmptyList}(\texttt{o}, \mathbf{S} \cup \{\texttt{value}\})$

```
}
```

If-rule.

NonEmptyList(o, S ∪ {value})

```
    }
```

If-rule.

NonEmptyList(o, S ∪ {value})

```
  }
```

If-rule.

NonEmptyList(o, S ∪ {value})

```
  return o;
}
```

LL `remove` algorithms

A recursive LL `remove` algorithm

The purpose of an `insert` algorithm is to mutate the heap representing some set S such that it represents the set S \ {`value`} for some given parameter `value`. A function implementing `remove` returns a pointer into the new heap. A specification for our LL data structure falls naturally out of this description:

{ List(head, S) } { `nhead = insert(head, value);` } { List(nhead, S \ {value}) }

The recursive `remove` algorithm follows a similar pattern to the `search` and `insert` algorithms we have seen. We have two cases: the list is empty or it is not. If it is empty, it represents ∅; we must then return a list representing ∅ \ {`value`}, which is ∅; we can therefore simply return `head`. If it is not empty, then as before, we compare `value` and the value v at the head of the list. If they are equal, then S \ {`value`} = T represented by the *tail* pointer; we delete the `head` node and return *tail*. If `value` < v, then T cannot contain `value`, so S \ {`value`} = S, and again we can return the `head` pointer. Otherwise, `value` > v, and we must recursively apply `remove` to the *tail* pointer.

image/svg+xml NonEmptyList(*head*, S) The list is non-empty. The values *v* and *tail* areunknown. **S** = {*v*} ∪ **T** and ∀ *t*∈**T**. *v* < *t*. *v* *head*

tail List(*tail*, **T**) **Precondition:** *head* points to a list representingset **S**. We are removing *value*. *head* List(*head*, **S**) The list is empty, and

represents the set ∅.∅ \ {*value*} = ∅. Return *head*. *head* EmptyList(*head*, **S**) **Is the pointer *head* null?** *head* is not null *head* is null

NonEmptyList(*head*, **S**) *value* ∉ {*v*}. All in **T** are > *v*, so *value* ∉ **T**. So*value* ∉ **S**, so **S** \ {*value*} = **S**. Return *head*. *v*> *value* *head* *tail* List(*tail*,

T) NonEmptyList(*head*, {*value*} ∪ **S**) **S** \ {*value*} = **T**. But we already have a listrepresenting T ... *value* *head* *tail* List(*tail*, **T**)

NonEmptyList(*head*, S) The head of the list is consistent with *value* ∉ **S**.We must remove *value* from the tail. *v*< *value* *head* *tail* List(*tail*, **T**)

compare(*value*, *v*) *value* < *v* *value* = *v* *value* > *v* NonEmptyList(*head*, **S** \ {*value*}) *head* now points to a list representing {*v*} ∪ **T** \ {*value*}, which bycommutativity and substitution is $\mathbf{S} \setminus \{value\}$. Return *head*. $v<$ *value head ntail* List(*tail*, $\mathbf{T}$) remove , *value* List(*ntail*, $\mathbf{T} \setminus$

$\{value\}$) **Recursively apply remove to *tail*, yielding newpointer *ntail*. Set *tail* field of first node to *ntail*.** We have List(*tail*, $\mathbf{T}$), but $\mathbf{S} \setminus$

$\{value\} = \mathbf{T}$, soList(*tail*, $\mathbf{S} \setminus \{value\}$). Return *tail*. *head tail* List(*tail*, $\mathbf{T}$) **Read the value tail into local variable.Delete *head* node.**

The annotated code for recursive `remove` follows.

```
module ll.remove.recursive;

import ll.node;

Node* remove(Node* head, int value) {
  Node* o;
```

Precondition.

List(head, $\mathbf{S}$)

```
  if (head == null) {
```

Assert if-condition.

List(head, $\mathbf{S}$) $\wedge$ head = null

Lemma: List(head, $\mathbf{S}$) $\wedge$ head = null $\Rightarrow$ EmptyList(head, $\mathbf{S}$)

EmptyList(head, $\mathbf{S}$)

EmptyList(head, $\mathbf{S}$) $\Rightarrow \mathbf{S} = \varnothing$

EmptyList(head, $\mathbf{S}$) $\wedge\ \mathbf{S} = \varnothing$

$\varnothing \setminus \mathbf{X} = \varnothing$, $\mathbf{S} = \varnothing$, substitution.

EmptyList(head, $\mathbf{S} \setminus$ {value})

Weakening lemma: EmptyList(head, $\mathbf{S}$) $\Rightarrow$ List(head, $\mathbf{S}$).

List(head, $\mathbf{S} \setminus$ {value})

```
    o = head;
```

Assignment.

List(o, $\mathbf{S} \setminus$ {value})

```
  }
  else {
```

Deny if-condition.

List(head, $\mathbf{S}$) $\wedge$ head $\neq$ null

Lemma: List(head, $\mathbf{S}$) $\wedge$ head $\neq$ null $\Rightarrow$ NonEmptyList(head, $\mathbf{S}$).

NonEmptyList(head, $\mathbf{S}$)

Open NonEmptyList(head, $\mathbf{S}$).

$\exists v, tail, \mathbf{T}.$
Compose(v, $\mathbf{T}$, $\mathbf{S}$) $\wedge$

$\text{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T})$

```
if (head.value == value) {
```

Assert if-condition: substitute `value` for v.

$\exists tail, \mathbf{T}.$
$\quad \mathsf{Compose}(\texttt{value}, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto \texttt{value}, tail * \mathsf{List}(tail, \mathbf{T})$

Lemma: $\mathsf{Compose}(\texttt{value}, \mathbf{T}, \mathbf{S}) \Rightarrow \mathbf{T} = \mathbf{S} \setminus \{\texttt{value}\}$. Discard $\mathsf{Compose}(\texttt{value}, \mathbf{T}, \mathbf{S})$.

$\exists tail, \mathbf{T}.$
$\quad \texttt{head} \mapsto \texttt{value}, tail * \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad \mathbf{T} = \mathbf{S} \setminus \{\texttt{value}\}$

Substitution. Discard $\mathbf{T} = \mathbf{S} \setminus \{\texttt{value}\}$.

$\exists tail.$
$\quad \texttt{head} \mapsto \texttt{value}, tail * \mathsf{List}(tail, \mathbf{S} \setminus \{\texttt{value}\})$

```
  o = head.tail;
```

Assignment.

$\texttt{head} \mapsto \texttt{value}, \texttt{o} * \mathsf{List}(\texttt{o}, \mathbf{S} \setminus \{\texttt{value}\})$

```
  delete head;
```

Release heap chunk.

