Automatically verified implementation of data structures based on AVL trees

Martin Clochard

Automatically verified implementation of data structures based on AVL trees

Martin Clochard[1,2]

[1] Ecole Normale Supérieure, Paris, F-75005
[2] Univ. Paris-Sud & INRIA Saclay & LRI-CNRS, Orsay, F-91405

Abstract. We propose verified implementations of several data structures, including random-access lists and ordered maps. They are derived from a common parametric implementation of self-balancing binary trees in the style of Adelson-Velskii and Landis trees. The development of the specifications, implementations and proofs is carried out using the Why3 environment. The originality of this approach is the genericity of the specifications and code combined with a high level of proof automation.

1 Introduction

Formal specification and verification of the functional behavior of complex data structures like collections of elements is known to be challenging [1,2]. In particular, tree data structures were considered in many formal developments using various verification environments. In this paper, we consider self-balancing binary trees, in the style of the so-called AVL trees invented by Adelson-Velskii and Landis [3]. We design a generic implementation of these self-balancing trees from which we derive and verify three instances: random access lists, ordered maps and mergeable priority queues. To reach the appropriate level of genericity in the common part of this development we use an abstract binary search mechanism, based in particular on a notion of monoidal measure on stored data. This notion is shared with an approach proposed by Hinze and Paterson [4] for the development of another general-purpose tree data structure they called *finger trees*. This abstraction allows us to clearly separate the concepts of balanced trees on one hand and search trees on the other hand.

Our development is conducted using the Why3 program verifier, and automated theorem provers to discharge proof obligations. The genericity of the development is obtained by using a *module cloning* mechanism of Why3, which we present briefly in Section 2. Section 3 develops the structure of self-balancing trees, independently of any notion of search. Then Section 4 presents an abstract notion of search trees based on generic *selectors*. Finally we present and verify the three proposed instances in Section 5. Related work in discussed in Section 6.

The Why3 formalization is available at `http://www.lri.fr/~clochard/AVL/avl-why3.tgz`.

2 Preliminary: Cloning modules in Why3

In Why3, generic development of components is done via the notion of *cloning* [5]. Cloning a module amounts to copy its contents while substituting some abstract symbols (types, predicates, functions, procedures) and eliminating some axioms by creating proof obligations for them. In case of procedure substitution, proof obligations are generated as well to check for specification inclusion. This cloning mechanism is used both as an instantiation mechanism for generic development as well as a way to declare standard parameters. For example, suppose that we want to write a development generic with respect to a structure of monoid. Then fresh parameters can be created by cloning a standard abstract module for monoid:

```
module Monoid
  type t
  constant zero : t
  function op t t : t
  axiom neutral : forall x. op x zero = x = op zero x
  axiom associative : forall x y z. op (op x y) z = op x (op y z)
end
module Generic
  clone export Monoid
  (* Generic definitions here *)
end
```

And the generic module can later be specialized to a concrete monoid, say integers, by instantiating the monoid abstract symbols.

```
clone Generic with type t = int,constant zero = Int.zero,
  function op = (+), lemma neutral, lemma associative
```

3 Balanced binary trees in AVL style

We first present a certified implementation of the rebalancing operations for AVL trees. Moreover, this implementation is used to directly derive a logarithmic-time implementation of catenable dequeues.

3.1 Representation and logic model

The very first step of a verified data structure implementation is to decide not only what is its internal representation but as importantly what it should represent, i.e its logical meaning. Having a simple logical reflection of the structure usually makes reasoning much easier. The internal representation of an AVL tree is a binary tree storing the height at every node for efficiency reasons.

```
type t 'a = Empty | Node (t 'a) (D.t 'a) (t 'a) int
```

The namespace `D` corresponds to the abstract data stored in the tree.

The chosen model is the list of data stored in the tree in infix order, since it is the part expected to be invariant by rebalancing. However, in order to specify rebalancing, the tree structure cannot be completely abstracted away because of the height requirements, so we also add the height to this model. Here is the Why3 formalization (`++` denotes list concatenation):

```
type m 'a = { lis : list (D.t 'a); hgt : int } (* type of the model *)
function list_model (t:t 'a) : m 'a = match t with Empty → Nil
  | Node l d r _ → list_model l ++ Cons d (list_model r) end
function height (t:t 'a) : int = match t with Empty → 0
  | Node l d r _ → let hl = height l in let hr = height r in
    1 + if hl < hr then hr else hl
  end
function m (t:t 'a) : m 'a = { lis = list_model t; hgt = height t }
```

3.2 Representation invariant

The balancing criterion for AVL is that the difference between the heights of two sibling trees does not exceed a given positive bound. The structural invariants are readily transformed into the following Why3 predicate:

```
predicate c (t:t 'a) = match t with Empty → true
  | Node l d r h → -balancing ≤ height r - height l ≤ balancing ∧
    c l ∧ c r ∧ h = height t
  end
```

Note that the constant `balancing` is left abstract as a positive integer. Most implementations use a concrete value, which is a trade-off between the potential tree depth and the cost of re-balancing the tree. Since the only impact of keeping it abstract showed to be writing a name instead of a constant, that decision was left to client code.

