Verifying 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 `Node`s 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*…*a*+*l*. The addresses occupied by the `Node`s 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 `Node`s, 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 `Node`s 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 `Node`s 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 $\varnothing$ $\{\mathbf{77}\}$ $\{\mathbf{30}, \mathbf{77}\}$ $\{\mathbf{29}, \mathbf{30}, \mathbf{77}\}$ $\{\mathbf{2}, \mathbf{29}, \mathbf{30}, \mathbf{77}\}$

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** $\{\boldsymbol{value}\} \cup \mathbf{T}$

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 $\varnothing$, 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, List(`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*, $\mathbf{T}$) NonEmptyList(*head*, $\mathbf{S}$),where $\mathbf{S} = \{v\} \cup \mathbf{T}$,and $\forall\ t \in \mathbf{T}.\ v < t.$ *head* EmptyList(*head*, $\mathbf{S}$),where $\mathbf{S}$

$= \varnothing$. List(*head*, $\mathbf{S}$)

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:

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

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

$$\begin{array}{lll} \mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \overset{\text{def}}{=} & \mathtt{head} = \mathtt{null} \wedge & \\ & \mathbf{S} = \varnothing \wedge & \mathbf{S} \text{ is the empty set and} \\ & \mathbf{emp}. & \text{there is no allocated memory.} \end{array}$$

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 $\mathsf{NonEmptyList}$.

$$\begin{array}{ll} \mathsf{Compose}(value, \mathbf{T}, \mathbf{S}) \overset{\text{def}}{=} & \{value\} \cup \mathbf{T} = \mathbf{S} \wedge \\ & \forall t \in \mathbf{T}.\ value < t. \end{array}$$

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 Nodes, 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 $\mathsf{NonEmptyList}$:

$$\begin{array}{ll} \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \overset{\text{def}}{=} & \exists value, tail, \mathbf{T}. \\ & \mathsf{Compose}(value, \mathbf{T}, \mathbf{S}) \wedge \\ & \mathtt{head} \mapsto value, tail \ast \mathsf{List}(tail, \mathbf{T}). \end{array}$$

Lemmata used in LL algorithms

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

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

$$\begin{array}{lll} \mathbf{1.} & \mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) & \text{given} \\ \mathbf{2.} & \mathsf{EmptyList}(\mathtt{head}, \mathbf{S}) \vee \mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) & \mathbf{1}, \vee\mathrm{I} \\ \mathbf{3.} & \mathsf{List}(\mathtt{head}, \mathbf{S}) & \mathbf{2}, \text{close predicate} \end{array}$$

$\text{NonEmptyList}(\texttt{head}, S) \Rightarrow \text{List}(\texttt{head}, S)$

(Serves the same use as for $\text{EmptyList}(\texttt{head}, S) \Rightarrow \text{List}(\texttt{head}, S)$).

1.	$\text{NonEmptyList}(\texttt{head}, S)$	given
2.	$\text{EmptyList}(\texttt{head}, S) \lor \text{NonEmptyList}(\texttt{head}, S)$	**1**, $\lor$I
3.	$\text{List}(\texttt{head}, S)$	**2**, close predicate

$\text{EmptyList}(\ \ , S) \Rightarrow \texttt{value} \notin S$

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

1.	$\text{EmptyList}(\texttt{head}, S)$	given
2.	$\texttt{head} = \texttt{null} \land S = \varnothing \land \textbf{emp}$	**1**, open predicate
3.	$S = \varnothing$	**2**, $\land$E
4.	$\texttt{value} \notin \varnothing$	Nothing in empty set
5.	$\texttt{value} \notin S$	**3**, **4**, equality

$\text{List}(\texttt{head}, S) \land \texttt{head} = \texttt{null} \Rightarrow \text{EmptyList}(\texttt{head}, S)$

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.	$\text{List}(\texttt{head}, S) \land \texttt{head} = \texttt{null}$	given
2.	$\text{List}(\texttt{head}, S)$	**1**, $\land$E
3.	$\texttt{head} = \texttt{null}$	**1**, $\land$E
4.	$\text{EmptyList}(\texttt{head}, S) \lor \text{NonEmptyList}(\texttt{head}, S)$	**2**, open predicate
5.	$\quad\text{NonEmptyList}(\texttt{head}, S)$	assume
6.	$\quad\exists v, t, T.\ \text{Compose}(v, T, S) \land$ $\quad\texttt{head} \mapsto v, t * \text{List}(t, T)$	**5**, open predicate
7.	$\quad\exists v, t.\ \texttt{head} \mapsto v, t$	**6**, $\land$E, frame off $\text{List}(t, T)$
8.	$\quad\texttt{head} \neq \texttt{null}$	**7**, pointer non-null
9.	$\neg\text{NonEmptyList}(\texttt{head}, S)$	**5**, , **8**, **RAA**
10.	$\text{EmptyList}(\texttt{head}, S)$	**4**, **9**, $\lor$E

$\text{List}(\texttt{head}, S) \land \texttt{head} \neq \texttt{null} \Rightarrow \text{NonEmptyList}(\texttt{head}, S)$

(Symmetrical to the converse lemma.)

$\text{Compose}(v, T, S) \Rightarrow v \in S$

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

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

2.	$\{v\} \cup \mathbf{T} = \mathbf{S} \land \forall t \in \mathbf{T}.\ value < t$	1, open predicate
3.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	2, ∧E
4.	$\{v\} \subseteq \mathbf{S}$	??
5.	$v \in \{v\}$	Element in singleton set
6.	$v \in \mathbf{S}$	4, 5, member of subset is member of set

$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land v \neq \texttt{value} \Rightarrow \texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S}$

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.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land v \neq \texttt{value}$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	1, ∧E
3.	$v \neq \texttt{value}$	1, ∧E
4.	$\{\texttt{value}\} \cup \mathbf{T} = \mathbf{S} \land \forall t \in \mathbf{T}.\ \texttt{value} < t$	2, open predicate
5.	$\{\texttt{value}\} \cup \mathbf{T} = \mathbf{S}$	4, ∧E
6.	$\forall t \in \mathbf{T}.\ \texttt{value} < t$	4, ∧E
7.	$\mathbf{T} \subseteq \mathbf{S}$	5
8.	$\quad \texttt{value} \in \mathbf{T}$	assume
9.	$\quad \texttt{value} \in \mathbf{S}$	7, 8, member of subset is member of set
10.	$\texttt{value} \in \mathbf{T} \rightarrow \texttt{value} \in \mathbf{S}$	8, , →I
11.	$\texttt{value} \notin \{v\}$	3
12.	$\quad \texttt{value} \in \mathbf{S}$	assume
13.	$\quad \texttt{value} \in \{v\} \lor \texttt{value} \in \mathbf{T}$	12, 5, ??
14.	$\quad \texttt{value} \in \mathbf{T}$	13, 11,
15.	$\texttt{value} \in \mathbf{S} \rightarrow \texttt{value} \in \mathbf{T}$	12, 14, →I
16.	$\texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S}$	10, 15, ↔I

$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land \texttt{value} < v \Rightarrow \texttt{value} \notin \mathbf{S}$

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

10.	$v < \texttt{value}$	**8**
11.	$\texttt{value} \notin \mathbf{S}$	**9**, **6**, **10**, RAA

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

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

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land v < \texttt{value}$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	**1**, $\land$E
3.	$v < \texttt{value}$	**1**, $\land$E
4.	$\{v\} \cup \mathbf{T} = \mathbf{S} \land \forall t \in \mathbf{T}.\ v < t$	**2**, open predicate
5.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	**4**, $\land$E
6.	$\forall t \in \mathbf{T}.\ v < t$	**4**, $\land$E
7.	$\{v\} \cup \mathbf{T} \cup \{\texttt{value}\} = \mathbf{S} \cup \{\texttt{value}\}$	**5**, $a = b \Rightarrow \mathrm{f}(a) = \mathrm{f}(b)$
8.	$\forall t \in \{\texttt{value}\}.\ v < t$	**3**, **??**
9.	$\forall t \in \mathbf{T} \cup \{\texttt{value}\}.\ v < t$	**6**, **8**, **??**
10.	$\mathsf{Compose}(v, \mathbf{T} \cup \{\texttt{value}\}, \mathbf{S} \cup \{\texttt{value}\})$	**7**, **9**, close predicate

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

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

1.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land w < v$	given
2.	$\mathsf{Compose}(v, \mathbf{T}, \mathbf{S})$	**1**, $\land$E
3.	$w < v$	**1**, $\land$E
4.	$\{v\} \cup \mathbf{T} = \mathbf{S} \land \forall t \in \mathbf{T}.\ v < t$	**2**, open predicate
5.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	**4**, $\land$E
6.	$\forall t \in \mathbf{T}.\ v < t$	**4**, $\land$E
7.	$\forall t \in \mathbf{T}.\ w < t$	**6**, **3**, transitivity of < relation
8.	$\{w\} \cup \mathbf{S} = \mathbf{S} \cup \{w\}$	Commutativity of set union
9.	$\forall t \in \{v\}.\ w < t$	**3**, **??**
10.	$\forall t \in (\{v\} \cup \mathbf{T}).\ w < t$	**7**, **9**, **??**
11.	$\forall t \in \mathbf{S}.\ w < t$	**10**, **5**, substitution
12.	$\mathsf{Compose}(w, \mathbf{S}, \mathbf{S} \cup \{w\})$	**8**, **11**, 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.	$\{v\} \cup \mathbf{T} = \mathbf{S} \land \forall t \in \mathbf{T}.\ v < t$	**1**, open predicate

3.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	2, $\wedge$E
4.	$\forall t \in \mathbf{T}.\ v < t$	2, $\wedge$E
5.	$\quad v \in \mathbf{T}$	assume
6.	$\quad v < v$	4, 5
7.	$v \notin \mathbf{T}$	5, 6, RAA
8.	$\{v\} \cap \mathbf{T} = \varnothing$	7
9.	$\mathbf{T} = \mathbf{S} \setminus \{v\}$	3, 8

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

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

1.	$\text{Compose}(v, \mathbf{T}, \mathbf{S}) \wedge v \neq w$	given
2.	$\text{Compose}(v, \mathbf{T}, \mathbf{S})$	1, $\wedge$E
3.	$v \neq w$	1, $\wedge$E
4.	$\{v\} \cup \mathbf{T} = \mathbf{S} \wedge \forall t \in \mathbf{T}.\ v < t$	2, open predicate
5.	$\{v\} \cup \mathbf{T} = \mathbf{S}$	4, $\wedge$E
6.	$\forall t \in \mathbf{T}.\ v < t$	4, $\wedge$E
7.	$(\{v\} \cup \mathbf{T}) \setminus \{w\} = \mathbf{S} \setminus \{w\}$	5, equal operations
8.	$(\{v\} \setminus \{w\}) \cup (\mathbf{T} \setminus \{w\}) = \mathbf{S} \setminus \{w\}$	7, distributivity of set minus over set union
9.	$\{v\} \setminus \{w\} = \{v\}$	3, ??
10.	$\{v\} \cup (\mathbf{T} \setminus \{w\}) = (\mathbf{S} \setminus \{w\})$	8, 9, substitution
11.	$\forall t \in (\mathbf{T} \setminus \{w\}).\ v < t$	6, for all in set then for all in subset
12.	$\{v\} \cup (\mathbf{T} \setminus \{w\}) = (\mathbf{S} \setminus \{w\}) \wedge \forall t \in (\mathbf{T} \setminus \{w\}).\ v < t$	10, 11, $\wedge$I
13.	$\text{Compose}(v, \mathbf{T} \setminus \{w\}, \mathbf{S} \setminus \{w\})$	12, 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: $\text{List}(\texttt{head}, \mathbf{S})$.