$\mathsf{List}(\ \texttt{o}, \mathbf{S} \setminus \{\texttt{value}\}\ )$

```
}
else {
```

Deny if-condition.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad v \neq \texttt{value}$

```
  if (head.value > value) {
```

Assert if-condition.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T}) \wedge$
$\quad v < \texttt{value}$

```
    Node* ntail = remove(head.tail, value);
```

Use specification for `remove`.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, tail * \mathsf{List}(\texttt{ntail}, \mathbf{T} \setminus \{\texttt{value}\}) \wedge$
$\quad v < \texttt{value}$

```
    head.tail = ntail;
```

Assignment.

$\exists v, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge$
$\quad \texttt{head} \mapsto v, \texttt{ntail} * \mathsf{List}(\texttt{ntail}, \mathbf{T} \setminus \{\texttt{value}\}) \wedge$
$\quad v < \texttt{value}$

∃I on `ntail` as *tail*.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
$\quad \mathtt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T} \setminus \{\mathtt{value}\}) \land$
$\quad v < \mathtt{value}$

Lemma: $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land v \neq \mathtt{value} \Rightarrow \mathsf{Compose}(v, \mathbf{T} \setminus \{\mathtt{value}\}, \mathbf{S} \setminus \{\mathtt{value}\})$. Discard $v < \mathtt{value}$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T} \setminus \{\mathtt{value}\}, \mathbf{S} \setminus \{\mathtt{value}\}) \land$
$\quad \mathtt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T} \setminus \{\mathtt{value}\})$

∃I on $\mathbf{T} \setminus \{\mathtt{value}\}$ as $\mathbf{T}$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S} \setminus \{\mathtt{value}\}) \land$
$\quad \mathtt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T})$

Close $\mathsf{NonEmptyList}(\ \mathtt{head}, \mathbf{S} \setminus \{\mathtt{value}\}\ )$.

$\mathsf{NonEmptyList}(\ \mathtt{head}, \mathbf{S} \setminus \{\mathtt{value}\}\ )$

Weakening lemma: $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\mathtt{head}, \mathbf{S})$.

$\mathsf{List}(\ \mathtt{head}, \mathbf{S} \setminus \{\mathtt{value}\}\ )$

```
    o = head;
```

Assignment.

$\mathsf{List}(\ \mathtt{o}, \mathbf{S} \setminus \{\mathtt{value}\}\ )$

```
}
else {
```

Deny if-condition. Use $\neg(v < \mathtt{value}) \Rightarrow \mathtt{value} \leq v$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
$\quad \mathtt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T}) \land$
$\quad \mathtt{value} \neq v \land \mathtt{value} \leq v$

$(a \neq b \land a \leq b) \Rightarrow b < a$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
$\quad \mathtt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T}) \land$
$\quad \mathtt{value} < v$

Lemma: $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land \mathtt{value} < v \Rightarrow \mathtt{value} \notin \mathbf{S}$. Discard $\mathtt{value} < v$.

$\exists v, tail, \mathbf{T}.$
$\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
$\quad \mathtt{head} \mapsto v, tail * \mathsf{List}(tail, \mathbf{T}) \land$
$\quad \mathtt{value} \notin \mathbf{S}$

Close $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S})$.

$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \land \mathtt{value} \notin \mathbf{S}$

$a \notin \mathbf{X} \Rightarrow \mathbf{X} \setminus \{a\} = \mathbf{X}$. Discard $\mathtt{value} \notin \mathbf{S}$.

$\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \land \mathbf{S} \setminus \{\mathtt{value}\} = \mathbf{S}$

```
          Substitution. Discard S \ {value} = S.
          NonEmptyList( head, S \ {value} )
          Weakening lemma: NonEmptyList(head, S) ⇒ List(head, S).
          List( head, S \ {value} )
          o = head;
          Assignment.
          List( o, S \ {value} )
        }
        If-rule.
        List( o, S \ {value} )
      }
      If-rule.
      List( o, S \ {value} )
    }
    If-rule.
    List( o, S \ {value} )
  return o;
}
```

Verifying the BST

The second data structure I address in this project is the BST. As before, I begin with a description of the data structure, then verify recursive algorithms for `search`, `insert`, and `remove`.

The BST data structure

At each `Node` of an LL, we had a link to a smaller LL: the data structure contained a single smaller version of itself. The sole difference that the binary tree introduces is that it contains *two* smaller binary trees. In the LL, the set was split into the element v at the `head` node and the set $\mathbf{T}$ held in the `tail` list. In the binary tree, the same set is split into an element v at the `root` node (analogous to the `head` node in the LL), and *two* sets $\mathbf{L}$ and $\mathbf{R}$, held respectively in what are called the `left` and `right` subtrees.

In our LL, we chose to order the elements of the set $\mathbf{S}$ in strictly ascending order; that is, the element v was chosen as the minimal element of $\mathbf{S}$. Such an ordered linked list might be termed a *Linked Search List*. Similarly, we can order the elements of the binary tree. Here, the choice of v from $\mathbf{S}$ is arbitrary. The set $\mathbf{L}$ is then chosen to hold all elements of $\mathbf{S}$ that are strictly less than v, and the set $\mathbf{R}$ holds all elements strictly greater than v. This particular type of binary tree with its elements ordered is called a *Binary Search Tree*, which I will hereon abbreviate as BST.

As with the LL, a pointer into the `root` node is required to complete the representation of the set. Also as with the LL, $\varnothing$ has a distinct representation: a pointer to `null` and no allocated heap chunks.

We described a general pointer `head` into a list representing the set $\mathbf{S}$ with the predicate List(`head`, $\mathbf{S}$). Similarly, we will now describe a general pointer `root` into a BST representing $\mathbf{S}$ with a predicate Tree(`root`, $\mathbf{S}$).

Where the two representations defined by the LL were termed EmptyList and NonEmptyList, we will analogously define EmptyTree and NonEmptyTree. As with our visualization of the LL, we can view the BST predicates graphically:

image/svg+xml *root* EmptyTree(*root*, $\mathbf{S}$),where $\mathbf{S} = \varnothing$. Tree(*root*, $\mathbf{S}$) NonEmptyTree(*root*, $\mathbf{S}$),where $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}, \forall\, l \in \mathbf{L}.\ l < v$, and $\forall$

$r \in \mathbf{R}.\ v < r.$ *left* *v* *right* *root* **L** Tree(*left*, **L**) **R**

Tree(*right*, **R**)

The Tree predicate is defined in terms of the two concrete representations:

$$\mathsf{Tree}(\texttt{root}, \mathbf{S}) \stackrel{\text{def}}{=} \mathsf{EmptyTree}(\texttt{root}, \mathbf{S}) \vee \mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S}).$$

The EmptyTree predicate is in fact just EmptyList under a new name:

$\mathsf{EmptyTree}(n, S) \stackrel{\text{def}}{=}$	$n = \texttt{null} \wedge$	An empty tree is indicated by a null pointer,
	$\mathbf{emp} \wedge$	uses no space on the heap, and
	$S = \varnothing.$	represents the empty set.