3.3 Code and Verification

Balancing is performed via smart constructors for tree nodes and catenation operators, specified in terms of the model to build the expected lists. The parts about the height are a bit more complex, as the information about the resulting height has to be precise enough for proof purposes. For example, here is the specification for the core balancing routine, which simulates the construction of a tree node when the two child sub-trees are slightly off balance:

```
val balance (l:t 'a) (d:D.t 'a) (r:t 'a) : t 'a
  requires { c l ∧ c r }
  requires { -balancing-1 ≤ (m l).hgt - (m r).hgt ≤ balancing+1 }
  ensures { let hl = (m l).hgt in let hr = (m r).hgt in
    let he = 1 + (if hl < hr then hr else hl) in
    let hres = (m result).hgt in 0 ≤ he - hres ≤ 1 ∧
    (-balancing ≤ hl - hr ≤ balancing → he = hres) }
  ensures { c result ∧ (m result).lis = (m l).lis ++ Cons d (m r).lis }
```

More complex balancing is done via another smart constructor making no hypothesis on the relative height of the two trees, and two catenation operators similar to the node constructors.

As for the verification part, it did not require any human help once the right specifications were written. All proof obligations were completely discharged by automated theorem provers.

Finally, the catenable dequeue implementation is immediate from the balancing code: catenation is provided, and all other operations (push and pop at both ends) are internal sub-routines of rebalancing. It is logarithmic-time, and also features a constant-time nearly-fair scission operation directly derived from pattern-matching over the tree.

4 Selection of elements in balanced trees

AVL trees were first introduced as binary search trees, so most operations over them involve a binary search by comparison. However, Hinze and Paterson [4] have shown that using a generalization of binary search based on monoidal annotations, one could implement a variety of data structures. In this section, we present and verify a generalized implementation of usual AVL routines (insertion, deletion, etc) using a similar approach.

4.1 Monoidal summary

The usual search mechanism in binary search trees is search by comparison using a total order. However, by keeping summaries of the contents in each subtrees, one can provide a variety of other mechanisms. For example, keeping the number of elements in each subtree gives positional information, which can be used to perform efficient random access. Hinze and Paterson [4] proposed monoids as a general mechanism to keep track of those summaries: the content of each subtree is summarized by the sum of the measures of its elements in some given monoid.

We use those annotations as well to provide different search mechanisms. They are integrated in the development with minimal changes as the height bookkeeping is done the same way. We also add parameters corresponding to an abstract monoid and measure.

4.2 Abstracting the binary search mechanism

In their paper about finger trees, Hinze and Paterson suggest to implement most data structure operations using a splitting mechanism, which finds an element where a predicate over the monoidal abstraction of the prefix flips. We could have used this technique, but it has some flaws when considering AVL trees. First and foremost, it completely ignores the fact that the internal tree nodes contain elements that could – and would – be used to guide the search. This is not the case for finger trees as elements are stored in the leaves. Second, the usual insertion/deletion/lookup routines coming with AVL trees would be replaced by

much slower (though still logarithmic-time) implementations based on complete splitting of the tree followed by catenation of the parts back together.

This last part, however, is perfectly fit for specification since what binary search does is exactly selecting a split of the list model.

Splits are formalized by the following definitions:

```
type split 'a = { left : list 'a; middle : option 'a; right : list 'a }
function rebuild (s:split 'a) : list 'a =
  s.left ++ option_to_list s.middle ++ s.right
```

The structure of splits corresponds exactly to the two possible outcomes of a binary search in the tree: either finding an element in a node or ending on an empty leaf. In order to describe the particular splits we wish to find, we use an abstract *selector* parameter:

```
type selector
predicate selected (s:selector) (sp:split (D.t 'a))
predicate selection_possible (s:selector) (l:list (D.t 'a))
```