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* ∈ S; *i.e.*, o ↔ *value* ∈ 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) ∧ o ↔ value ∈ 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.
    List(head, S) ∧ head = null
    Lemma: List(head, S) ∧ head = null ⇒ EmptyList(head, S).
```

```
      EmptyList(head, S)

      Lemma: EmptyList(head, S) ⇒ value ∉ S.
      EmptyList(head, S) ∧ value ∉ S

      Weakening lemma: EmptyList(head, S) ⇒ 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
}
else {

      Deny if-condition.
      List(head, S) ∧ head ≠ null

      Lemma: List(head, S) ∧ head ≠ null ⇒ NonEmptyList(head, S).
      NonEmptyList(head, S)

      Open NonEmptyList(head, S).
      ∃v, tail, T.
          Compose(v, T, S) ∧
          head ↦ v, tail ∗ List(tail, T)
    if (head.value == value) {

        Assert if-condition: substitute value for v.
        ∃tail, T.
            Compose(value, T, S) ∧
            head ↦ value, tail ∗ List(tail, T)

        Compose(value, T, S) ⇒ value ∈ S.
        ∃tail, T.
            Compose(value, T, S) ∧
            head ↦ value, tail ∗ List(tail, T) ∧
            value ∈ S

        Close NonEmptyList(head, S).
        NonEmptyList(head, S) ∧ value ∈ S

        Weakening lemma: NonEmptyList(head, S) ⇒ List(head, S).
        List(head, S) ∧ value ∈ S
      o = true;

        Assignment.
        List(head, S) ∧ value ∈ S ∧ o = true

        true ↔ true.
        List(head, S) ∧ o ↔ value ∈ S
```

```
}
else {
    Deny if-condition.
    $\exists v, tail, \mathbf{T}.$
        $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
        $\texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \land$
        $v \neq \texttt{value}$
  if (head.value < value) {
      Assert if-condition.
      $\exists v, tail, \mathbf{T}.$
          $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
          $\texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \land$
          $v < \texttt{value}$
      Lemma: $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land v < \texttt{value} \Rightarrow \texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S}$.
      $\exists v, tail, \mathbf{T}.$
          $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
          $\texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \land$
          $v < \texttt{value} \land$
          $\texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S}$
    o = search(head.tail, value);
      Inductive use of specification. Note $|\mathbf{T}| < |\mathbf{S}|$.
      $\exists v, tail, \mathbf{T}.$
          $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
          $\texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \land$
          $v < \texttt{value} \land$
          $\texttt{value} \in \mathbf{T} \leftrightarrow \texttt{value} \in \mathbf{S} \land$
          $\texttt{o} \leftrightarrow \texttt{value} \in \mathbf{T}$
      Transitivity of double implication. Discard unrequired assertions.
      $\exists v, tail, \mathbf{T}.$
          $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
          $\texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \land$
          $\texttt{o} \leftrightarrow \texttt{value} \in \mathbf{S}$
      Close $\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S})$.
      $\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S}) \land \texttt{o} \leftrightarrow \texttt{value} \in \mathbf{S}$
      Weakening: $\mathsf{NonEmptyList}(\texttt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\texttt{head}, \mathbf{S})$.
      $\mathsf{List}(\texttt{head}, \mathbf{S}) \land \texttt{o} \leftrightarrow \texttt{value} \in \mathbf{S}$
  }
  else {
      Deny if-condition. Use $\neg(v < \texttt{value}) \Rightarrow \texttt{value} \leq v$.
      $\exists v, tail, \mathbf{T}.$
          $\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$
          $\texttt{head} \mapsto v, tail \ast \mathsf{List}(tail, \mathbf{T}) \land$
          $\texttt{value} \neq v \land \texttt{value} \leq v$
      $(a \neq b \land a \leq b) \Rightarrow b < a$.
```

```
              ∃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 $\mathbf{S}$ such that it represents the set $\mathbf{S} \cup \{$`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:

$$\{\,\mathsf{List}(\mathtt{head}, \mathbf{S})\,\}\ \{\,\mathtt{nhead\ =\ insert(head,\ value);}\,\}\ \{\,\mathsf{NonEmptyList}(\mathtt{nhead}, \mathbf{S} \cup \{\mathtt{value}\})\,\}$$

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` ∈ $\mathbf{S}$, so ∅ ∪ {`value`} = $\mathbf{S}$. The `head` variable already points into a set representing $\mathbf{S}$, so we just return `head`. If v > `value`, then `value` ∉ $\mathbf{S}$, and we can construct a valid LL representing $\mathbf{S}$ ∪ {`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 $\mathbf{T}$ ∪ {`value`}; by replacing the *tail* field of the first node with this new pointer, we obtain a list representing {v} ∪ $\mathbf{T}$ ∪ {`value`}, which is equal to $\mathbf{S}$ ∪ {`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.
   List(head, S)
  if (head == null) {
     Assert if-condition.
     List(head, S) ∧ head = null
```

Lemma: List(head, **S**) ∧ head = null ⇒ EmptyList(head, **S**)

EmptyList(head, **S**)

Open EmptyList(head, **S**). Discard head = null.

S = ∅ ∧ **emp**

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

Specification for new Node(value).

S = ∅ ∧ **emp** ∗ NonEmptyList(o, {value})

emp ∗ X = X.

S = ∅ ∧ NonEmptyList(o, {value})

X = ∅ ∪ **X**.

S = ∅ ∧ NonEmptyList(o, ∅ ∪ {value})

Substitution. Discard **S** = ∅.

NonEmptyList(o, **S** ∪ {value})

```
}
else {
```

Deny if-condition.

List(head, **S**) ∧ head ≠ null

Lemma: List(head, **S**) ∧ head ≠ null ⇒ NonEmptyList(head, **S**).

NonEmptyList(head, **S**)

Open NonEmptyList(head, **S**).

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

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

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

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

Compose(value, **T**, **S**) ⇒ value ∈ **S**.

∃*tail*, **T**.
 Compose(value, **T**, **S**) ∧
 head ↦ value, *tail* ∗ List(*tail*, **T**) ∧
 value ∈ **S**

Close NonEmptyList(head, **S**).

NonEmptyList(head, **S**) ∧ value ∈ **S**

a ∈ **B** ⇒ {*a*} ⊆ **B**

NonEmptyList(head, **S**) ∧ {value} ⊆ **S**

A ⊆ **B** ⇒ **B** ∪ **A** = **B**

```
    NonEmptyList(head, S) ∧ {value} ∪ S = S

    Substitution. Discard {value} ∪ S = S.

    NonEmptyList(head, {value} ∪ S)
  o = head;

    Assignment.

    NonEmptyList(o, {value} ∪ S)
}
else {

    Deny if-condition.

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

      Assert if-condition.

      ∃v, tail, T.
            Compose(v, T, S) ∧
            head ↦ v, tail ∗ List(tail, T) ∧
            v < value
    Node* ntail = insert(head.tail, value);

      Use specification for insert.

      ∃v, tail, T.
            Compose(v, T, S) ∧
            head ↦ v, tail ∗ List(ntail, T ∪ {value}) ∧
            v < value
    head.tail = ntail;

      Assignment. Reintroduce existential quantification.

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

      Lemma: Compose(v, T, S) ∧ v < value ⇒ Compose(v, T ∪ {value}, S ∪ {value})

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

      Introduce existential quantification on T ∪ {value}.

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

      Close NonEmptyList(head, S ∪ {value}).

      NonEmptyList(head, S ∪ {value})
    o = head;

      Assignment.
```

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

```
  }
  else {
```

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

$$\begin{aligned}&\exists v, \mathit{tail}, \mathbf{T}.\\ &\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land\\ &\quad \texttt{head} \mapsto v, \mathit{tail} \ast \mathsf{List}(\mathit{tail}, \mathbf{T}) \land\\ &\quad \texttt{value} \neq v \land \texttt{value} \leq v\end{aligned}$$

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

$$\begin{aligned}&\exists v, \mathit{tail}, \mathbf{T}.\\ &\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land\\ &\quad \texttt{head} \mapsto v, \mathit{tail} \ast \mathsf{List}(\mathit{tail}, \mathbf{T}) \land\\ &\quad \texttt{value} < v\end{aligned}$$

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

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

$$\begin{aligned}&\exists v, \mathit{tail}, \mathbf{T}.\\ &\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land\\ &\quad \texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \texttt{head} \mapsto v, \mathit{tail} \ast \mathsf{List}(\mathit{tail}, \mathbf{T}) \land\\ &\quad \texttt{value} < v\end{aligned}$$

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

$$\begin{aligned}&\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land \texttt{value} < v \land\\ &\texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \mathsf{NonEmptyList}(\texttt{head}, \mathbf{S})\end{aligned}$$

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

$$\begin{aligned}&\mathsf{Compose}(\texttt{value}, \mathbf{S}, \mathbf{S} \cup \{\texttt{value}\}) \land\\ &\texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \mathsf{NonEmptyList}(\texttt{head}, \mathbf{S})\end{aligned}$$

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

$$\begin{aligned}&\mathsf{Compose}(\texttt{value}, \mathbf{S}, \mathbf{S} \cup \{\texttt{value}\}) \land\\ &\texttt{nhead} \mapsto \texttt{value}, \texttt{head} \ast \mathsf{List}(\texttt{head}, \mathbf{S})\end{aligned}$$

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

$$\begin{aligned}&\exists v, \mathit{tail}, \mathbf{T}.\\ &\quad \mathsf{Compose}(v, \mathbf{T}, \mathbf{S} \cup \{\texttt{value}\}) \land\\ &\quad \texttt{nhead} \mapsto v, \mathit{tail} \ast \mathsf{List}(\mathit{tail}, \mathbf{T})\end{aligned}$$

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.

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

```
}
```

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 **S** ∖ {*value*}. Return *head*. *v*< *value* *head* *ntail* List(*tail*, **T**) remove , *value* List(*ntail*, **T** ∖

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

$\{value\} = \mathbf{T}$, 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, $\mathbf{S}$) $\Rightarrow$ $\mathbf{S} = \emptyset$

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

$\emptyset \setminus \mathbf{X} = \emptyset$, $\mathbf{S} = \emptyset$, 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, $\mathbf{T}$, $\mathbf{S}$) $\wedge$
head $\mapsto v, tail \ast$ List($tail$, $\mathbf{T}$)

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

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

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

$\quad\mathsf{Compose}(\texttt{value}, \mathbf{T}, \mathbf{S}) \land$

$\quad\texttt{head} \mapsto \texttt{value}, \mathit{tail} * \mathsf{List}(\mathit{tail}, \mathbf{T})$

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

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

$\quad\texttt{head} \mapsto \texttt{value}, \mathit{tail} * \mathsf{List}(\mathit{tail}, \mathbf{T}) \land$

$\quad\mathbf{T} = \mathbf{S} \setminus \{\texttt{value}\}$

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

$\exists \mathit{tail}.$

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

```
    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.

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

$\quad\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$

$\quad\texttt{head} \mapsto v, \mathit{tail} * \mathsf{List}(\mathit{tail}, \mathbf{T}) \land$

$\quad v \neq \texttt{value}$

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

Assert if-condition.

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

$\quad\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$

$\quad\texttt{head} \mapsto v, \mathit{tail} * \mathsf{List}(\mathit{tail}, \mathbf{T}) \land$

$\quad v < \texttt{value}$

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

Use specification for `remove`.

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

$\quad\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$

$\quad\texttt{head} \mapsto v, \mathit{tail} * \mathsf{List}(\texttt{ntail}, \mathbf{T} \setminus \{\texttt{value}\}) \land$

$\quad v < \texttt{value}$

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

Assignment.

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

$\quad\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$

$\quad\texttt{head} \mapsto v, \texttt{ntail} * \mathsf{List}(\texttt{ntail}, \mathbf{T} \setminus \{\texttt{value}\}) \land$

$\quad v < \texttt{value}$

$\exists$I on `ntail` as *tail*.