To describe the NonEmptyTree, we first define the ordering of its elements. We need a new BST Compose predicate on the total set **S**, the root element v, and the subsets **L** and **R**:

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \stackrel{\text{def}}{=}$	$\mathbf{L} \cup \{v\} \cup \mathbf{R} = \mathbf{S} \wedge$	**L**, v, and **R** make up the set **S**,

$$\forall\, l \in \mathbf{L}\,.\; l < v \;\wedge \qquad \text{all values in } \mathbf{L} \text{ are less than } v\text{, and}$$
$$\forall\, r \in \mathbf{R}\,.\; v < r\,. \qquad \text{all values in } \mathbf{R} \text{ are greater than } v.$$

Using our Compose predicate, we can describe a tree at address `root` that stores the value v at the root and represents the set $\mathbf{S}$. We define an intermediary predicate, TopOfTree:

$\mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S}) \stackrel{\text{def}}{=}$	$\exists l,r,\mathbf{L},\mathbf{R}.$	
	$\mathtt{root} \mapsto v,l,r$	`root` points to a `Node` with value v and pointers l and r
	$*\ \mathsf{Tree}(l, \mathbf{L})$	where l points into a BST representing $\mathbf{L}$,
	$*\ \mathsf{Tree}(r, \mathbf{R})$	r points into a BST representing $\mathbf{R}$, and
	$\wedge\ \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}).$	the values in the tree are totally ordered.

The TopOfTree predicate allows us to gradually expand our knowledge of the tree as an algorithm proceeds. Before we know the value at the root, though, we just know it's non-empty:

$$\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S}) \stackrel{\text{def}}{=} \quad \exists v.\ \mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S})$$

Here is the module defining the structure of a node in the tree:

```
module bst.node;

struct Node {
  int value;   // One value in the set held in this subtree
  Node*[2] c;  // Pointers to the two subtrees (0 is left, 1 is right)

  this(int value) {
    this.value = value;  // The subtree pointers are initialized to null.
  }
}
```

Notice the use of a 'link array': rather than separate named `left` and `right` pointers, these two pointers are held in a two-element array, addressed respectively as `c[0]` and `c[1]`. This enables us to parameterize procedures by the index where the symmetry of the tree would otherwise demand two symmetrical blocks of code.

Lemmata used in BST algorithms

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \wedge \mathtt{root} = \mathtt{null} \Rightarrow \mathsf{EmptyTree}(\mathtt{root}, \mathbf{S})$

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \Rightarrow \mathbf{S} = \varnothing$

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \Rightarrow \mathsf{Tree}(\mathtt{root}, \mathbf{S})$

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \wedge \mathtt{root} \neq \mathtt{null} \Rightarrow \mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S})$

$\mathsf{TopOfTree}(\mathtt{root}, \mathtt{value}, \mathbf{S}) \Rightarrow \mathtt{value} \in \mathbf{S}$

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathtt{value} < v \Rightarrow \mathtt{value} \in \mathbf{L} \leftrightarrow \mathtt{value} \in \mathbf{S}$

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathtt{value} < v \Rightarrow \mathsf{Compose}(\ \mathbf{L} \cup \{\mathtt{value}\}, v, \mathbf{R}, \mathbf{S} \cup \{\mathtt{value}\}\ )$

Symmetrical: $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge v < \mathtt{value} \Rightarrow \mathsf{Compose}(\ \mathbf{L}, v, \mathbf{R} \cup \{\mathtt{value}\}, \mathbf{S} \cup \{\mathtt{value}\}\ )$

$\mathsf{Compose}(\mathbf{L}, v, \varnothing, \mathbf{S}) \Rightarrow \mathsf{Max}(v, \mathbf{S})$

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(r, \mathbf{R}) \Rightarrow \mathsf{Compose}(\mathbf{L}, v, \mathbf{R} \setminus \{r\}, \mathbf{S} \setminus \{r\}) \wedge \mathsf{Max}(r, \mathbf{S})$

$\mathsf{Compose}(\varnothing, v, \mathbf{R}, \mathbf{S}) \Rightarrow \mathbf{R} = \mathbf{S} \setminus \{v\}$

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(\mathtt{nv}, \mathbf{L}) \Rightarrow \mathsf{Compose}(\mathbf{L} \setminus \{\mathtt{nv}\}, \mathtt{nv}, \mathbf{R}, \mathbf{S} \setminus \{v\})$

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathtt{value} < v \Rightarrow \mathsf{Compose}(\mathbf{L} \setminus \{\mathtt{value}\}, v, \mathbf{R}, \mathbf{S} \setminus \{\mathtt{value}\})$

Recursive BST algorithms

Recursive search

image/svg+xml *head* points to a non-empty tree with v at theroot and two, possibly empty, subtrees. NonEmptyTree(*head*, $\mathbf{S} = \mathbf{L} \cup \{v\}$

$\cup$ **R**) *left* v *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) **Precondition:** *head* is a tree representing someset **S**. We are searching for *value*. **S**

head Tree(*head*, **S**) The tree at *head* is empty, and represents theset $\varnothing$. *value* $\notin \varnothing$. Return false. $\mathbf{S} = \varnothing$ *head* = null EmptyTree(*head*, **S**) **Is**

***head* null?** *head* = null *head* $\neq$ null *value* $\in \{v\}$, so *value* $\in \mathbf{L} \cup \{v\} \cup \mathbf{R}$, so*value* $\in S$. Return true. NonEmptyTree(*head*, $\mathbf{S} = \mathbf{L} \cup \{value\} \cup$

R) *left* *value* *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) *value* $\notin \{v\}$, and *value* $\notin \mathbf{R} \because \forall r \in \mathbf{R}.\ v<r$, so(*value* $\in$ **S**) $\leftrightarrow$ (*value* $\in$ **L**). Search *left*.

NonEmptyTree(*head*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *left* v>*value* *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) **compare(*value*, *v*)** *value* < v *value* = v *value*

> v *value* $\notin \{v\}$, and *value* $\notin \mathbf{L} \because \forall l \in \mathbf{L}.\ v>l$, so(*value* $\in$ **S**) $\leftrightarrow$ (*value* $\in$ **R**). Search *right*. NonEmptyTree(*head*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *left* v<*value*

right *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**)

```
module bst.search.recursive;

import bst.node;
import bst.descend;


bool search(Node* root, in int value) {
  bool o;
```

Function precondition.

$\mathsf{Tree}(\mathtt{root}, \mathbf{S})$

```
  if (root == null) {
```

Assert if-condition.