Informally, the `selector` describes the class of splits we want to find, represented by the `selected` predicate. The `selection_possible` describes the lists in which splits corresponding to the `selector` can be found using binary search. This compatibility mean that one can reduce the problem of finding a split in its class by bisection over the node structure, potentially using the summary of the branches to guide the search. We achieve this decription by adding an abstract routine parameter performing this reduction:

```
type part = Here | Left selector | Right selector
val selected_part (ghost llis rlis:list (D.t 'a))
  (s:selector) (l:M.t) (d:D.t 'a) (r:M.t) : part
  requires { selection_possible s (llis ++ Cons d rlis) }
  requires { l = M.sum D.measure llis ∧ r = M.sum D.measure rlis }
  returns { Here →
      selected s { left = llis; middle = Some d; right = rlis }
    | Left sl → selection_possible sl llis ∧
      forall sp. selected sl sp ∧ rebuild sp = llis →
        selected s { sp with right = sp.right ++ Cons d rlis }
    | Right sr → selection_possible sr rlis ∧
      forall sp. selected sr sp ∧ rebuild sp = rlis →
        selected s { sp with left = llis ++ Cons d sp.left } }
```

Note that the routine is expected to compute new selectors as the reduced problem may be different. Also, we need to ensures that whenever the search ends on a leaf, the only possible split is the selected one. This is expressed by an axiom:

```
axiom selection_empty : forall s:selector. selection_possible s Nil →
  selected s { left = Nil; middle = None; right = Nil }
```

4.3 Certified routines based on binary search

Using the abstract binary search mechanism, we certified the implementation of a generalization of the usual routines over AVL trees: lookup, insertion, deletion,

as well as splitting. The code is skipped here as it is standard, it is just a matter of replacing the decisions usually done by comparison by a case analysis on the result of `selected_part`, interested readers may find the code for those routines in appendix A.3. We then focus on the specifications of those routines. For example, let us see how we could specify insertion. Earlier, we mentioned that those procedures could be build on top of splitting: one could perform insertion by splitting the tree, replacing the potential middle element, and rebuilding it afterwards. It turns out to be the right specification for insertion:

```
val insert (ghost r:ref (split (D.t 'a))) (s:selector)
    (d:D.t 'a) (t:t 'a) : t 'a
  requires { selection_possible s (m t).lis ∧ c t }
  ensures { c result ∧ (m result).lis = !r.left ++ Cons d !r.right }
  ensures { selected s !r ∧ rebuild !r = (m t).lis }
  writes { r }
```

Note that we use a ghost reference instead of an existential quantifier for the split. While using an existential is possible, there are two reasons for using such a reference instead. First, existentially quantified goals tend to be hard for automated provers. In this case, we can very easily give them an explicit witness via the reference. Second, in case the client code is really hard to prove, one can help the automated provers by providing logical cuts. Such cuts will be much easier to write if the existentially quantified value is known.

The three remaining routines have pretty similar specification:

- Deletion is the converse of insertion: any potential middle element is removed of the split before rebuilding.
- Lookup amounts to return the middle of the split.
- Splitting returns a split with lists represented by AVL trees.

5 Verified Instances

5.1 Random-access sequences

The first instance use positional selection, which naturally gives random-access sequences. This is obtained by instantiating the monoid by integers and measuring all elements by 1, which gives fast access to the length of the sub-lists. Using that information, binary search is done by finding in which of the three pieces of the list lies the n-th element. Note that reducing the problem to a sub-list requires the index to change. Also, as random-access lists are completely polymorphic, data elements are instantiated with fully polymorphic values (`D.t 'a = 'a`).

The formal specification of this kind of selection is straightforward:

```
type selector = { index : int; hole : bool }
predicate selected (s:selector) (sp:split 'a) =
  s.index = length sp.left ∧ (s.hole ↔ sp.middle = None)
predicate selection_possible (s:selector) (l:list 'a) =
  if s.hole then 0 ≤ s.index ≤ length l else 0 ≤ s.index < length l
```

The extra boolean field is intended to specify whether we want the list to be cut in two pieces or split around the n-th element. Having both allows to derive most positional operations over random-access lists directly from the abstract selection routines:

- Assignment is derived from abstract insertion
- Positional insertion is also derived from abstract insertion
- Positional lookup is implemented by abstract lookup
- Positional deletion is derived from abstract deletion
- Both kind of positional splits are derived from abstract splitting.

However, the specifications had to be rewritten as the obtained ones did not match the desired ones for random-access lists. This was done by writing specification wrappers around those operations. The automatic verification of this wrapper did not required human help beyond making explicit a trivial induction correlating the length of the list to its monoidal summary.