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

$\quad\mathsf{Compose}(v, \mathbf{T}, \mathbf{S}) \land$

$\quad\texttt{head} \mapsto v, \mathit{tail} * \mathsf{List}(\mathit{tail}, \mathbf{T} \setminus \{\texttt{value}\}) \land$

```
        $v < \mathtt{value}$

        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 $v < \mathtt{value}$.

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

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

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

        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 $\neg(v < \mathtt{value}) \Rightarrow \mathtt{value} \leq v$.

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

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

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

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

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

        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 $\mathbf{S} \setminus \{\mathtt{value}\} = \mathbf{S}$.

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

        Weakening lemma: $\mathsf{NonEmptyList}(\mathtt{head}, \mathbf{S}) \Rightarrow \mathsf{List}(\mathtt{head}, \mathbf{S})$.
        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 **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 **L** and **R**, held respectively in what are called the `left` and `right` subtrees.

In our LL, we chose to order the elements of the set **S** in strictly ascending order; that is, the element v was chosen as the minimal element of **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 **S** is arbitrary. The set **L** is then chosen to hold all elements of **S** that are strictly less than v, and the set **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.

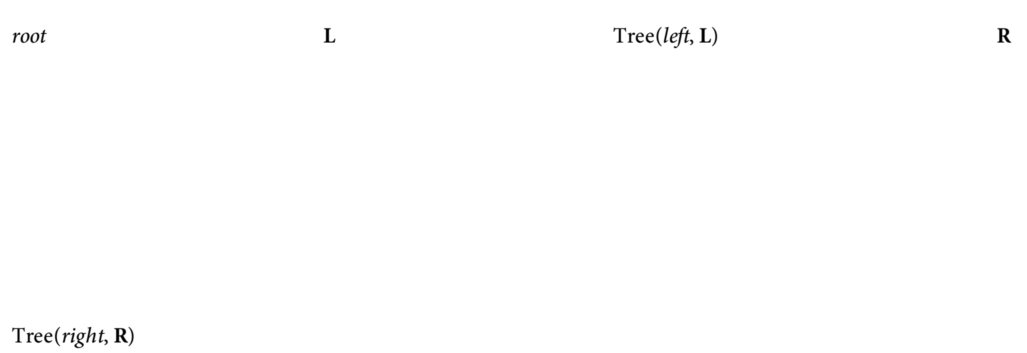
As with the LL, a pointer into the `root` node is required to complete the representation of the set. Also as with the LL, ∅ has a distinct representation: a pointer to `null` and no allocated heap chunks.

We described a general pointer `head` into a list representing the set **S** with the predicate List(`head`, **S**). Similarly, we will now describe a general pointer `root` into a BST representing **S** with a predicate Tree(`root`, **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*, **S**),where **S** = ∅. Tree(*root*, **S**) NonEmptyTree(*root*, **S**),where **S** = **L** ∪ {*v*} ∪ **R**,∀ *l*∈**L**. *l* < *v*, and∀

r∈**R**. *v* < *r*. *left* *v* *right*



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

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

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

$\mathsf{EmptyTree}(n, S) \overset{\text{def}}{=}$ $n = \texttt{null} \wedge$ 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**:

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

$\forall l \in \mathbf{L}.\ l < v \land$ all values in $\mathbf{L}$ are less than v, and

$\forall r \in \mathbf{R}.\ v < r.$ all values in $\mathbf{R}$ are greater than v.

Using our $\mathsf{Compose}$ 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, $\mathsf{TopOfTree}$:

$\mathsf{TopOfTree}(\mathtt{root}, v, \mathbf{S}) \overset{\text{def}}{=} \exists l, r, \mathbf{L}, \mathbf{R}.$

$\mathtt{root} \mapsto v, l, r$ `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

$\land \ \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}).$ the values in the tree are totally ordered.

The $\mathsf{TopOfTree}$ 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:

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

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

Here are the boring ones:

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

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

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

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

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

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

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

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

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

Recursive BST algorithms

Recursive `search`

The purpose of the `search` algorithm is identical to that in the LL, and its specification has an identical structure:

$\{$ Tree(root, $\mathbf{S}$) $\}$ `{o = search(root, value);}` $\{$ Tree(o, $\mathbf{S}$) $\wedge$ o $\leftrightarrow$ value $\in \mathbf{S}$ $\}$

image/svg+xml *head* points to a non-empty tree with *v* at theroot 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 searching for *value*. **S** *head* Tree(*head*, **S**) The tree at *head* is empty, and represents theset ∅. *value* ∉ ∅. Return false. **S** = ∅ *head* = null EmptyTree(*head*, **S**) **Is *head* null?** *head* = null *head* ≠ null *value* ∈ {*v*}, so *value* ∈ **L** ∪ {v} ∪ **R**, so*value* ∈ *S*. Return true. NonEmptyTree(*head*, **S** = **L** ∪ {*value*} ∪ **R**) *left value right head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) *value* ∉ {*v*}, and *value* ∉ **R** ∵ ∀*r*∈**R**. *v*<*r*, so(*value* ∈ **S**) ↔ (*value* ∈ **L**). Search *left.* 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 value* ∉ {*v*}, and *value* ∉ **L** ∵ ∀*l*∈**L**. *v*>*l*, so(*value* ∈ **S**) ↔ (*value* ∈ **R**). Search *right.* NonEmptyTree(*head*, **S** = **L** ∪ {*v*} ∪ **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.

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, $\mathbf{S}$)

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

```
    $\mathrm{EmptyTree}(\texttt{root}, S) \land S = \emptyset$
    $S = \emptyset \Rightarrow \texttt{value} \notin S$
    $\mathrm{EmptyTree}(\texttt{root}, S) \land \texttt{value} \notin S$
    o = false;
    Assignment.
    $\mathrm{EmptyTree}(\texttt{root}, S) \land \texttt{value} \notin S \land \texttt{o} = \texttt{false}$
    ?
    $\mathrm{EmptyTree}(\texttt{root}, S) \land \texttt{o} \leftrightarrow (\texttt{value} \in S)$
    Weakening lemma: $\mathrm{EmptyTree}(\texttt{root}, S) \Rightarrow \mathrm{Tree}(\texttt{root}, S)$
    $\mathrm{Tree}(\texttt{root}, S) \land \texttt{o} \leftrightarrow (\texttt{value} \in S)$
}
else {
    Deny if-condition.
    $\mathrm{Tree}(\texttt{root}, S) \land \texttt{root} \neq \texttt{null}$
    Lemma: $\mathrm{Tree}(\texttt{root}, S) \land \texttt{root} \neq \texttt{null} \Rightarrow \mathrm{NonEmptyTree}(\texttt{root}, S)$
    $\mathrm{NonEmptyTree}(\texttt{root}, S)$
    bool eq = rootEq(root, value);
    Specification for rootEq.
    $\exists v.\ \mathrm{TopOfTree}(\texttt{root}, v, S) \land \texttt{eq} \leftrightarrow v = \texttt{value}$
    if (eq) {
        Assert if-test.
        $\exists v.\ \mathrm{TopOfTree}(\texttt{root}, v, S) \land \texttt{eq} \leftrightarrow v = \texttt{value} \land \texttt{eq}$
        Use $\texttt{eq} \leftrightarrow v = \texttt{value}$. Discard eq.
        $\exists v.\ \mathrm{TopOfTree}(\texttt{root}, v, S) \land v = \texttt{value}$
        Substitution.
        $\mathrm{TopOfTree}(\texttt{root}, \texttt{value}, S)$
        Lemma: $\mathrm{TopOfTree}(\texttt{root}, \texttt{value}, S) \Rightarrow \texttt{value} \in S$
        $\mathrm{TopOfTree}(\texttt{root}, \texttt{value}, S) \land \texttt{value} \in S$
        Reintroduce ∃I on value as $v$.
        $\exists v.\ \mathrm{TopOfTree}(\texttt{root}, v, S) \land \texttt{value} \in S$
        Close $\mathrm{Tree}(\texttt{root}, S)$.
        $\mathrm{Tree}(\texttt{root}, S) \land \texttt{value} \in S$
        o = true;
        Assignment.
        $\mathrm{Tree}(\texttt{root}, S) \land \texttt{value} \in S \land \texttt{o} = \texttt{true}$
        ?
        $\mathrm{Tree}(\texttt{root}, S) \land \texttt{o} \leftrightarrow \texttt{value} \in S$
```

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

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

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

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

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

Assert if-condition.

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

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 $\texttt{value} < v$.

$$\begin{aligned}&\exists v, l, r, \mathbf{L}, \mathbf{R}.\\ &\qquad \texttt{root} \mapsto v, l, r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \land\\ &\qquad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land\\ &\qquad \texttt{value} \in \mathbf{L} \leftrightarrow \texttt{value} \in \mathbf{S}\end{aligned}$$

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

Use specification for search.

$$\begin{aligned}&\exists v, l, r, \mathbf{L}, \mathbf{R}.\\ &\qquad \texttt{root} \mapsto v, l, r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \land\\ &\qquad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land\\ &\qquad \texttt{value} \in \mathbf{L} \leftrightarrow \texttt{value} \in \mathbf{S} \land\\ &\qquad \texttt{o} \leftrightarrow (\texttt{value} \in \mathbf{L})\end{aligned}$$

Transitivity of double implication.

$$\begin{aligned}&\exists v, l, r, \mathbf{L}, \mathbf{R}.\\ &\qquad \texttt{root} \mapsto v, l, r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \land\\ &\qquad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land\\ &\qquad \texttt{o} \leftrightarrow (\texttt{value} \in \mathbf{S})\end{aligned}$$

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

$$\begin{aligned}&\exists v.\\ &\qquad \mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S}) \land \texttt{o} \leftrightarrow (\texttt{value} \in \mathbf{S})\end{aligned}$$

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;
}
```

<u>Recursive `insert`</u>

The specification for `insert` follows the same structure as for the LL:

{ Tree(root, S) } {`o = insert(root, value);`} { NonEmptyTree(o, S ∪ {value}) }

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 some set **S**. We are inserting *value*. **S**

head Tree(*head*, **S**) The tree at *head* is empty, and represents the set ∅. 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,

because that 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, because that 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*} ∪

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

NonEmptyTree(*nright*,**R** ∪ {*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, S)
  if (root == null) {
      Assert if-condition.
      Tree(root, S) ∧ root = null
      Lemma: Tree(root, S) ∧ root = null ⇒ EmptyTree(root, S)
      EmptyTree(root, S)
      Lemma: EmptyTree(root, S) ⇒ S = ∅
      EmptyTree(root, S) ∧ S = ∅
    o = new Node(value);
      NonEmptyTree(root, S ∪ {value})
  }
  else {
      Deny if-condition.
      Tree(root, S) ∧ root ≠ null
      Lemma: Tree(root, S) ∧ root ≠ null ⇒ NonEmptyTree(root, S)
      NonEmptyTree(root, S)
    bool eq = rootEq(root, value);
      Specification for rootEq.
      ∃v. TopOfTree(root, v, S) ∧ eq ↔ v = value
    if (eq) {
        Assert if-test.
        ∃v. TopOfTree(root, v, S) ∧ eq ↔ v = value ∧ eq
        Use eq ↔ v = value. Discard eq.
        ∃v. TopOfTree(root, v, S) ∧ v = value

        Substitution.
        TopOfTree(root, value, S)
        o = root;
        NonEmptyTree(root, S ∪ {value})
      }
      else {
        Deny if-condition.
        ∃v. TopOfTree(root, v, S) ∧ eq ↔ v = value ∧ ¬eq

        Use eq → (v = value). Discard ¬eq.
        ∃v. TopOfTree(root, v, S) ∧ v ≠ value

        Open TopOfTree(root, v, S).
        ∃v, l,r,L,R.
             root ↦ v,l,r ∗ Tree(l, L) ∗ Tree(r, R) ∧
             Compose(L, v, R, S) ∧
             v ≠ value
        if (value < root.value) {
          Assert if-condition.
          ∃v, l,r,L,R.
               root ↦ v,l,r ∗ Tree(l, L) ∗ Tree(r, R) ∧
               Compose(L, v, R, S) ∧
               value < v
          Node* left = root.c[0];
          ∃E of l as left.
          ∃v, r,L,R.
               root ↦ v,left,r ∗ Tree(left, L) ∗ Tree(r, R) ∧
               Compose(L, v, R, S) ∧
               value < v
          Node* nleft = insert(left, value);
          Specification for insert.
          ∃v, r,L,R.
               root ↦ v,left,r ∗ Tree(nleft, L ∪ {value}) ∗ Tree(r, R) ∧
               Compose(L, v, R, S) ∧
               value < v
          root.c[0] = nleft;
          Assignment.
          ∃v, r,L,R.
               root ↦ v,nleft, r ∗ Tree(nleft, L ∪ {value}) ∗ Tree(r, R) ∧
               Compose(L, v, R, S) ∧
               value < v

          ∃I on nleft as l.
          ∃v, l, r, L, R.
               root ↦ v,l, r ∗ Tree(l, L ∪ {value}) ∗ Tree(r, R) ∧
```

```
        Compose(L, v, R, S) ∧
        value < v

        Lemma: Compose(L, v, R, S) ∧ value < v ⇒ Compose(L ∪ {value}, v, R, S ∪ {value}).
        Discard Compose(L, v, R, S) and value < v.

        ∃v, l, r, L, R.
            root ↦ v,l, r ∗ Tree(l, L ∪ {value}) ∗ Tree(r, R) ∧
            Compose(L ∪ {value}, v, R, S ∪ {value})

        ∃I on L ∪ {value} as L.

        ∃v, l, r, L, R.
            root ↦ v,l, r ∗ Tree(l, L) ∗ Tree(r, R) ∧
            Compose(L, v, R, S ∪ {value})

        Close NonEmptyTree(root, S ∪ {value}).

        NonEmptyTree(root, S ∪ {value})
      }
      else {
        (Symmetrical case…)
        root.c[1] = insert(root.c[1], value);
        NonEmptyTree(root, S ∪ {value})
      }
      If-rule.

      NonEmptyTree(root, S ∪ {value})
      o = root;
      Assignment.

      NonEmptyTree(o, S ∪ {value})
    }
    If-rule.

    NonEmptyTree(o, S ∪ {value})
  }
  If-rule.