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \wedge \mathtt{root} = \mathtt{null}$

Lemma: $\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \wedge \mathtt{root} = \mathtt{null} \Rightarrow \mathsf{EmptyTree}(\mathtt{root}, \mathbf{S})$

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S})$

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \Rightarrow \mathbf{S} = \varnothing$

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \land \mathbf{S} = \varnothing$

$\mathbf{S} = \varnothing \Rightarrow \mathtt{value} \notin \mathbf{S}$

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \land \mathtt{value} \notin \mathbf{S}$

```
o = false;
```

Assignment.

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \land \mathtt{value} \notin \mathbf{S} \land \mathtt{o} = \mathtt{false}$

?

$\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \land \mathtt{o} \leftrightarrow (\mathtt{value} \in \mathbf{S})$

Weakening lemma: $\mathsf{EmptyTree}(\mathtt{root}, \mathbf{S}) \Rightarrow \mathsf{Tree}(\mathtt{root}, \mathbf{S})$

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \land \mathtt{o} \leftrightarrow (\mathtt{value} \in \mathbf{S})$

```
}
else {
```

Deny if-condition.

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \land \mathtt{root} \neq \mathtt{null}$

Lemma: $\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \land \mathtt{root} \neq \mathtt{null} \Rightarrow \mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S})$

$\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S})$

```
  bool eq = rootEq(root, value);
```

Specification for `rootEq`.

$\exists v.\ \mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S}) \land \mathtt{eq} \leftrightarrow v = \mathtt{value}$

```
  if (eq) {
```

Assert if-test.

$\exists v.\ \mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S}) \land \mathtt{eq} \leftrightarrow v = \mathtt{value} \land \mathtt{eq}$

Use $\mathtt{eq} \leftrightarrow v = \mathtt{value}$. Discard `eq`.

$\exists v.\ \mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S}) \land v = \mathtt{value}$

Substitution.

$\mathsf{TopOfTree}(\mathtt{root}, \mathtt{value}, \mathbf{S})$

Lemma: $\mathsf{TopOfTree}(\mathtt{root}, \mathtt{value}, \mathbf{S}) \Rightarrow \mathtt{value} \in \mathbf{S}$

$\mathsf{TopOfTree}(\mathtt{root}, \mathtt{value}, \mathbf{S}) \land \mathtt{value} \in \mathbf{S}$

Reintroduce $\exists$I on `value` as v.

$\exists v.\ \mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S}) \land \mathtt{value} \in \mathbf{S}$

Close $\mathsf{Tree}(\mathtt{root}, \mathbf{S})$.

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \land \mathtt{value} \in \mathbf{S}$

```
    o = true;
```

Assignment.

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \land \mathtt{value} \in \mathbf{S} \land \mathtt{o} = \mathtt{true}$

?

$\mathsf{Tree}(\mathtt{root}, \mathbf{S}) \land \mathtt{o} \leftrightarrow \mathtt{value} \in \mathbf{S}$

```
}
else {
```

Deny if-condition.

$\exists v.\ \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S}) \land \texttt{eq} \leftrightarrow v = \texttt{value} \land \neg\texttt{eq}$

Use $\texttt{eq} \to (v = \texttt{value})$. Discard $\neg\texttt{eq}$.

$\exists v.\ \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S}) \land v \neq \texttt{value}$

Open $\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad v \neq \texttt{value}$

```
  if (value < root.value) {
```

Assert if-condition.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} < v$

Lemma: $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \texttt{value} < v \Rightarrow \texttt{value} \in \mathbf{L} \leftrightarrow \texttt{value} \in \mathbf{S}$. Discard $\texttt{value} < v$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} \in \mathbf{L} \leftrightarrow \texttt{value} \in \mathbf{S}$

```
    o = search(root.left, value);
```

Use specification for search.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} \in \mathbf{L} \leftrightarrow \texttt{value} \in \mathbf{S} \land$
$\quad \texttt{o} \leftrightarrow (\texttt{value} \in \mathbf{L})$

Transitivity of double implication.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{o} \leftrightarrow (\texttt{value} \in \mathbf{S})$

Close $\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})$.

$\exists v.$
$\quad \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S}) \land \texttt{o} \leftrightarrow (\texttt{value} \in \mathbf{S})$

Close $\mathsf{Tree}(\texttt{root}, \mathbf{S})$.

$\mathsf{Tree}(\texttt{root}, \mathbf{S}) \land \texttt{o} \leftrightarrow (\texttt{value} \in \mathbf{S})$

```
  }
  else {
```

(Symmetrical case…)

```
    o = search(root.right, value);
```

```
      Tree(root, S) ∧ o ↔ (value ∈ S)
    }
      If-rule.
      Tree(root, S) ∧ o ↔ (value ∈ S)
    }
    If-rule.
    Tree(root, S) ∧ o ↔ (value ∈ S)
  }
  If-rule. Function postcondition.
  Tree(root, S) ∧ o ↔ (value ∈ S)
  return o;
}
```

Recursive `insert`

image/svg+xml *head* points to a non-empty tree with some *v* at the root and two, possibly empty, subtrees. NonEmptyTree(*head*, **S** = **L** ∪ {*v*} ∪ **R**) *left* *v* *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) **Precondition:** *head* is a tree representing someset **S**. We are inserting *value*. **S**

head Tree(*head*, **S**) The tree at *head* is empty, and represents theset ∅. We can return a one-node tree. **S** = ∅ *head* = null

EmptyTree(*head*, **S**) **Is *head* null?** *head* ≠ null *head* = null *value* ∈ {*v*}, so *value* ∈ **L** ∪ {v} ∪ **R**, so*value* ∈ **S**, so **S** ∪ {*value*} = **S**. Return *head*.

NonEmptyTree(*head*, **S** = **L** ∪ {*value*} ∪ **R**) *left* *value* *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) We cannot insert into the *right* tree,

becausethat would break ∀*r*∈**R**. *v*<*r*. NonEmptyTree(*head*, **S** = **L** ∪ {*v*} ∪ **R**) *left* *v*>*value* *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**)

compare(*value*, *v*) *value* < *v* *value* = *v* *value* > *v* We cannot insert into the *left* tree, becausethat would break ∀*l*∈**L**. *v*>*l*.