As an example of the resulting specifications, here is the one for the assignment procedure:

```
val set (n:int) (d:'a) (l:t 'a) : t 'a
  requires { c l ∧ 0 ≤ n < length (m l) }
  ensures { c result ∧ length (m result) = length (m l) }
  ensures { forall i:int. i ≠ n → nth i (m result) = nth i (m l) }
  ensures { nth n (m result) = Some d }
```

5.2 Maps and sets

Another instance correspond to the abstract data structures usually implemented with AVL trees: ordered sets and associative arrays. Those naturally correspond to the case of comparison-based binary search in sorted sequences.

Several new parameters are added to reflect the ordering structure.

- An abstract key datatype
- A function extracting keys from data
- A computable ordering relation over keys

From those parameters, binary search trees lookup, insertion, and deletion are obtained by using straightforward instances for the selection parameters:

- Selection is done by keys, so the selector type is instantiated by keys.
- Selection can be done only in sorted sequences.

  ```
  predicate selection_possible (_:'b) (l:list (D.t 'a)) = increasing l
  ```

- A split is selected by a key if it corresponds to elements with keys lower, equal and greater than the selector respectively.

  ```
  predicate selected (k:Key.t) (sp:split (D.t 'a)) =
    upper_bound k sp.left ∧ lower_bound k sp.right ∧
    match sp.middle with None → true | Some d → eq k (key d) end
  ```

– Binary search is done by mirroring comparison

As we do not need the extra summaries here, the monoid is instantiated by the unit monoid. Although this instantiation yields a perfectly valid implementation for ordered associative arrays, it is unsatisfactory from the specification point of view as the data structure is still modeled by a list. This is not a suitable model for associative arrays, which are intended to represent finite key-values mappings. In order to get specifications based on such modeling, we wrote specification wrappers over the implementation. The new model was obtained by interpreting the previous list model as an association list:

```
type m 'a = { func : Key.t → option (D.t 'a); card : int }
function association (l:list (D.t 'a)) : Key.t → option (D.t 'a) =
  match l with
  | Nil → \k. None
  | Cons d q → \k. if eq k (key d) then Some d else association q k
  end
function m (t:t 'a) : m 'a = {
  func = association (AVL.m t);
  card = length (AVL.m t);
}
predicate c (t:t 'a) = AVL.c t ∧ increasing (AVL.m t)
```

Note that this instantiation does not break the abstraction barrier: the specification wrappers and selectors are based on the model of the AVL trees only.

The obtained specifications indeed corresponds to the expected behavior of an associative array. For example, here is the specification for insertion (others look alike):

```
val insert (d:D.t 'a) (t:t 'a) : t 'a
  requires { c t }
  ensures { c result }
  ensures { c result ∧ (if (m t).func (key d) = None
      then (m result).card = (m t).card + 1
      else (m result).card = (m t).card) ∧
    forall k:Key.t. if eq k (key d) then (m result).func k = Some d
      else (m result).func k = (m t).func k }
```

The verification of those specification wrappers was not completely immediate, as it required a number of facts over sorted association lists that could be proved only by induction. Mostly, it required a bridge lemma between the notion of selected split of the list and a similar notion stated in terms of key-value mappings. This required a small amount of manual work to state the corresponding lemmas and to make explicit the inductive structure of the proofs.

Finally, certified implementations of ordered maps and sets were derived from this implementation by writing immediate specification wrappers over instances of this implementation.

– Sets were obtained from an instance identifying keys and elements. For specifications, the model was reduced to the predicate of presence.

– Maps were obtained by instantiating the elements with couple formed of a key and a polymorphic value. As keys were irrelevant as outputs of the model mapping, that part was removed from specifications.

5.3 Mergeable Priority queues

The last instance presented in this paper is selection of the element with the smallest key, which is an immediate implementation of mergeable priority queues. The corresponding monoid is the minimum monoid over keys extended with the positive infinity, which gives fast access to the smallest key of sub-lists. Then, binary search can be done by taking a path leading to a minimum element. For the ordering and keys, we reuse the same setting as for associative arrays.

The specification of minimum selection is quite direct as well: it amounts to say that the split has a middle element and that it is minimal.

```
type selector = unit
predicate selected (_:unit) (sp:split (D.t 'a)) =
  match sp.middle with
  | None → false
  | Some d → lower_bound (key d) sp.left ∧ lower_bound (key d) sp.right
  end
predicate selection_possible (_:unit) (l:list (D.t 'a)) = l ≠ Nil
```

Binary search can obviously be done by taking the path to the minimum element.