  NonEmptyTree(o, S ∪ {value})
  return o;
}
```

Recursive BST `remove` algorithms

Recursive BST `removeMax`

The LL algorithm for `remove` could simply return the tail of the list if and when it found the element to remove. With the BST, it is not that simple: once we have found the value to remove, we need to somehow merge the values in the left and right subtrees into a single tree. The standard way of doing this is to remove an element from one of the subtrees (say, the left subtree), replace the root value with that element, then return the root. In order to maintain the order of the elements in the tree, we must pick the maximum element from the left subtree, or the minimum element from the right subtree. I (arbitrarily) choose the former.

Before we give the specification for `removeMax`, we must define what it means to be a 'maximum' element. We can wrap this up in a predicate:

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

The `removeMax` procedure then has the following specification:

$$\{\, \mathsf{NonEmptyTree}(\texttt{root}, \mathbf{S}) \,\}\ \{\, \texttt{(o, max) = removeMax(root);} \,\}\ \{\, \mathsf{Tree}(\texttt{o}, \mathbf{S} \setminus \{\texttt{max}\}) \land \mathsf{Max}(\texttt{max}, \mathbf{S}) \,\}$$

`removeMax` is implemented recursively. As it is passed a NonEmptyTree, it does not have to check for a `null` root pointer. It checks whether the right subtree is empty. If it is, then the value at the root is the maximum in the set. Otherwise, the maximum in the set is the maximum in the right-hand set, and so we recursively call `removeMax` on the right subtree.

The `removeMax` annotations rely on the following lemmata:

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

This is used by `removeMax` to identify the maximum element: if the right subtree is empty, then the root holds the maximum element in the set.

image/svg+xml v **L** **R** **L** ∪ {v} ∪ **R** **S**

$-\infty$ ∞ $\{v\}$

1.	$\mathsf{Compose}(\mathbf{L}, v, \varnothing, \mathbf{S})$	given
2.	$\mathbf{L} \cup \{v\} \cup \varnothing = \mathbf{S} \land \forall l \in \mathbf{L}.\ l < v \land \forall r \in \varnothing.\ v < r$	**1**, open predicate
3.	$\mathbf{L} \cup \{v\} \cup \varnothing = \mathbf{S}$	**2**, ∧E
4.	$\forall l \in \mathbf{L}.\ l < v$	**2**, ∧E
5.	$\{v\} \subseteq \mathbf{S}$	**3**
6.	$v \in \{v\}$	
7.	$v \in \mathbf{S}$	**6**, **5**, member of subset member of set
8.	$v = v$	
9.	$v \leq v$	**8**, weakening
10.	$\forall s \in \{v\}.\ s \leq v$	**9**, ??
11.	$\forall s \in \mathbf{L}.\ s \leq v$	**4**, weakening
12.	$\forall s \in (\mathbf{L} \cup \{v\}).\ s \leq v$	**10**, **11**, $\forall x \in \mathbf{A}.\ \mathsf{P}(x) \land \forall x \in \mathbf{B}.\ \mathsf{P}(x) \Rightarrow \forall x \in (\mathbf{A} \cup \mathbf{B}).\ \mathsf{P}(x)$
13.	$\mathbf{L} \cup \{v\} = \mathbf{S}$	**3**, $\mathbf{A} \cup \varnothing = \mathbf{A}$
14.	$\forall s \in \mathbf{S}.\ s \leq v$	**12**, **13**, substitution

15. $v \in \mathbf{S} \land \forall s \in \mathbf{S}.\, s \leq v$ **7**, **14**, $\land$I

16. $\mathsf{Max}(v, \mathbf{S})$ **15**, close predicate

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

This lemma is used in the `removeMax` function to demonstrate that removing the maximum element from the right subtree will give us the maximum element of the whole set.

$\mathbf{L}$ $\quad$ $\mathbf{R} \setminus \{r\}$ $\quad$ $\mathbf{L} \cup \{v\} \cup (\mathbf{R} \setminus \{r\})$ $\quad$ $\mathbf{S}$

$\mathbf{S} \setminus \{r\}$

1.	$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathsf{Max}(r, \mathbf{R})$	given
2.	$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$	**1**, $\land$E
3.	$\mathsf{Max}(r, \mathbf{R})$	**1**, $\land$E
4.	$\mathbf{L} \cup \{v\} \cup \mathbf{R} = \mathbf{S} \land$ $\forall l \in \mathbf{L}.\, l < v \land$ $\forall r \in \mathbf{R}.\, v < r$	**2**, open predicate
5.	$\mathbf{L} \cup \{v\} \cup \mathbf{R} = \mathbf{S}$	**4**, $\land$E
6.	$\forall l \in \mathbf{L}.\, l < v$	**4**, $\land$E
7.	$\forall r \in \mathbf{R}.\, v < r$	**4**, $\land$E
8.	$r \in \mathbf{R} \land \forall x \in \mathbf{R}.\, x \leq r$	**3**, open predicate
9.	$r \in \mathbf{R}$	**8**, $\land$E
10.	$\forall x \in \mathbf{R}.\, x \leq r$	**8**, $\land$E
11.	$(\mathbf{L} \cup \{v\} \cup \mathbf{R}) \setminus \{r\} = \mathbf{S} \setminus \{r\}$	**5**, $a = b \Rightarrow f(a) = f(b)$
12.	$(\mathbf{L} \setminus \{r\}) \cup (\{v\} \setminus \{r\}) \cup (\mathbf{R} \setminus \{r\}) = \mathbf{S} \setminus \{r\}$	**11**, $(\mathbf{A} \cup \mathbf{B}) \setminus \mathbf{C} = (\mathbf{A} \setminus \mathbf{C}) \cup (\mathbf{B} \setminus \mathbf{C})$

13.	$v < r$	7, 9
14.	$\forall l \in \mathbf{L}.\ l < r$	6, 13, transitivity
15.	$r \notin \mathbf{L}$	14, $\neg(r < r)$
16.	$\{r\} \not\subseteq \mathbf{L}$	15
17.	$\mathbf{L} \setminus \{r\} = \mathbf{L}$	16
18.	$r \neq v$	13, weakening
19.	$\{r\} \cap \{v\} = \varnothing$	18
20.	$\{v\} \setminus \{r\} = \{v\}$	19
21.	$\mathbf{L} \cup \{v\} \cup (\mathbf{R} \setminus \{r\}) = \mathbf{S} \setminus \{r\}$	12, 17, 20, substitution
22.	$\forall r \in (\mathbf{R} \setminus \{r\}).\ v < r$	7, true of set members, true of subset members
23.	$\mathbf{L} \cup \{v\} \cup (\mathbf{R} \setminus \{r\}) = (\mathbf{S} \setminus \{r\}) \wedge$ $\forall l \in \mathbf{L}.\ l < v \wedge$ $\forall r \in (\mathbf{R} \setminus \{r\}).\ v < r$	21, 6, 22, $\wedge$I
24.	Compose($\mathbf{L}$, v, $\mathbf{R} \setminus \{r\}$, $\mathbf{S} \setminus \{r\}$)	23, close predicate
25.	$\mathbf{R} \subseteq \mathbf{S}$	5
26.	$r \in \mathbf{S}$	9, 25, member of subset member of set
27.	$\forall s \in \mathbf{L}.\ s \leq r$	14, weakening
28.	$v \leq r$	13, weakening
29.	$\forall s \in \{v\}.\ s \leq r$	28
30.	$\forall s \in (\mathbf{L} \cup \{v\} \cup \mathbf{R}).\ s \leq r$	27, 29, 10, true of members of subsets, true of members of union
31.	$\forall s \in \mathbf{S}.\ s \leq r$	30, 5, substitution
32.	$r \in \mathbf{S} \wedge \forall s \in \mathbf{S}.\ s \leq r$	26, 31, $\wedge$I
33.	Max(r, $\mathbf{S}$)	32, close predicate
34.	Compose($\mathbf{L}$, v, $\mathbf{R} \setminus \{r\}$, $\mathbf{S} \setminus \{r\}$) $\wedge$ Max(r, $\mathbf{S}$)	24, 33, $\wedge$I

image/svg+xml The tree represents $\mathbf{L} \cup \{v\} \cup \mathbf{R} \setminus \{max\}$, which= $\mathbf{S} \setminus \{max\}$ because the sets are disjoint. NonEmptyTree(*root*, $\mathbf{S} \setminus \{max\}$)

l v nr root **L** Tree(*l*, **L**) removeMax Tree(*nr*, $\mathbf{R} \setminus \{max\}$) **R** We have retrieved *v*, the maximum value in **S**,and subtracted it from **S** to give **L** =

$\mathbf{S} \setminus \{v\}$. *root max* = *v* **L** Tree(*newRoot*, $\mathbf{L} = \mathbf{S} \setminus \{v\}$) *newRoot* = *l* **Precondition:** *root* points to a non-empty tree.We will remove the maximum

value in **S**. NonEmptyTree(*root*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *l v r root* **L** Tree(*l*, **L**) **R** Tree(*r*, **R**) *v* is the maximum value in **S**. $\mathbf{L} = \mathbf{S} \setminus \{v\}$.

NonEmptyTree(*root*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \varnothing$) *l v r root* **L** Tree(*l*, **L**) EmptyTree(*r*, $\varnothing$) ***r* = null?** *r* = null *r* ≠ null **(*nr*, *max*) = removeMax(*r*);*root.r* =**

***nr*; return (*root*, *max*); max = *v*; newRoot = *l*; delete *root*;return (*newRoot*, *max*);** *r* points to a NonEmptyTree. The maximumvalue of **S** is in $\mathbf{R}$. NonEmptyTree($root$, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) l v r $root$ $\mathbf{L}$ Tree(l, $\mathbf{L}$) $\mathbf{R}$ NonEmptyTree(r, $\mathbf{R}$)

The code is simple:

```
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];
```

$\exists$E on r as $\texttt{r}$.

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

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

Assert if-condition. Substitution.

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

Specification for `removeMax`.

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

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

Assignment.

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

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

∃I on `rightRoot` as r.

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

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

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

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

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

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

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

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

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

Weaken

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

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

Assignment.

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

```
  }
```

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;
}
```

Recursive BST `removeRoot`

The `removeRoot` procedure uses `removeMax` to merge two subtrees. It is passed the tree with the two subtrees that must be merged, and returns a tree representing the union of the sets represented by those subtrees.

$$\{\,\mathsf{TopOfTree}(\texttt{root}, v, \mathbf{S})\,\}\ \{\texttt{o = removeRoot(root);}\}\ \{\,\mathsf{Tree}(\texttt{o}, \mathbf{S} \setminus \{v\})\,\}$$

The following lemma is used by `removeRoot`:

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

This is used by the `removeRoot` function to show that we can remove the value at the root by replacing it with the removed maximum value from the left subtree.

image/svg+xml *v* **L** **R** *nv* **L** ∪ {*v*} ∪ **R**

$\mathbf{L} \setminus \{nv\}$ $(\mathbf{L} \setminus \{nv\}) \cup \{nv\} \cup \mathbf{R} = \mathbf{S} \setminus \{nv\}$ $\mathbf{S}$ $\mathbf{R}$

-∞ ∞ $\{v\}$ $\{nv\}$

1.	$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathsf{Max}(nv, \mathbf{L})$	given
2.	$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$	**1**, $\land$E
3.	$\mathsf{Max}(nv, \mathbf{L})$	**1**, $\land$E
4.	$\mathbf{L} \cup \{v\} \cup \mathbf{R} = \mathbf{S} \land$ $\forall l \in \mathbf{L}.\, l < v \land$ $\forall r \in \mathbf{R}.\, v < r$	**2**, open predicate
5.	$\mathbf{L} \cup \{v\} \cup \mathbf{R} = \mathbf{S}$	**4**, $\land$E
6.	$\forall l \in \mathbf{L}.\, l < v$	**4**, $\land$E
7.	$\forall r \in \mathbf{R}.\, v < r$	**4**, $\land$E
8.	$(\mathbf{L} \cup \{v\} \cup \mathbf{R}) \setminus \{v\} = \mathbf{S} \setminus \{v\}$	**5**, $a = b \Rightarrow f(a) = f(b)$
9.	$(\mathbf{L} \setminus \{v\}) \cup (\{v\} \setminus \{v\}) \cup (\mathbf{R} \setminus \{v\}) = \mathbf{S} \setminus \{v\}$	**8**, $(\mathbf{A} \cup \mathbf{B}) \setminus \mathbf{C} = (\mathbf{A} \setminus \mathbf{C}) \cup (\mathbf{B} \setminus \mathbf{C})$
10.	$(\mathbf{L} \setminus \{v\}) \cup (\mathbf{R} \setminus \{v\}) = \mathbf{S} \setminus \{v\}$	**9**, $\mathbf{A} \setminus \mathbf{A} = \varnothing$
11.	$\mathbf{L} \setminus \{v\} = \mathbf{L}$	??
12.	$\mathbf{R} \setminus \{v\} = \mathbf{R}$	??
13.	$\mathbf{L} \cup \mathbf{R} = \mathbf{S} \setminus \{v\}$	**10**, **11**, **12**, substitution
14.	$(\mathbf{L} \setminus \{nv\}) \cup \{nv\} = \mathbf{L}$	??