NonEmptyTree(*head*, **S** = **L** ∪ {*v*} ∪ **R**) *left* *v*<*value* *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) We require ∅ ∪ {*value*} = {*value*}; new tree

is∅ ∪ {*value*} ∪ ∅ = {*value*}. Return *nhead*. NonEmptyTree(*head*, ∅ ∪ {*value*} ∪ ∅) null *value* null EmptyTree(null, ∅) *nhead* ∅ ∅ **S** = ∅ *head*

= null **Construct one-node tree containing *value*,pointed at by new variable *nhead*.** Tree represents **L** ∪ {*value*} ∪ {*v*} ∪ **R**, which= **S** ∪

{*value*}. Return *head*. NonEmptyTree(*head*, **L** ∪ {*value*} ∪ {*v*} ∪ **R**) *nleft* *v*>*value* *right* *head* **R** Tree(*right*, **R**) **L** insert ,

NonEmptyTree(*nleft*,**L** ∪ {*value*}) *value* **Recursively call insert(*left*, *value*), yielding*nleft*; set *left* field to *nleft*.** Tree represents **L** ∪ {*v*} ∪

$\mathbf{R} \cup \{value\}$, which= $\mathbf{S} \cup \{value\}$. Return *head*. NonEmptyTree(*head*, $\mathbf{L} \cup \{v\} \cup \mathbf{R} \cup \{value\}$) *left* *v*>*value* *nright* *head* insert ,

NonEmptyTree(*nright*,$\mathbf{R} \cup \{value\}$) *value* **R** **L** Tree(*left*, **L**) **Recursively call insert(*right*, *value*), yielding*nright*; set *right* field to**

***nright*.**

```
module bst.insert.recursive;

import bst.node;

Node* insert(Node* root, int value) {
  Node* o;
```

Function precondition.

Tree(root, $\mathbf{S}$)

```
  if (root == null) {
```

Assert if-condition.

Tree(root, $\mathbf{S}$) $\wedge$ root = null

Lemma: Tree(root, $\mathbf{S}$) $\wedge$ root = null $\Rightarrow$ EmptyTree(root, $\mathbf{S}$)

EmptyTree(root, $\mathbf{S}$)

Lemma: EmptyTree(root, $\mathbf{S}$) $\Rightarrow \mathbf{S} = \varnothing$

EmptyTree(root, $\mathbf{S}$) $\wedge\ \mathbf{S} = \varnothing$

```
    o = new Node(value);
```

NonEmptyTree(root, $\mathbf{S} \cup$ {value})

```
  }
  else {
```

Deny if-condition.

Tree(root, $\mathbf{S}$) $\wedge$ root $\neq$ null

Lemma: Tree(root, $\mathbf{S}$) $\wedge$ root $\neq$ null $\Rightarrow$ NonEmptyTree(root, $\mathbf{S}$)

NonEmptyTree(root, $\mathbf{S}$)

```
    bool eq = rootEq(root, value);
```

Specification for rootEq.

$\exists v.$ TopOfTree(root, v, $\mathbf{S}$) $\wedge$ eq $\leftrightarrow v$ = value

```
    if (eq) {
```

Assert if-test.

$\exists v.$ TopOfTree(root, v, $\mathbf{S}$) $\wedge$ eq $\leftrightarrow v$ = value $\wedge$ eq

Use eq $\leftrightarrow v$ = value. Discard eq.

$\exists v.$ TopOfTree(root, v, $\mathbf{S}$) $\wedge$ v = value

Substitution.

$\mathsf{TopOfTree}(\texttt{root}, \texttt{value}, \mathbf{S})$

```
o = root;
```

$\mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S} \cup \{\texttt{value}\})$

```
}
else {
```

Deny if-condition.

$\exists v.\ \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S}) \land \texttt{eq} \leftrightarrow v = \texttt{value} \land \neg\texttt{eq}$

Use $\texttt{eq} \rightarrow (v = \texttt{value})$. Discard $\neg\texttt{eq}$.

$\exists v.\ \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S}) \land v \neq \texttt{value}$

Open $\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad v \neq \texttt{value}$

```
if (value < root.value) {
```

Assert if-condition.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} < v$

```
Node* left = root.c[0];
```

$\exists$E of l as $\texttt{left}$.

$\exists v, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, \texttt{left}, r \ast \mathsf{Tree}(\texttt{left}, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} < v$

```
Node* nleft = insert(left, value);
```

Specification for $\texttt{insert}$.

$\exists v, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, \texttt{left}, r \ast \mathsf{Tree}(\texttt{nleft}, \mathbf{L} \cup \{\texttt{value}\}) \ast \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} < v$

```
root.c[0] = nleft;
```

Assignment.

$\exists v, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, \texttt{nleft}, r \ast \mathsf{Tree}(\texttt{nleft}, \mathbf{L} \cup \{\texttt{value}\}) \ast \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} < v$

$\exists$I on $\texttt{nleft}$ as l.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, r \ast \mathsf{Tree}(l, \mathbf{L} \cup \{\texttt{value}\}) \ast \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land$
$\quad \texttt{value} < v$

Lemma: $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathtt{value} < v \Rightarrow \mathsf{Compose}(\ \mathbf{L} \cup \{\mathtt{value}\}, v, \mathbf{R}, \mathbf{S} \cup \{\mathtt{value}\}\ )$.
Discard $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$ and $\mathtt{value} < v$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \mathtt{root} \mapsto v, l, r * \mathsf{Tree}(l, \mathbf{L} \cup \{\mathtt{value}\}) * \mathsf{Tree}(r, \mathbf{R}) \wedge$
$\quad \mathsf{Compose}(\ \mathbf{L} \cup \{\mathtt{value}\}, v, \mathbf{R}, \mathbf{S} \cup \{\mathtt{value}\}\ )$

∃I on $\mathbf{L} \cup \{\mathtt{value}\}$ as $\mathbf{L}$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \mathtt{root} \mapsto v, l, r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \wedge$
$\quad \mathsf{Compose}(\ \mathbf{L}, v, \mathbf{R}, \mathbf{S} \cup \{\mathtt{value}\}\ )$

Close $\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S} \cup \{\mathtt{value}\})$.

$\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S} \cup \{\mathtt{value}\})$

```
    }
    else {
```

(Symmetrical case…)

```
      root.c[1] = insert(root.c[1], value);
```

$\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S} \cup \{\mathtt{value}\})$

```
    }
```

If-rule.

$\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S} \cup \{\mathtt{value}\})$

```
    o = root;
```

Assignment.

$\mathsf{NonEmptyTree}(\mathtt{o}, \mathbf{S} \cup \{\mathtt{value}\})$

```
  }
```

If-rule.

$\mathsf{NonEmptyTree}(\mathtt{o}, \mathbf{S} \cup \{\mathtt{value}\})$

```
 }
```

If-rule.

$\mathsf{NonEmptyTree}(\mathtt{o}, \mathbf{S} \cup \{\mathtt{value}\})$

```
 return o;
}
```

Recursive `remove`

image/svg+xml *head* points to a non-empty tree with some *v* at the root and two, possibly empty, subtrees. NonEmptyTree(*head*, **S** = **L** ∪ {*v*} ∪ **R**) *left* *v* *right* *head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) **Precondition:** *head* is a tree representing someset **S**. We are removing *value*. **S** *head* Tree(*head*, **S**) The tree at *head* is empty, and represents theset ∅. ∅ \ {*value*} = ∅. Return *head*. **S** = ∅ *head* = null EmptyTree(*head*, **S**) **Is *head* null?** *head* ≠ null *head* = null We need to remove the value at the root. Wehave a helper function for that: removeRoot.