15.	$(\mathbf{L} \setminus \{nv\}) \cup \{nv\} \cup \mathbf{R} = (\mathbf{S} \setminus \{v\})$	**13, 14,** substitution
16.	$\forall l \in (\mathbf{L} \setminus \{nv\}).\ l < nv$	???
17.	$\forall r \in \mathbf{R}.\ nv < r$	???
18.	$(\mathbf{L} \setminus \{nv\}) \cup \{nv\} \cup \mathbf{R} = (\mathbf{S} \setminus \{v\}) \land$ $\forall l \in (\mathbf{L} \setminus \{nv\}).\ l < nv \land$ $\forall r \in \mathbf{R}.\ nv < r$	**15, 16, 17,** ∧I
19.	$\mathsf{Compose}(\mathbf{L} \setminus \{nv\}, nv, \mathbf{R}, \mathbf{S} \setminus \{v\})$	??

image/svg+xml **Precondition:** *root* points to a non-empty treewith *v* at the root. We will return $\mathbf{S} \setminus \{v\}$. TopOfTree(*root*, *v*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *l v r root* **L** Tree(*l*, **L**) **R** Tree(*r*, **R**) ***l* = null?** $l \neq$ null $l =$ null ***newRoot* = *root.r*; delete root; return *newRoot*;** The set **L** is empty, so $\mathbf{R} = \mathbf{S} \setminus \{v\}$. TopOfTree(*root*, *v*, $\mathbf{S} = \varnothing \cup \{v\} \cup \mathbf{R}$) *l v r root* EmptyTree(*l*, ∅) **R** Tree(*r*, **R**) Tree(*newRoot*, $\mathbf{S} \setminus \{v\}$) satisfies the functionpostcondition. *root* Tree(*newRoot*, $\mathbf{R} = \mathbf{S} \setminus \{v\}$) *newRoot* = *r* **R** The set **L** is not empty. We cannot simplyreturn the right subtree. TopOfTree(*root*, *v*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *l v r root* **L** NonEmptyTree(*l*, **L**) **R** Tree(*r*, **R**) ***r* = null?** $r \neq$ null $r =$ null The set **R** is empty, so $\mathbf{L} = \mathbf{S} \setminus \{v\}$. TopOfTree(*root*, *v*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *l v r root* **L** NonEmptyTree(*l*, **L**) EmptyTree(*r*, ∅) NonEmptyTree(*newRoot*, $\mathbf{L} = \mathbf{S} \setminus \{v\}$) impliesTree(*newRoot*, $\mathbf{S} \setminus \{v\}$). Postcondition. *root* NonEmptyTree(*newRoot*, $\mathbf{L} = \mathbf{S} \setminus \{v\}$) *newRoot* = *l* **L** ***newRoot* = *root.l*; delete root; return *newRoot*;** Both subtrees are non-empty, so we cannot justreturn *l* or *r*. TopOfTree(*root*, *v*, $\mathbf{S} = \mathbf{L} \cup \{v\} \cup \mathbf{R}$) *l v r root* **L** NonEmptyTree(*l*, **L**) **R** NonEmptyTree(*r*, **R**) Tree represents $\mathbf{L} \setminus \{max\} \cup \{max\} \cup \mathbf{R}$, whichis $\mathbf{L} \cup \mathbf{R}$, which is $\mathbf{S} \setminus \{v\}$. Postcondition. TopOfTree(*root*, *max*, $\mathbf{S} = \mathbf{L} \setminus \{max\} \cup \{max\} \cup \mathbf{R}$) *nl max r root* **L** Tree(*nl*, $\mathbf{L} \setminus \{max\}$) **R** NonEmptyTree(*r*, **R**) removeMax **(*nl*, *max*) = removeMax(*l*); *root.v* = *max*; *root.l* = *nl*;**

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.

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

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

$$\begin{array}{l}\exists v, l, r, \mathbf{L}, \mathbf{R}.\\ \quad \mathtt{root} \mapsto v,l,r \ast \mathsf{Tree}(l, \mathbf{L}) \ast \mathsf{Tree}(r, \mathbf{R}) \land\\ \quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})\end{array}$$

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

Assert if-condition. Substitution.

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

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

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

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

$$\begin{array}{l}\exists v, l, r, \mathbf{L}, \mathbf{R}.\\ \quad \mathtt{root} \mapsto v,\mathtt{null},r \ast \mathbf{emp} \ast \mathsf{Tree}(r, \mathbf{R}) \land\\ \quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathbf{L} = \emptyset\end{array}$$

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

$$\begin{array}{l}\exists v, r, \mathbf{L}, \mathbf{R}.\\ \quad \mathtt{root} \mapsto v,\mathtt{null},r \ast \mathsf{Tree}(r, \mathbf{R}) \land\\ \quad \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \mathbf{L} = \emptyset\end{array}$$

Substitution.

$$\begin{array}{l}\exists v, r, \mathbf{R}.\\ \quad \mathtt{root} \mapsto v,\mathtt{null},r \ast \mathsf{Tree}(r, \mathbf{R}) \land\\ \quad \mathsf{Compose}(\emptyset, v, \mathbf{R}, \mathbf{S})\end{array}$$

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

Assignment.

$$\begin{array}{l}\exists v, \mathbf{R}.\\ \quad \mathtt{root} \mapsto v,\mathtt{null},\mathtt{o} \ast \mathsf{Tree}(\mathtt{o}, \mathbf{R}) \land\\ \quad \mathsf{Compose}(\emptyset, v, \mathbf{R}, \mathbf{S})\end{array}$$

```
    delete root;
```

Free heap chunk.

$$\exists \mathbf{R}.\ \mathsf{Tree}(\mathtt{o}, \mathbf{R}) \land \mathsf{Compose}(\emptyset, v, \mathbf{R}, \mathbf{S})$$

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

$$\exists \mathbf{R}.\ \mathsf{Tree}(\mathtt{o}, \mathbf{R}) \land \mathsf{Compose}(\emptyset, v, \mathbf{R}, \mathbf{S}) \land \mathbf{R} = \mathbf{S} \setminus \{v\}$$

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

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

```
}
else {
```

Deny if-condition.

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

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

Lemma: .

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

```
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: .

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

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

Specification of `removeMax`.

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

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

Assignment.

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

∃I on `newLeft` as l.

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

Weakening lemma: .

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

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

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

∃I on $\mathbf{L} \setminus \{\texttt{nv}\}$ as $\mathbf{L}$.

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

```
          Close TopOfTree(root, nv, S \ {v}).
          TopOfTree(root, nv, S \ {v})

          ∃I on nv as v.
          ∃v. TopOfTree(root, v, S \ {v})

          Close NonEmptyTree(root, S \ {v}).
          NonEmptyTree(root, S \ {v})

          Weakening
          Tree(root, S \ {v})
        o = root;
          Assignment.
          Tree(o, S \ {v})
      }
      If-rule.
      Tree(o, S \ {v})
    }
    If-rule. Postcondition.
    Tree(o, S \ {v})
    return root;
}
```

Recursive *BST* `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 *value* at the root. Wehave a helper function for that: removeRoot. NonEmptyTree(*head*, **S** = **L** ∪ {*value*} ∪ **R**) *left value right head* **L** Tree(*left*, **L**) **R** Tree(*right*, **R**) {*v*} and **R** do not contain *value*, and so we haveremoved from them. We now remove from **L**. 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* (symmetrical) 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** remove , Tree(*nleft*, **L** \ {*value*}) *value*

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

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

Tree(*nhead*, **L** ∪ **R**) ***nhead*** = **removeRoot(*head*). Return *nhead*.**

$\underline{\text{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land w < v \Rightarrow \text{Compose}(\mathbf{L} \setminus \{w\}, v, \mathbf{R}, \mathbf{S} \setminus \{w\})}$

image/svg+xml *v* **L** **R** *w* **L** ∪ {*v*} ∪ **R**

$\mathbf{L} \setminus \{w\}$ $(\mathbf{L} \setminus \{w\}) \cup \{v\} \cup \mathbf{R} = \mathbf{S} \setminus \{w\}$ $\mathbf{S}$ $\mathbf{R}$

$-\infty$ ∞ $\{v\}$

1.	$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \wedge w < v$	given
2.	$\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S})$	**1**, $\wedge$E
3.	$w < v$	**1**, $\wedge$E
4.	$\mathbf{L} \cup \{v\} \cup \mathbf{R} = \mathbf{S} \wedge$ $\forall l \in \mathbf{L}.\, l < v \wedge$ $\forall r \in \mathbf{R}.\, v < r$	**2**, open predicate
5.	$\mathbf{L} \cup \{v\} \cup \mathbf{R} = \mathbf{S}$	**4**, $\wedge$E
6.	$\forall l \in \mathbf{L}.\, l < v$	**4**, $\wedge$E
7.	$\forall r \in \mathbf{R}.\, v < r$	**4**, $\wedge$E
8.	$(\mathbf{L} \cup \{v\} \cup \mathbf{R}) \setminus \{w\} = \mathbf{S} \setminus \{w\}$	**5**, $a = b \Rightarrow f(a) = f(b)$
9.	$(\mathbf{L} \setminus \{w\}) \cup (\{v\} \setminus \{w\}) \cup (\mathbf{R} \setminus \{w\}) = \mathbf{S} \setminus \{w\}$	**8**, $(\mathbf{A} \cup \mathbf{B}) \setminus \mathbf{C} = (\mathbf{A} \setminus \mathbf{C}) \cup (\mathbf{B} \setminus \mathbf{C})$
10.	$\mathbf{R} \setminus \{w\} = \mathbf{R}$	??
11.	$\{v\} \setminus \{w\} = \{v\}$	??
12.	$(\mathbf{L} \setminus \{w\}) \cup \{v\} \cup \mathbf{R} = \mathbf{S} \setminus \{w\}$	**9**, **10**, **11**, substitution
13.	$\forall l \in (\mathbf{L} \setminus \{w\}).\, l < v$	**6**, $\forall x \in \mathbf{A}.\, \mathsf{P}(x) \wedge \mathbf{B} \subseteq \mathbf{A} \Rightarrow \forall x \in \mathbf{B}.\, \mathsf{P}(x)$
14.	$(\mathbf{L} \setminus \{w\}) \cup \{v\} \cup \mathbf{R} = \mathbf{S} \setminus \{w\} \wedge$ $\forall l \in \mathbf{L} \setminus \{w\}.\, l < v \wedge$	**12**, **13**, **7**, $\wedge$I

$\forall r \in \mathbf{R}.\ v < r$

15. $\mathsf{Compose}(\mathbf{L} \setminus \{w\}, v, \mathbf{R}, \mathbf{S} \setminus \{w\})$ **14**, close predicate

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).
     ∃v. TopOfTree(root, v, S)
     Open TopOfTree(root, v, S).
     ∃v, l, r, L, R.
         root ↦ v,l,r ∗ Tree(l, L) ∗ Tree(r, R) ∧
         Compose(L, v, R, S)
    if (value == root.value) {
       Assert if-condition. Substitution.
       ∃l, r, L, R.
           root ↦ value,l,r ∗ Tree(l, L) ∗ Tree(r, R) ∧
           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.
    $\exists v, l, r, \mathbf{L}, \mathbf{R}.$
        $\texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
        $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land v \neq \texttt{value}$
    if (value < root.value) {
        Assert if-condition.
        $\exists v, l, r, \mathbf{L}, \mathbf{R}.$
            $\texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
            $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \texttt{value} < v$
        o.c[0] = remove(o.c[0], value);
        Specification for remove. Assignment. $\exists$I.
        $\exists v, l, r, \mathbf{L}, \mathbf{R}.$
            $\texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L} \setminus \{\texttt{value}\}) * \mathsf{Tree}(r, \mathbf{R}) \land$
            $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land \texttt{value} < v$
        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 $\texttt{value} < v$.
        $\exists v, l, r, \mathbf{L}, \mathbf{R}.$
            $\texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L} \setminus \{\texttt{value}\}) * \mathsf{Tree}(r, \mathbf{R}) \land$
            $\mathsf{Compose}(\mathbf{L} \setminus \{\texttt{value}\}, v, \mathbf{R}, \mathbf{S} \setminus \{\texttt{value}\})$
        $\exists$I on $\mathbf{L} \setminus \{\texttt{value}\}$ as $\mathbf{L}$.
        $\exists v, l, r, \mathbf{L}, \mathbf{R}.$
            $\texttt{root} \mapsto v,l,r * \mathsf{Tree}(l, \mathbf{L}) * \mathsf{Tree}(r, \mathbf{R}) \land$
            $\mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S} \setminus \{\texttt{value}\})$
        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.
```

```
      Tree(o, S \{value})
  }
    If-rule. Function postcondition.
    Tree(o, S \{value})
  return o;
}
```

Verifying the RBT

As before, I verify recursive algorithms for `search`, `insert`, and `remove`.

Introduction, motivation, history

The ideal BST is one that enables each comparison step in the algorithms to cut the size of the tree under consideration in two—that is, where the size of the left subtree is equal to the size of the right subtree. Indeed, this principle was the motivating factor in using a BST instead of an LL. However, the algorithms on the BST provide no guarantee that this will be the case. For example, insertion the values $1 \ldots n$ into an empty BST will result in a tree with the same topology as a LL. What is needed is some guarantee of *balance*, and this is what the RBT provides.

The RBT data structure

The RBT is an augmentation of the BST, and as such it is also a recursive data structure: a specific RBT contains smaller RBTs (subject to restrictions that will be explained). As before, the recursive algorithms on the RBT follow the recursive definition of the data structure, and many of the proofs we used for the BST (such as the lemmata relating to the Compose predicate) can be reused here.

The specific augmentation that the RBT makes is to modify the `Node` definition such that each node in the structure carries a single extra bit of information. Typically, and for simplicity, this bit is carried as a single extra boolean field in the `Node`. This bit is called the node's `color`, and the two values the bit may take on are named not `true` and `false`, but rather ‘red’ and ‘black’ (the primary reason for which is convenience in visual representations of the data structure). In the code, we will assign `false` to play the role of red and `true` to black. Here is the `Node` structure for the RBT:

```
module rb.node;

struct Node {
  int value;
  Node*[2] c;
  bool black;

  this(int value) {
    this.value = value;
    this.c[0] = this.c[1] = null;
    this.black = false;
  }
}


bool red(Node * node) {
  return node != null && node.black == false;
}
```

The distribution of red bits and black bits over an RBT is not arbitrary. In explaining the restrictions, it is helpful to think of a tree as being either 'red-rooted' or 'black-rooted', where this indicates the color stored in the `color` field of the root node. We shall develop separate predicates for each: RT shall describe a red-rooted RBT, and BT a black-rooted one.

A red-rooted tree, RT, is subject to the restriction that neither of its child subtrees are red-rooted. This is equivalent to saying that if a node is red, then its parent is black; and is again equivalent to saying that there shall be no two red nodes 'in a row' in any proper path. These equivalent rules are known as the *red rule*. I prefer the first formulation of it, as it more readily translates to our inductive definition of an RBT.

There is no such rule for black-rooted trees: their child subtrees are unrestricted in the color of their root. Black nodes are subject to a different, more complex, restriction: that *from any node in the tree* (whether red or black), *all proper paths from that node to a leaf contain the same number of black nodes*. This rule and equivalent statements of it are called the *black rule*.

To encode this rule in our predicates, let us call the number of black nodes on the path from some node the *black-height* of that node. In terms of our inductive definition, we say that the black-height is an attribute of the tree, and as such our predicates are parameterized by it: BT(`root`, **S**, *h*) describes a black-rooted RBT, at root address `root`, representing set **S**, with a black-height of *h*. In exactly the same way, RT(`root`, **S**, *h*) describes a *red-rooted* RBT, at root address `root`, representing set **S**, with a black-height of *h*.

Consider a black-rooted tree at black-height *h*—what is the black-height of one of its subtrees? Since all paths to the leaves contain the same number of black nodes, just consider a single path to a single leaf. This path runs through one of the child nodes of the root. Since the root node is one of the *h* black nodes, the other *h*−1 black nodes must be in the section of the path descending from the child node to the leaf. As this was for an arbitrary path through an arbitrary child node, both subtrees are at black-height *h*−1. Now consider a red-rooted tree at black-height *h*—what is the black-height of its subtrees? Since the root is red, it never contributes to the *h* black nodes on any path, and so all *h* black nodes are in the black-height of the subtree; *i.e.*, the subtrees of a red-rooted tree at height *h* are also at height *h*.

Finally, the RBT has a separate representation of ∅, and once again, it is a pointer to `null` with no allocated memory. Despite having no root `Node` with a `color` field, the empty tree is interpreted as being 'black-rooted', with a black-height of 1.

As before, the RBT's recursive structure can be shown graphically:

image/svg+xml *root* EBT(*root*, **S**, *h*),where **S** = ∅and h = 0. BT(*root*, **S**, *h*) NBT(*root*, **S**, *h*),where Compose(**L**, *v*, **R**, **S**),and *h* > 0.

l *v* *r* *c* *root*

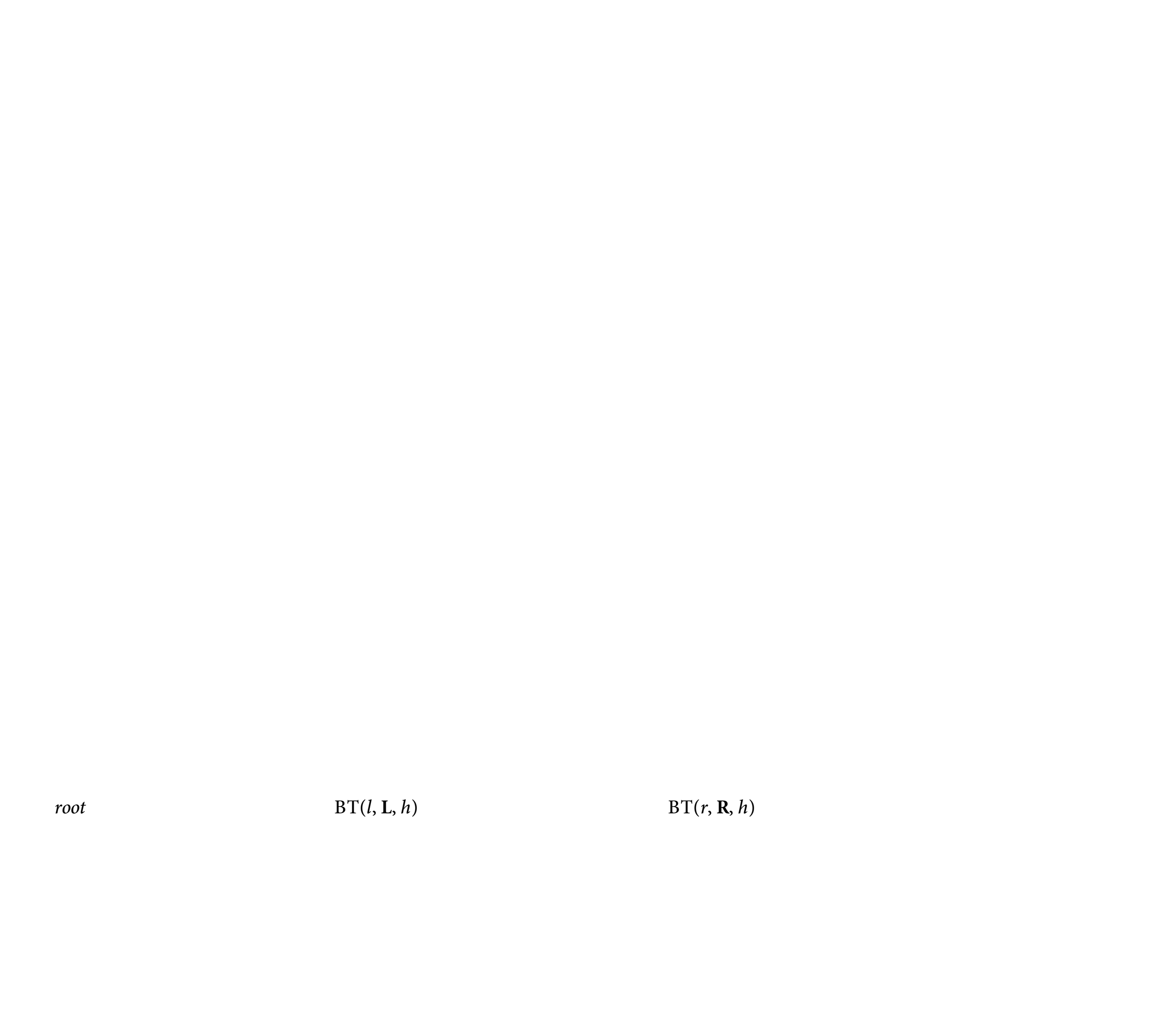
L RBT(l, **L**, h-1) **R** RBT(r, **R**, h-1)

RT(*root*, **S**, *h*),where Compose(**L,** *v,* **R, S**). *l* *v*

r	*c*	*root*	**L**	BT(*l*,

$\mathbf{L}, h)$ $\mathbf{R}$ $\mathrm{BT}(r, \mathbf{R}, h)$ $\mathrm{RBT}(\mathit{root}, \mathbf{S}, h)$

RT($root$, $\mathbf{S}$, h) l v r



We can now translate our visual representations into separation logic. An RBT is either red or black:

$$\mathsf{RBT}(\texttt{root}, \mathbf{S}, h) \overset{\text{def}}{=} \mathsf{RT}(\texttt{root}, \mathbf{S}, h) \lor \mathsf{BT}(\texttt{root}, \mathbf{S}, h)$$

The RT is easier to describe, as there is only one type. A red-rooted tree pointed to by `root` represents set **S** with a tree of black-height h iff

$$\begin{array}{lll}
\mathsf{RT}(\texttt{root}, \mathbf{S}, h) \overset{\text{def}}{=} & \exists\, v, l, r, \mathbf{L}, \mathbf{R}. & \text{there exist element } v \text{ and sets } \mathbf{L} \text{ and } \mathbf{R} \\
& \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land & \text{such that } \mathbf{L}, v \text{ and } \mathbf{R} \text{ make up } \mathbf{S}\text{, and} \\
& \texttt{root} \mapsto v, l, r, \text{false} \ast & \texttt{root} \text{ points to a red node containing } v \text{ and child pointers} \\
& \mathsf{BT}(l, \mathbf{L}, h) \ast & \text{to black trees representing } \mathbf{L} \\
& \mathsf{BT}(r, \mathbf{R}, h) & \text{and } \mathbf{R}\text{, with the same black-height.}
\end{array}$$

There are multiple types of BT. As the empty tree counts as black-rooted, our principal distinction is between empty and non-empty black-rooted trees. We'll name these 'EBTs' and 'NBTs':

$$\mathsf{BT}(\texttt{root}, \mathbf{S}, h) \overset{\text{def}}{=} \mathsf{EBT}(\texttt{root}, \mathbf{S}, h) \lor \mathsf{NBT}(\texttt{root}, \mathbf{S}, h)$$

Empty black-rooted trees are again easy to describe: a `null` pointer and empty heap. We just add that the empty tree has a black-height of 1:

$$\begin{aligned}
\mathsf{EBT}(\texttt{root}, \mathbf{S}, h) = \quad & \texttt{root} = \texttt{null} \land && \texttt{root}\text{ is }\texttt{null}, \\
& \mathbf{S} = \varnothing \land && \text{the tree represents the empty set,} \\
& \mathbf{emp} \land && \text{the heap is empty,} \\
& h = 1 && \text{and we have a black-height of 1.}
\end{aligned}$$

The non-empty black-rooted tree analogous to the multiple node types in a 2,3,4-tree. There are two-nodes, in which both children are also black. There are three-nodes, in which one child is a BT and one is an RT. That is, the RBT has two representations of a 'three-node': where the left child is the RT ('left-leaning'), and where the right child is the RT ('right-leaning'). Finally, there are four-nodes, where both children are RTs. Creating predicates for each of these will become unwieldy, and so we just use our RBT predicate on the subtrees. A non-empty black-rooted tree pointed to by `root` represents set **S** with a tree of black-height *h* iff

$$\begin{aligned}
\mathsf{NBT}(\texttt{root}, \mathbf{S}, h) \overset{\text{def}}{=} \quad & \exists\, v, l, r, \mathbf{L}, \mathbf{R}. && \text{there exist element } v \text{ and sets } \mathbf{L} \text{ and } \mathbf{R} \\
& \mathsf{Compose}(\mathbf{L}, v, \mathbf{R}, \mathbf{S}) \land && \text{such that } \mathbf{L}, v \text{ and } \mathbf{R} \text{ make up } \mathbf{S}\text{, and} \\
& \texttt{root} \mapsto v, l, r, \text{false} \ast && \texttt{root}\text{ points to a red node containing } v \text{ and child pointers} \\
& \mathsf{RBT}(l, \mathbf{L}, h{-}1) \ast && \text{to black trees representing } \mathbf{L} \\
& \mathsf{RBT}(r, \mathbf{R}, h{-}1) && \text{and } \mathbf{R}\text{, with one less black-height.}
\end{aligned}$$

Recursive RBT algorithms

As the RBT data structure is just an augmentation of the BST data structure, the RBT `search` algorithm is basically identical to the BST `search` algorithm, and as such so is the proof of its correctness. I therefore exclude it here.

A recursive RBT `insert` algorithm

`blacken`*ing the root*

The `blacken` function is the last operation to be performed on the tree after an insertion. When `insert_aux` returns an RVT in mid-operation, it has to perform recolorings or rotations to transform the tree into an RBT or RVT at the same height. However, at the root we don't care about maintaining the height, which enables an easier solution: coloring the root of an RVT black turns it into a valid NBT at one greater black-height. We do not need to check whether the returned tree *is* an RVT, because blackening the root of an RT or BT also gives us a valid NBT.

```
module rb.blacken;

import rb.node;

Node* blacken(Node* root) {
```

Function precondition.