NonEmptyTree(*head*, $\mathbf{S} = \mathbf{L} \cup \{value\} \cup \mathbf{R}$) *left value right head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) We cannot insert into the *right* tree,

becausethat would break $\forall r \in \mathbf{R}.\ v<r$. NonEmptyTree(*head*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *left v>value right head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**)

compare(*value*, *v*) *value = v value < v value < v* (symmetrical) Tree represents $(\mathbf{L} \setminus \{value\}) \cup \{v\} \cup \mathbf{R}$, which= $\mathbf{S} \setminus \{value\}$, $\because value \notin \{v\} \cup$

R. Return *head*. NonEmptyTree(*head*, $(\mathbf{L} \setminus \{value\}) \cup \{v\} \cup \mathbf{R}$) *nleft v>value right head* **R** Tree(*right*, **R**) **L** remove , Tree(*nleft*, $\mathbf{L} \setminus \{value\}$) *value*

Recursively call remove(*left*, *value*), yielding*nleft*; set *left* field to *nleft*. removeRoot returns tree representing $\mathbf{L} \cup \mathbf{R}$,which is $\mathbf{S} \setminus \{value\}$.

Pass up return value. NonEmptyTree(*head*, $\mathbf{S} = \mathbf{L} \cup \{value\} \cup \mathbf{R}$) *left value right head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) removeRoot

Tree(*nhead*, $\mathbf{L} \cup \mathbf{R}$) ***nhead* = removeRoot(*head*). Return *nhead*.**

$\mathsf{Max}(m, \mathbf{S}) \stackrel{\text{def}}{=} m \in \mathbf{S} \wedge \forall s \in \mathbf{S}.\ s \leq m$

Here's `removeMax`:

```
module bst.remove.removeMax.recursive;


import bst.node;
import bst.remove.removeMax.RemoveMaxRet;

import std.stdio;

RemoveMaxRet removeMax(Node* root) {
  assert(root != null);

  int max;
  Node* newRoot;
```

Function precondition.

$\mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S})$

Open $\mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S})$.

$\exists v.\ \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})$

Open $\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,l,r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \wedge$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

```
  auto r = root.c[1];
```

∃E on r as `r`.

$\exists v, l, \mathbf{L}, \mathbf{R}.$

$\quad \texttt{root} \mapsto v, l, \texttt{r} \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(\texttt{r}, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

```
if (r == null) {
```

Assert if-condition. Substitution.

$\exists v, l, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, \texttt{null} \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(\texttt{null}, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

Lemma: $\mathsf{Tree}(\texttt{null}, \mathbf{R}) \Rightarrow \mathsf{EmptyTree}(\texttt{null}, \mathbf{R})$

$\exists v, l, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, \texttt{null} \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{EmptyTree}(\texttt{null}, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

Lemma: $\mathsf{EmptyTree}(r, \mathbf{R}) \Rightarrow \mathbf{R} = \emptyset$. Substitution. Discard $\mathbf{R} = \emptyset$.

$\exists v, l, \mathbf{L}.$
$\quad \texttt{root} \mapsto v, l, \texttt{null} \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{EmptyTree}(\texttt{null}, \emptyset) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \emptyset, \mathbf{S})$

Open $\mathsf{EmptyTree}(\texttt{null}, \emptyset)$. Discard $\texttt{null} = \texttt{null}$ and $\emptyset = \emptyset$.

$\exists v, l, \mathbf{L}.$
$\quad \texttt{root} \mapsto v, l, \texttt{null} \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathbf{emp} \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \emptyset, \mathbf{S})$

$\mathsf{X} \ast \mathbf{emp} = \mathsf{X}$.

$\exists v, l, \mathbf{L}.$
$\quad \texttt{root} \mapsto v, l, \texttt{null} \ast \mathsf{Tree}(l, \mathbf{L}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \emptyset, \mathbf{S})$

Lemma: $\mathsf{Compose}(\mathbf{L}, v, \emptyset, \mathbf{S}) \Rightarrow \mathsf{Max}(v, \mathbf{S}) \land \mathbf{L} = \mathbf{S} \setminus \{v\}$.
Discard $\mathsf{Compose}(\mathbf{L}, v, \emptyset, \mathbf{S})$.

$\exists v, l.$
$\quad \texttt{root} \mapsto v, l, \texttt{null} \ast \mathsf{Tree}(l, \mathbf{L}) \land$
$\quad \mathsf{Max}(v, \mathbf{S}) \land \mathbf{L} = \mathbf{S} \setminus \{v\}$

Substitution. Discard $\mathbf{L} = \mathbf{S} \setminus \{v\}$.

$\exists v, l.$
$\quad \texttt{root} \mapsto v, l, \texttt{null} \ast \mathsf{Tree}(l, \mathbf{S} \setminus \{v\}) \land \mathsf{Max}(v, \mathbf{S})$

```
  max = root.value;
  newRoot = root.c[0];
```

Assignment (twice).

$\texttt{root} \mapsto \texttt{max}, \texttt{newRoot}, \texttt{null} \ast \mathsf{Tree}(\texttt{newRoot}, \mathbf{S} \setminus \{\texttt{max}\}) \land \mathsf{Max}(\texttt{max}, \mathbf{S})$

```
  delete root;
```

Free heap chunk.

$\mathsf{Tree}(\texttt{newRoot}, \mathbf{S} \setminus \{\texttt{max}\}) \land \mathsf{Max}(\texttt{max}, \mathbf{S})$

```
}
else {
```

Deny if-condition.

$\exists v, l, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v, l, \texttt{r} \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(\texttt{r}, \mathbf{R}) \land$

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge$
$\texttt{r} \neq \texttt{null}$

Lemma: $\mathsf{Tree}(\texttt{r}, \mathbf{R}) \wedge \texttt{r} \neq \texttt{null} \Rightarrow \mathsf{NonEmptyTree}(\texttt{r}, \mathbf{R})$. Discard $r \neq \texttt{null}$.

$\exists v, l, \mathbf{L}, \mathbf{R}.$
$\texttt{root} \mapsto v{,}l{,}\texttt{r} * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{NonEmptyTree}(\texttt{r}, \mathbf{R}) \wedge$
$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

```
auto d = removeMax(r); auto rightMax = d.max; auto rightRoot = d.root;
```

Specification for `removeMax`.

$\exists v, l, \mathbf{L}, \mathbf{R}.$
$\texttt{root} \mapsto v{,}l{,}\texttt{r} * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(\texttt{rightRoot}, \mathbf{R} \setminus \{\texttt{rightMax}\}) \wedge$
$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{R})$

```
root.c[1] = rightRoot;
```

Assignment.

$\exists v, l, \mathbf{L}, \mathbf{R}.$
$\texttt{root} \mapsto v{,}l{,}\texttt{rightRoot} * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(\texttt{rightRoot}, \mathbf{R} \setminus \{\texttt{rightMax}\}) \wedge$
$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{R})$

$\exists$I on `rightRoot` as r.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\texttt{root} \mapsto v{,}l{,}r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R} \setminus \{\texttt{rightMax}\}) \wedge$
$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{R})$

Lemma: $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(r, \mathbf{R}) \Rightarrow \mathsf{Compose}(\mathbf{L}, v, \mathbf{R} \setminus \{r\}, \mathbf{S} \setminus \{r\}) \wedge \mathsf{Max}(r, \mathbf{S})$.
Discard $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$ and $\mathsf{Max}(\texttt{rightMax}, \mathbf{R})$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\texttt{root} \mapsto v{,}l{,}r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R} \setminus \{\texttt{rightMax}\}) \wedge$
$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R} \setminus \{\texttt{rightMax}\}, \mathbf{S} \setminus \{\texttt{rightMax}\}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{S})$

$\exists$I on $\mathbf{R} \setminus \{\texttt{rightMax}\}$ as $\mathbf{R}$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\texttt{root} \mapsto v{,}l{,}r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \wedge$
$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S} \setminus \{\texttt{rightMax}\}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{S})$

Close $\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S} \setminus \{\texttt{rightMax}\})$.

$\exists v.\ \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S} \setminus \{\texttt{rightMax}\}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{S})$

Close $\mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{rightMax}\})$.

$\mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{rightMax}\}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{S})$

Weaken

$\mathsf{Tree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{rightMax}\}) \wedge \mathsf{Max}(\texttt{rightMax}, \mathbf{S})$

```
max = rightMax;
newRoot = root;
```

Assignment.

$\mathsf{Tree}(\texttt{newRoot}, \mathbf{S} \setminus \{\texttt{max}\}) \wedge \mathsf{Max}(\texttt{max}, \mathbf{S})$

```
}
```

If-rule.

$\mathsf{Tree}(\texttt{newRoot}, \mathbf{S} \setminus \{\texttt{max}\}) \land \mathsf{Max}(\texttt{max}, \mathbf{S})$

```
  RemoveMaxRet o = {max: max, root: newRoot};
  return o;
}
```

Here's `removeRoot`:

```
module bst.removeRoot;

import bst.node;
import bst.remove.removeMax.removeMax;
import bst.remove.removeMax.RemoveMaxRet;


Node* removeRoot(Node* root) {
  Node* o;
```

Function precondition.

$\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})$

Open $\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

```
  if (root.c[0] == null) {
```

Assert if-condition. Substitution.

$\exists v, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,\texttt{null},r * \mathsf{Tree}(\texttt{null}, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

Lemma: $\mathsf{Tree}(\texttt{null}, \mathbf{S}) \Rightarrow \mathsf{EmptyTree}(\texttt{null}, \mathbf{S})$.

$\exists v, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,\texttt{null},r * \mathsf{EmptyTree}(\texttt{null}, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$

Open $\mathsf{EmptyTree}(\texttt{null}, \mathbf{L})$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,\texttt{null},r * \mathbf{emp} * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathbf{L} = \varnothing$

Use $\mathsf{X} * \mathrm{emp} = \mathsf{X}$.

$\exists v, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,\texttt{null},r * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathbf{L} = \varnothing$

Substitution.

$\exists v, r, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,\texttt{null},r * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\varnothing, v, \mathbf{R}, \mathbf{S})$

```
    o = root.c[1];
```

Assignment.

$\exists v, \mathbf{R}.$

$$\texttt{root} \mapsto v,\texttt{null},\texttt{o} \ast \mathsf{Tree}(\texttt{o}, \mathbf{R}) \wedge \mathsf{Compose}(\varnothing, v, \mathbf{R}, \mathbf{S})$$

```
delete root;
```

Free heap chunk.

$$\exists \mathbf{R}.\ \mathsf{Tree}(\texttt{o}, \mathbf{R}) \wedge \mathsf{Compose}(\varnothing, v, \mathbf{R}, \mathbf{S})$$

Lemma: $\mathsf{Compose}(\varnothing, v, \mathbf{R}, \mathbf{S}) \Rightarrow \mathbf{R} = \mathbf{S} \setminus \{v\}$

$$\exists \mathbf{R}.\ \mathsf{Tree}(\texttt{o}, \mathbf{R}) \wedge \mathsf{Compose}(\varnothing, v, \mathbf{R}, \mathbf{S}) \wedge \mathbf{R} = \mathbf{S} \setminus \{v\}$$

Substitution. Discard $\mathsf{Compose}(\varnothing, v, \mathbf{R}, \mathbf{S})$ and $\mathbf{R} = \mathbf{S} \setminus \{v\}$.

$$\mathsf{Tree}(\texttt{o}, \mathbf{S} \setminus \{v\})$$

```
}
else {
```

Deny if-condition.

$$\exists v, l, r, \mathbf{L}, \mathbf{R}.\ \texttt{root} \mapsto v,l,r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \wedge \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge l \neq \texttt{null}$$

Lemma: .

$$\exists v, l, r, \mathbf{L}, \mathbf{R}.\ \texttt{root} \mapsto v,l,r \ast \mathsf{NonEmptyTree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \wedge \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$$

```
  if (root.c[1] == null) {
```

This branch mostly symmetrical to the previous…

```
    o = root.c[0];
    delete root;
```

$$\mathsf{Tree}(\texttt{o}, \mathbf{S} \setminus \{v\})$$

```
  }
  else {
```

Deny if-condition. Use lemma: .

$$\exists v, l, r, \mathbf{L}, \mathbf{R}.\ \texttt{root} \mapsto v,l,r \ast \mathsf{NonEmptyTree}(l, \mathbf{L}) \ast \mathsf{NonEmptyTree}(r, \mathbf{R}) \wedge \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$$

```
    RemoveMaxRet r = removeMax(root.c[0]); auto nv = r.max; auto newLeft = r.root;
```

Specification of `removeMax`.

$$\exists v, l, r, \mathbf{L}, \mathbf{R}.\ \texttt{root} \mapsto v,l,r \ast \mathsf{Tree}(\texttt{newLeft}, \mathbf{L} \setminus \{\texttt{nv}\}) \ast \mathsf{NonEmptyTree}(r, \mathbf{R}) \wedge \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(\texttt{nv}, \mathbf{L})$$

```
    root.value = max; root.c[0] = newLeft;
```

Assignment.

$$\exists v, r, \mathbf{L}, \mathbf{R}.\ \texttt{root} \mapsto \texttt{nv},\texttt{newLeft},r \ast \mathsf{Tree}(\texttt{newLeft}, \mathbf{L} \setminus \{\texttt{nv}\}) \ast \mathsf{NonEmptyTree}(r, \mathbf{R}) \wedge \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge \mathsf{Max}(\texttt{nv}, \mathbf{L})$$

$\exists$I on `newLeft` as l.

$$\exists v, l, r, \mathbf{L}, \mathbf{R}.\ \texttt{root} \mapsto \texttt{nv},l,r \ast \mathsf{Tree}(l, \mathbf{L} \setminus \{\texttt{nv}\}) \ast \mathsf{NonEmptyTree}(r, \mathbf{R}) \wedge$$

$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathsf{Max}(\mathsf{nv}, \mathbf{L})$

Weakening lemma: .