$\mathsf{RBT}(\texttt{root}, \mathbf{S}, h) \lor \mathsf{RVT}(\texttt{root}, \mathbf{S}, h)$

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

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

Weakening

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

```
    Member of set containing itself
    BT(root, S, h) ∧ h ∈ { h, h+1 }
    ∃I on h as nh.
    ∃nh. BT(o, S, nh) ∧ nh ∈ { h, h+1 }
  }
  else {
    RT(root, S, h) ∨ NBT(root, S, h) ∨ RVT(root, S, h)
    if (root.black) {
      NBT(root, S, h)
      Weakening
      BT(root, S, h)
      Member of set containing itself
      BT(root, S, h) ∧ h ∈ { h, h+1 }
      ∃I on h as nh.
      ∃nh. BT(o, S, nh) ∧ nh ∈ { h, h+1 }
    }
    else {
      RT(root, S, h) ∨ RVT(root, S, h)
      root.black = true;
      BT(root, S, h+1)
      Member of set containing itself
      BT(root, S, h) ∧ h ∈ { h, h+1 }
      ∃I on h as nh.
      ∃nh. BT(o, S, nh) ∧ nh ∈ { h, h+1 }
    }
    If-rule.
    ∃nh. BT(o, S, nh) ∧ nh ∈ { h, h+1 }
  }

  If-rule. Function postcondition.
  ∃nh. BT(o, S, nh) ∧ nh ∈ { h, h+1 }
  return root;
}
```

The `insert_bal` *helper procedure*

The `insert_bal` procedure is used by `insert_aux` to perform any necessary recoloring or rotations. It takes a tree that may have an RVT as one, specified, subtree, and transforms the tree to return either an RVT or an RBT. More precisely, if the root of the tree passed to it is red, it will return either an RVT or an RBT; if the root of the tree is black, it will return an RBT.

The procedure works as follows. First, it checks the color of the root of the specified subtree. If the subtree is a BT, then the whole tree is either a RT or BT depending on the root color. The tree is then an RBT, and we're done.

If the subtree was red-rooted, it is either an RT or RVT. We check both sub-subtrees to determine which it is. If both sub-subtrees are black, then the subtree itself is an RT. Depending on the color of the root, the entire tree is either an RVT or a NBT, which satisfies the postcondition, and we pass up the tree.

Otherwise, one sub-subtree is an RT, making the subtree an RVT. This must be fixed by recoloring or rotation. Recoloring is ‘cheaper’, so we try that first: if the *other* subtree is red-rooted, then we blacken both subtrees and color the root red, creating an RT.

If the other subtree is black-rooted, we must rotate. At this point, we have four black-rooted sub-subtrees at some height *h*, and three `Nodes` ‘on top’ of these. With either a single or double rotation (as shown in the diagram to follow), we form a new BT at height *h*+1, and return that.

image/svg+xml **Precondition:** Non-empty tree. Subtrees *r*and *l* at height h; *l* is RVT(h), RT(h) or BT(h). *v h r lv h lr ll* **Test color of *l* node** *l* is red *l* is black Left subtree, as black-rooted, must be BT(h). *v h r lv BT(h) lr ll h-1 h-1 l* is either RVT(h) or RT(h). We can testsubtrees; but blackening it is easier if possible. *v h r lv h lr ll h h* **Test color of root of *r*** *r* is black *r* is red *l* is either RVT(h) or RT(h). The subtree *r* is an RT; *v* istherefore black. *v lv h lr ll h h rv RT(h) rr rl BT(h) BT(h) l* is blackened RVT(h) or RT(h) → BT(h+1). *r* is blackenedRT(h) → BT(h+1). The tree is BT(h+1) → RT(h+1); done. *v lv BT(h+1) lr ll h h rv BT(h+1) rr rl BT(h) BT(h) RT(h+1)* Color swaps are not a solution. If *l* is an RVT,we will have to do a rotation. *v BT(h) r lv h lr ll h h* **Blacken l and r. Redden v.** *ll* is an RT(h). *l* is an RVT(h). Therefore *lr* is black, and*v* is black because insertion was into an RT. *v BT(h) r lv RVT(h) lr llr BT(h) BT(h) BT(h) llv lll RT(h) ll* remains an RT(h). *v* becomes root of a new RT(h). *lv* isroot of new BT(h+1). BT(h+1) → BT(h+1); done. *BT(h) r lv RT(h) lr llr BT(h) BT(h) BT(h) llv lll RT(h) v BT(h+1)* **Single rotate right with recoloring. Test color of root of *ll*** *ll* is black-rooted *ll* is red-rooted Color swaps are not a solution. If *l* is an RVT,we will have to do a rotation. *v BT(h) r lv h lr ll h BT(h) ll* is an RT(h). *l* is an RVT(h). Therefore *lr* is black, and*v* is black because insertion was into an RT. *v BT(h) r lrv RT(h) lrr lrl BT(h) BT(h) BT(h) lv lll RVT(h) lv* and *v* become roots of two new RT(h). *lrv* is root of anew BT(h+1). BT(h+1) → BT(h+1); done. *BT(h) r lrv RT(h) lrr lrl BT(h) BT(h) BT(h) lv lll RT(h) v BT(h+1)* **Double rotate right with recoloring. Test color of root of *lr*** *lr* is black-rooted *lr* is red-rooted *l* is an RT(h). If *v* is red, tree RT(h) → RVT(h);if *v* is black, BT(h) → RT(h); both are valid. *v BT(h) r lv RT(h) lr ll BT(h) BT(h)*

```
module rb.insert.balance;

import rb.node;
import rb.blacken;
static import rb.rotate;
```

```
Node* balance(Node* root, int dir) {
  if (red(root.c[dir])) {
    if (red(root.c[!dir])) {
      root.c[!dir] = blacken(root.c[!dir]);
      root.c[dir]  = blacken(root.c[dir]);
      root.black = false;
    }
    else {
      if (red(root.c[dir].c[dir])) {
 root = rb.rotate.single(root, !dir);
      }
      else {
 if (red(root.c[dir].c[!dir])) {
   root = rb.rotate.dbl(root, !dir);
 }
      }
    }
  }
  return root;
}
```

The `insert_aux` *procedure*

This procedure is really an augmented BST `insert`. It is structured in the same way, with the sole addition that the tree is passed through `insert_bal` before returning it. The call to `insert_bal` enables it to satisfy its postcondition. There are really two specifications for `insert`, depending on whether it is passed an RT or a BT:

{ BT(root, S, *h*) } {`o = insert_aux(root, value);`} { RBT(o, S ∪ {value}, *h*) }
{ RT(root, S, *h*) } {`o = insert_aux(root, value);`} { RBT(o, S ∪ {value}, *h*) ∨ RVT(o, S ∪ {value}, *h*) }

Here is how the `insert_aux` procedure works: image/svg+xml *s* is a non-empty tree containing some value *v*at the root and two

possibly empty subtrees. *v l i i r h* **Precondition:** *s* is a tree. We are going to insertthe value *w* into it. *s* The tree *s* is empty, and represents the set

∅. *s BT(h=0)* ∅ ∪ {*w*} = {w}, so we return a single-node tree.Same height, and BT → RT is fine. We're done. *s BT(0) BT(0) RT(0)* **Is the pointer**

to tree *s* null? Pointer to *s* is not null Pointer to *s* is null **Create and return new red node containing *w*. *v* = *w*?** *v* ≠ *w v* = *w w* is already in the set.

Returning the sametree satisfies the specification, so we're done. *w l i i r h* Value *w* is not at the top of the tree. It may bein a subtree. *v* ≠ *w l i i r*

h Value *w* is not at the root. We want it to be inthe left subtree, *l*. *v* > *w l i i r h* **_v_ < _w_?** *v* < *w v* > *w* **(Symmetrical to the case for _v_ > _w_)** The

ins_aux call may have returned an RVT(i).The tree may therefore be invalid. *v* > *w l i i r h* ins_aux , *w* bal() will fix the tree to be either valid

or anRVT. Both are valid ins_aux return values. *v* > *w l i i r h* ins_aux , *w* bal **Assign ins_aux(_l_, _w_) to the _l_ pointer. Apply bal() to the whole**

tree.

```
module rb.insert.insert_aux;

import rb.node;
import rb.insert.balance;

Node* insert_aux(Node* root, int value) {
  Node* o;
   (BT(root, S, h) ∧ sentinel) ∨ (RT(root, S, h) ∧ ¬sentinel)
  if (root == null) {
    o = new Node(value);
  }
  else {
    if (root.value == value) {
      o = root;
    }
    else {
      int dir = root.value < value;
      root.c[dir] = insert_aux(root.c[dir], value);
      o =  balance(root, dir);
    }
  }
   ((BT(root, S ∪ {value}, h) ∨ RT(root, S ∪ {value}, h)) ∧ sentinel) ∨
   ((RT(root, S ∪ {value}, h) ∨ RVT(root, S ∪ {value}, h)) ∧ ¬sentinel)
  return o;
}
```

The `insert` *procedure*

This simple procedure is used on the entire RBT. It makes an auxiliary call to `insert_aux`, then fixes the returned tree using a call to `blacken`. The height of the final tree is either the previous height or one greater than the previous height.

$$\{\,\mathrm{BT}(\texttt{root}, S, h)\,\}\ \{\texttt{o = insert(root, value);}\}\ \{\,\exists nh.\ \mathrm{BT}(\texttt{o}, S \cup \{\texttt{value}\}, nh) \wedge nh \in \{\,h, h{+}1\,\}\,\}$$

```
module rb.insert.recursive;

import rb.node;
import rb.blacken;

Node* insert(Node* root, int value) {
   Function precondition.
   BT(root, S, h)
  Node* i = insert_aux(root, value);
   Specification for insert_aux.
   RBT(i, S ∪ {value}, h) ∨ RVT(i, S ∪ {value}, h)
  Node* o = blacken(i);
   Specification for blacken.
  ∃nh. BT(o, S ∪ {value}, nh) ∧ nh ∈ { h, h+1 }
  return o;
}
```

A recursive RBT `remove` algorithm

Here is how the `balance_black_sibling` procedure works: image/svg+xml **Precondition:** the subtree *l* is black at height *h*,the other subtree is black-rooted at *h*+1. *v l BT(h) rl h rr h rv BT(h+1) [was RT(h+1) or BT(h+2)]* The far child of the sibling is an RT. We willblacken it and do a rotation. *v l BT(h) rl h rr RT(h) rv BT(h+1) [was RT(h+1) or BT(h+2)]* The far child of the sibling is an RT. We cannotblacken it, so we next look to blacken *rl*. *v l BT(h) rl h rr BT(h) rv BT(h+1) [was RT(h+1) or BT(h+2)]* The new root is now a valid RT(h+1), as wasthe root before deletion, so we're done. *v BT(h+1) l BT(h) rl h rr BT(h+1) rv RT(h+1)* **Test the color of the root of *rr*.** *rr* is an RT The new root is now a valid BT(h+2), as wasthe root before deletion, so we're done. *v BT(h+1) l BT(h) rl h rr BT(h+1) rv BT(h+2)* **Single rotate left. Blacken *rr* to be a BT(h+1). Color the new root with the old color.** *rr* is a BT The root was red The root was black The near child of the sibling is an RT. This red will beblackened in the course of a double rotation. *v l BT(h) rll BT(h) rr BT(h) rv BT(h+1) [was RT(h+1) or BT(h+2)] rlv RT(h) rlr BT(h)* The root was previously RT(h+1). It is now RT(h+1).We are done. *v l h rll BT(h) rr BT(h) rv BT(h+1) BT(h+1) rlv RT(h+1) rlr BT(h)* The root was previously BT(h+2). It is now BT(h+2).We are done. *v l h rll BT(h) rr BT(h) rv BT(h+1) BT(h+1) rlv BT(h+2) rlr BT(h)* **Double rotate left. Color previous root black. Color new root with color of previous root.** The root was red The root was black Both children of the sibling are black, so we canredden *rv* to become RT(h). *v l BT(h) rl BT(h) rr BT(h) rv BT(h+1) [was RT(h+1) or BT(h+2)]* We have a BT(h+1). The root was red, so itwas an RT(h+1). We are done. *v l BT(h) rl BT(h) rr BT(h) rv RT(h) BT(h+1)* We have a BT(h+1). The root was black, so itwas a BT(h+2). We pass up the violation. *v l BT(h) rl BT(h) rr BT(h) rv RT(h) BT(h+1)* The root was red The root was black **Redden subtree at *rv*. If the root is black, pass up the violation; else blacken the root and finish. Test the color of the root of *rl*.** *rl* is an RT *rl* is a BT

Here is how the `balance` procedure works: image/svg+xml *rv* is the root of a BT(*h*+1). Its children aretherefore at height *h*. *v l*

BT(h) rl h rr h rv BT(h+1) [was RT(h+1) or BT(h+2)] rv is the root of an RT(*h*+1). Its children aretherefore BT(*h*+1), and the root was a BT(*h*+2). *v l BT(h) rl BT(h+1) rr BT(h+1) rv RT(h+1) [was BT(h+2)]* **Precondition:** the subtree *l* is black at height *h*,the other subtree is non-empty at height *h*+1. *v l BT(h) rl rr rv h+1 [was RT(h+1) or BT(h+2)]* **Test the color of *rv*.** *rv* is black *rv* is red bal_b applied to the red-rooted tree will return a validtree at height *h*+1. We are done. *v l BT(h) rl BT(h+1) rr BT(h+1) rv BT(h+2) h+1* bal_b **Pass through to bal_bwith the same parameters. Single rotate left. Apply bal_b to the left subtree.** The return state of bal_b() will satisfy the postcondition ofbalance(). The violation may still be passed up. *v l BT(h) rl h rr h rv BT(h+1)* bal_b