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \mathtt{root} \mapsto \mathsf{nv}, l, r * \mathsf{Tree}(l, \mathbf{L} \setminus \{\mathsf{nv}\}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathsf{Max}(\mathsf{nv}, \mathbf{L})$

Lemma: $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathsf{Max}(\mathsf{nv}, \mathbf{L}) \Rightarrow \mathsf{Compose}(\mathbf{L} \setminus \{\mathsf{nv}\}, \mathsf{nv}, \mathbf{R}, \mathbf{S} \setminus \{v\})$.
Discard $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$ and $\mathsf{Max}(\mathsf{nv}, \mathbf{L})$.

$\exists l, r, \mathbf{L}, \mathbf{R}.$
$\quad \mathtt{root} \mapsto \mathsf{nv}, l, r * \mathsf{Tree}(l, \mathbf{L} \setminus \{\mathsf{nv}\}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L} \setminus \{\mathsf{nv}\}, \mathsf{nv}, \mathbf{R}, \mathbf{S} \setminus \{v\})$

$\exists$I on $\mathbf{L} \setminus \{\mathsf{nv}\}$ as $\mathbf{L}$.

$\exists l, r, \mathbf{L}, \mathbf{R}.$
$\quad \mathtt{root} \mapsto \mathsf{nv}, l, r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, \mathsf{nv}, \mathbf{R}, \mathbf{S} \setminus \{v\})$

Close $\mathsf{TopOfTree}(\mathtt{root}, \mathsf{nv}, \mathbf{S} \setminus \{v\})$.

$\mathsf{TopOfTree}(\mathtt{root}, \mathsf{nv}, \mathbf{S} \setminus \{v\})$

$\exists$I on nv as v.

$\exists v.\ \mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S} \setminus \{v\})$

Close $\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S} \setminus \{v\})$.

$\mathsf{NonEmptyTree}(\mathtt{root}, \mathbf{S} \setminus \{v\})$

Weakening

$\mathsf{Tree}(\mathtt{root}, \mathbf{S} \setminus \{v\})$

```
      o = root;
```

Assignment.

$\mathsf{Tree}(\mathtt{o}, \mathbf{S} \setminus \{v\})$

```
    }
```

If-rule.

$\mathsf{Tree}(\mathtt{o}, \mathbf{S} \setminus \{v\})$

```
  }
```

If-rule. Postcondition.

$\mathsf{Tree}(\mathtt{o}, \mathbf{S} \setminus \{v\})$

```
  return root;
}
```

Here's `remove`:

```
module bst.remove.recursive;

import bst.node;
import bst.remove.removeRoot;
```

```
Node* remove(Node* root, int value) {
  Node* o;
```

Function precondition.

Tree(root, **S**)

```
  if (root == null) {
```

Assert if-condition. Lemma: .

EmptyTree(root, **S**)

```
    o = root;
```

Tree(o, **S** \ {value})

```
  }
  else {
```

Deny if-condition. Lemma: .

NonEmptyTree(root, **S**)

Open NonEmptyTree(root, **S**).

$\exists v.$ TopOfTree(root, v, **S**)

Open TopOfTree(root, v, **S**).

$\exists v, l, r,$ **L**, **R**.
root $\mapsto v,l,r \ast$ Tree(l, **L**) $\ast$ Tree(r, **R**) $\wedge$
Compose(**L**, v, **R**, **S**)

```
    if (value == root.value) {
```

Assert if-condition. Substitution.

$\exists l, r,$ **L**, **R**.
root $\mapsto$ value,$l,r \ast$ Tree(l, **L**) $\ast$ Tree(r, **R**) $\wedge$
Compose(**L**, value, **R**, **S**)

Close TopOfTree(root, value, **S**).

TopOfTree(root, value, **S**).

```
      o = removeRoot(root);
```

Specification for `removeRoot`.

Tree(o, **S** \ {value})

```
    }
    else {
```

Deny if-condition.

$\exists v, l, r,$ **L**, **R**.
root $\mapsto v,l,r \ast$ Tree(l, **L**) $\ast$ Tree(r, **R**) $\wedge$
Compose(**L**, v, **R**, **S**) $\wedge\ v \neq$ value

```
      if (value < root.value) {
```

Assert if-condition.

$\exists v, l, r,$ **L**, **R**.
root $\mapsto v,l,r \ast$ Tree(l, **L**) $\ast$ Tree(r, **R**) $\wedge$
Compose(**L**, v, **R**, **S**) $\wedge$ value $< v$

```
        o.c[0] = remove(o.c[0], value);
```

Specification for `remove`. Assignment. $\exists$**l**.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L} \setminus \{\texttt{value}\}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \texttt{value} < v$

Lemma: $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \texttt{value} < v \Rightarrow \mathsf{Compose}(\mathbf{L} \setminus \{\texttt{value}\}, v, \mathbf{R}, \mathbf{S} \setminus \{\texttt{value}\})$.
Discard $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$ and $\texttt{value} < v$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L} \setminus \{\texttt{value}\}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L} \setminus \{\texttt{value}\}, v, \mathbf{R}, \mathbf{S} \setminus \{\texttt{value}\})$

$\exists$I on $\mathbf{L} \setminus \{\texttt{value}\}$ as $\mathbf{L}$.

$\exists v, l, r, \mathbf{L}, \mathbf{R}.$
$\quad \texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
$\quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S} \setminus \{\texttt{value}\})$

Close $\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S} \setminus \{\texttt{value}\})$.

$\exists v.\ \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S} \setminus \{\texttt{value}\})$

Close $\mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{value}\})$.

$\mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{value}\})$

Weakening

$\mathsf{Tree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{value}\})$

```
    }
    else {
```

Symmetrical…

```
      o.c[1] = remove(o.c[1], value);
```

$\mathsf{Tree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{value}\})$

```
    }
```

If-rule.

$\mathsf{Tree}(\texttt{root}, \mathbf{S} \setminus \{\texttt{value}\})$

```
    o = root;
```

Assignment.

$\mathsf{Tree}(\texttt{o}, \mathbf{S} \setminus \{\texttt{value}\})$

```
  }
```

If-rule.

$\mathsf{Tree}(\texttt{o}, \mathbf{S} \setminus \{\texttt{value}\})$

```
 }
```

If-rule. Function postcondition.

$\mathsf{Tree}(\texttt{o}, \mathbf{S} \setminus \{\texttt{value}\})$

```
 return o;
}
```