Here is how the `remove_aux` procedure works: image/svg+xml The tree *s* is empty, and represents the set ∅.∅ \ {*w*} = ∅, so we return the same tree. *s BT(h=0)* **Precondition:** *s* is a tree. We are going to remove the value *w* if it exists in the tree. *s h s* is a non-empty tree containing some value *v*at the root and two possibly empty subtrees. *v l i i r h* **Is the pointer to tree *s* null?** Pointer to *s* is not null Pointer to *s* is null The value *w* to remove is at the root, and inneither subtree. Subtrees may be empty. *w l i r i h* **Is *v* = *w*?** *v* = *w v* ≠ *w* The value *w* to remove is not at the root. If itis in the tree, it is in a subtree. Which one? *v* ≠ *w l i r i h* **Is *v* < *w*?** *v* < *w v* > *w* **(Symmetrical to the case for *v* > *w*)** The value *w* to remove, if it is in the tree, is in*l*. We need to remove from *l*. *v* > *w l r i h i* The tree does not contain *w*. If not*fixed*, thenthe rem_aux call passed up a tree at height *i*-1. *v* > *w l r i h* rem_aux , *w* **Apply rem_aux to the left subtree.Record its return value *fixed*.** The call to rem_aux returned a tree at height *i*, and whichshall not cause a red violation, and so we're done. *v* > *w l r h* rem_aux , *w i i* ***fixed*?** No, ¬*fixed* Yes, *fixed* rem_aux returned a tree at height *i*-1, which requiresbalancing. Pass through bal(), which may pass up violation. *v* > *w l r h* rem_aux , *w i i-1* bal The value *w* to remove is at the root. Subtree*l* is not empty; *r* might be so. *w l r i h i* The value *w* to remove is at the root. Subtree*l* is empty. We want to return the values in *r*. *w BT(0) l r 0 h* The root is at the same height as the subtrees,so *h*=0. *r* must be empty. *w BT(0) l BT(0) r RT(0)* RT(0) → BT(0) is the same height and cannot cause a redviolation. Therefore we're done. *s BT(0)* **Return the null pointer and finish.** Just returning *r* will cause a black violation,because the black root is missing from it. *w BT(0) l r 0 BT(1)* **Test the color of the root** The root is black The root is red A non-null height-0 subtree must be red withtwo null subtrees. *w BT(0) l BT(1) r RT(0) BT(0) BT(0)* BT(1) → BT(1) is the same height and color. We're done. *r BT(0) BT(1) BT(0)* **Blacken *r* and return it.** Both subtrees represent ∅. The tree represents{w}, and {w} \ {w} = ∅, so we must return null. *BT(0) l r BT(1) BT(0) w* We return the empty set represented by the null pointer, butwe reduced the black height, so we pass up the violation. *s BT(0)* **Return the null pointer and pass up violation. *r* = null?** *r* ≠ null *r* = null ***l* = null?** *l* ≠ null l = null The value *w* to remove is at the root. Neithersubtree is empty. *w l r i h i* Subtree *r* is empty. Subtree *l* is non-null atheight 0, so must be RT(0), so root is black. *w BT(1) BT(0) l r BT(0) BT(0) RT(0)* **Blacken *l* and return it. (Symmetrical case.)** The tree does not contain *w*. If not *fixed*, thenthe rem_aux call passed up a tree at height *i*-1. *max l r i h* rem_aux , *max* **r = null?** r ≠ null *r* = null **Remove *max*, the maximum value in *l*, from *l*.Replace *w* with *max*. Record return value *fixed*.** The call to rem_aux returned a tree at height *i*,and so we're done. *max l r i i h* rem_aux , *max* rem_aux returned a tree at height *i*-1, which requiresbalancing. Pass through bal(), which may pass up violation. *max l r h* rem_aux , *max i i-1* bal

***fixed*?** No, ¬*fixed* Yes, *fixed*

```
module rb.remove.recursive;

import rb.node;
import rb.blacken;
static import rb.rotate;
import rb.print;

import std.stdio;


int findMax(Node* root) {
   T(root, S)
  int max;

  if (root.c[1]) {
     R != empty
    max = findMax(root.c[1]);
     max is max(R) max(R) is max(S)
  }
  else {
    R is empty
    max = root.value;
    max is max({value}) max is max(S)
  }
  max is max(S)
  return max;
}

Node* clr(bool c, Node* root, bool* fixed) {
  root some color * T(root.c[0], L, h) * T(root.c[0], R, i) ∧ !fixed
  *fixed = true;
  root.black = c;
  root.c[0].black = root.c[1].black = true;

  fixed ∧ (c == red ∧ RT(root, S)) || (c == black ∧ EBT(root, S))
  return root;
}

Node* balB(Node* root, int dir, bool* fixed) {
  !fixed
  Node* o;

  if (red(root.c[!dir].c[!dir])) {
    T(fix, h) * root * T(near, h) * black sibling * RT(far, h)
    bool oldColor = root.black;
    root = rb.rotate.single(root, dir);
    T(fix, h) * red root * T(near, h) * black sibling * RT(far, h)
    o = clr(oldColor, root, fixed);
    fixed ∧
  }
  else {
    if (red(root.c[!dir].c[dir] )) {
      o = clr(root.black, rb.rotate.dbl(   root, dir), fixed);
    }
    else {
      *fixed = !root.black;
      root.black = true;
      root.c[!dir].black = false;
      o = root;
    }
  }

  return o;
}


Node* balance(Node* root, int dir, bool* fixed) {
  one less BH in `dir`. root.c[!dir] != null.
  if (red(root.c[!dir])) {  // do case-reducing rotation.
    root = rb.rotate.single(root, dir);
    root.c[dir] = balB(root.c[dir], dir, fixed);
    return root;
  }
  else return balB(root, dir, fixed);
}


Node* try_blacken(Node* root, bool b, bool* fixed) {
  if (!b) {
    *fixed = true;
    return root;
  }
  else if (red(root)) {
    *fixed = true;
    return blacken(root);
  }
  else return root;
}


Node* remove_aux(Node* root, int value, bool* fixed) {
  if (root == null) {
    *fixed = true;
    return root;
  }
  else if (root.value == value) {
    if (root.c[0]) {
      if (root.c[1]) {
 int max = findMax(root.c[0]);
 root.value = max;
 root.c[0] = remove_aux(root.c[0], max, fixed);
 return (*fixed) ? root : balance(root, 0, fixed);
      }
      else return try_blacken(root.c[0], root.black, fixed);
    }
    else return try_blacken(root.c[1], root.black, fixed);
  }
  else {
    int dir = root.value < value;
    root.c[dir] = remove_aux(root.c[dir], value, fixed);
    return (*fixed) ? root : balance(root, dir, fixed);
  }
}


Node* remove(Node* root, int value) {
  bool* dummy = new bool;
  *dummy = false;
  return blacken(remove_aux(root, value, dummy));
}
```

Concurrent indexes

What is a concurrent index?

The previous algorithms and data structures are designed for sequential access. Performing concurrent operations on them results in undefined behavior.

One naive implementation of a concurrent index might simply enforce that, at any one time, at most one operation from {`search`, `insert`, `remove`} is running. A slightly better implementation might restrict to either any number of `read`s, or one of {`insert`, `remove`}.

This is still too restrictive: looking at the various indexing algorithms, many of them should, with minimal modification, permit multiple concurrent `insert` operations—the BST, for example. With no knowledge of implementation, what restrictions should we expect from an index? We might make the observation that, conceptually, the keys in an index are independent. The fact that key `k` is in the index or not, or that it has some value in the index, provides no information about key `l` (where $k \neq l$). Therefore, we might expect restrictions to be placed only individual keys, rather than on the entire index. For example, we might have the only restriction that, for any key `k`, at most one operation from {`search(k)`, `insert(k, _)`, `remove(k)`} is running.

A concurrent index API specification

$\{\mathsf{In}(i, k, v)\}$	`r = i->search(k);`	$\{\mathsf{In}(i, k, v) \land r = v\}$
$\{\mathsf{Out}(i, k)\}$	`r = i->search(k);`	$\{\mathsf{Out}(i, k) \land r = \mathrm{null}\}$
$\{\mathsf{In}(i, k, v')\}$	`i->insert(k, v);`	$\{\mathsf{In}(i, k, v')\}$
$\{\mathsf{Out}(i, k)\}$	`i->insert(k, v);`	$\{\mathsf{In}(i, k, v)\}$
$\{\mathsf{In}(i, k, v)\}$	`i->remove(k);`	$\{\mathsf{Out}(i, k)\}$
$\{\mathsf{Out}(i, k)\}$	`i->remove(k);`	$\{\mathsf{Out}(i, k)\}$

Concurrent index algorithms

The ‘array’ approach to indexing, with one ‘slot’ for every possible key in the universe of keys, would provide this, but carries its usual disadvantages. The hash table might provide a similar level of concurrency—one `remove` operation per hash value—but again carries its usual disadvantages.

What about the BST? We might expect highly concurrent `search`, as this does not mutate the tree. Our `insert` operations may also be, as they only mutate leaves. The `remove` operation only mutates the removed node and its in-order predecessor, and so might only lock these. It seems the BST offers good support for highly concurrent operations.

Can this support carry through to balanced tree algorithms? The basic reason the BST is promising is that the tree is highly static and operations are highly ‘local’. We can, then, dismiss variations like the splay tree, where every operation performs rotations at every level down to the key under consideration, which must lead to horrendous locking considerations (and to my knowledge, no concurrent splay tree algorithms have been attempted). But at first glance, *all* strictly balanced trees have the same problem: maintaining the invariant that promises balance can require fixing-up the tree all the way along one path from the root to a leaf.

The way most concurrent balanced trees tackle this is to ‘relax’ the balancing invariant: simply, the tree is allowed to be in states that break the guarantee of balance. The balancing logic is moved into a separate operation which can be performed independently of the API operations—*e.g.*, in a separate thread.

AVL tree Many concurrent AVL tree algorithms exist %% \cite{bronson_avl} proposes a relaxed.

Red-Black tree ...

Concurrent Red-Black trees

There’s more than one way to make a ‘concurrent RBT’.

Chromatic trees

Introduced by Nurmi and Soisalon-Soininen in 1991/95. Their tree only stores values in the leaves, and the internal nodes are merely ‘routers’. Their paper uses the red/black edge interpretation of the RB tree.

Instead of two states red/black, each edge is given a weight (an unsigned integer). Red=0 and black=1. Values < 0 impossible. Values > 1 termed ‘overweighted’.

The ‘equal number of black nodes on any path’ rule is transformed into: ‘equal sum of weights on any path’. Without overweighting, this is equivalent (due to the values of red/black).

The ‘no two red nodes’ rule is transformed into: ‘the parent edges of the leaves are not red’.

The data structure spec now provides no guarantees of balancedness. The two rules of the sequential RB tree together guarantee that longest path length / shortest path length less than or equal to 2. The chromatic tree may have any lengths, in the case that all weights are zero (i.e. all edges are ‘red’) except for those of the leaves (which >0).

Note also that the B-tree isomorphism also disappears. (Or does it? In an arbitrarily-long run of reds, is there an analogous B-tree?)

Insertion/deletion perform no rebalancing, and maintain the CRB properties. However, they may introduce violations of the red rule (consecutive reds).

Rebalancing is done by a separate thread. Where the sequential RB tree balancing is done bottom-up (we insert the new node at a leaf, then rebalance moving towards the root), the chromatic rebalancing algorithm is top-down.

Concurrency control is done by three kinds of lock: r, w, x. For any given node, the following patterns are possible:

R	W	X
0	0	0
1...inf	0	0

Each thread, whether reading, writing or updating, locks nodes as it traverses the tree. Rebalancing is done purely by changing node contents. This is the locking scheme of Ellis for AVL trees\cite{ellis}, with modifications. E.g., Ellis proposes that writers w-lock the entire path, so that global balancing can be done; the decoupling makes w-lock coupling sufficient.

Larsen's tree\cite{larsen}

This builds on the chromatic tree, with simplifications to the rebalancing. It requires a \enquote{problem queue}. Hanke finds that Larsen’s tree, due to the queue and due to bottom-up conflict handling, does not perform as well as the chromatic tree.

Hyperred-Black trees

Gabarro et al criticize the Chromatic tree: in a run of red nodes, only the top pair may be updated. This is because the rebalancing rules require that the grandparent is black (recall that this is a guarantee with the sequential RB tree). Runs of red nodes may well happen: sorted insertion is common, and all new nodes are colored red. (BUT in the chromatic tree paper above, leaf nodes (parent edges) are black!) The suggested change is that as well as being hyperblack (‘overweighted’), nodes may be hyperred (‘underweighted’), i.e., weights ≤ 0 are degrees of red, and weights ≥ 1 are degrees of black. The ‘blackness’ definition is maintained. The authors have proofs of correctness. However, they have experimental results that show a less-than-impressive performance *vs.* the chromatic tree.

Concurrent indexes in the wild

Some predicate definitions