From this instantiation, one can map the priority queue operations to the abstract AVL ones:

– finding the minimum is exactly lookup.
– removing the minimum is deletion.
– adding an element can be implemented by prepending the new element.
– merging two priority queues can be done by catenation.

Again, those operations were wrapped under new specifications with a better-suited model. Since the order of the elements inside the structure is irrelevant, the priority queue is represented by a finite bag:

```
type m 'a = { bag : D.t 'a → int; card : int }
function as_bag (l:list 'a) : 'a → int = match l with
  | Nil → \x. 0
  | Cons x q → \y. if x = y then as_bag q y + 1 else as_bag q y
  end
```

Here is an example of the final specifications, namely the one for the `remove_min` operation:

```
val remove_min (ghost r:ref (D.t 'a)) (t:t 'a) : t 'a
  requires { c t ∧ (m t).card ≥ 1 }
  writes  { r }
  ensures { c result ∧ (m t).card = (m result).card + 1 ∧
    (m t).bag !r = (m result).bag !r + 1 ∧
    (forall d. d ≠ !r → (m t).bag d = (m result).bag d) ∧
    (forall d. (m t).bag d > 0 → le (key !r) (key d)) }
```

6 Related Work

Verified balanced binary search trees. Numerous verified implementation of balanced binary search trees have been proposed. For example, implementations of AVL trees have been verified in Coq [6], Isabelle [7] and ACL2 [8], and similar verifications of red-black trees have been carried out [9,10,11,12]. A number of them used some kind of proof automation, though developments in proofs assistants are mostly manual. However, those implementations are not as generic as they are restricted to the usual binary search trees.

Finger trees. Finger trees were introduced by Hinze and Paterson [4] as a structure general enough to derive several common data structure from it, which is exactly the same level of genericity intended by our certified implementation. However, rather few certified implementation of finger trees were carried out. Mathieu Sozeau verified the implementation of Hinze and Paterson using the Program extension of Coq [13], and another verification was carried out using Isabelle [14]. In both cases, proofs are mostly manual while our implementation is verified correct with nearly no human interaction. Also, excepted for Sozeau's implementation of random-access sequences, there was no attempt to check that the specification was indeed strong enough to verify the common instances.

7 Conclusions and Perspectives

This work presents a generic certified implementation of AVL trees and the verification of three common data structures derived from that generic core. The verification overhead is rather light, as it corresponds to less than 1400 non-empty lines of Why3 for the whole presented development, which amounts to about 550 lines of implementation. Moreover, most of this verification cost corresponds to code specification, as proofs are mostly discharged by automated provers without needing to provide hints. Details about the development size can be found in appendix.

In conclusion, we would like to assess that a high level of abstraction in programs like the one used in this development mingles very well with proof automation. This is first caused by the separation between unrelated concepts like balancing and binary search. Mixing such concepts in a single routine widen greatly the search space of automated provers, as they cannot identify that only one of those is related to a particular goal. Also, another benefit of genericity is that some routines are written and proven once, while proving directly the instances would require a lot of duplication.

We expect that such generic approaches would help to the development of certified libraries, which are a first step towards developing verified programs of consequent sizes.

References

1. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing (2007) 159–189

2. Leino, K.R.M., Moskal, M.: VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In: Proceedings of Tools and Experiments Workshop at VSTTE. (2010)

3. Adel'son-Vel'skiĭ, G.M., Landis, E.M.: An algorithm for the organization of information. Soviet Mathematics–Doklady **3**(5) (September 1962) 1259–1263

4. Hinze, R., Paterson, R.: Finger Trees: A Simple General-purpose Data Structure. J. Funct. Program. **16**(2) (March 2006) 197–217

5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011) 53–64

6. Filliâtre, J.C., Letouzey, P.: Functors for Proofs and Programs. In: Proceedings of The European Symposium on Programming. Volume 2986 of Lecture Notes in Computer Science., Barcelona, Spain (April 2004) 370–384

7. Nipkow, T., Pusch, C.: AVL Trees. Archive of Formal Proofs (March 2004) http://afp.sf.net/entries/AVL-Trees.shtml, Formal proof development.

8. Ralston, R.: ACL2-certified AVL Trees. In: Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and Its Applications. ACL2 '09, ACM (2009) 71–74

9. Appel, A.: Efficient Verified Red-Black Trees (2011) http://www.cs.princeton.edu/~appel/papers/redblack.pdf.

10. Charguéraud, A.: Program Verification Through Characteristic Formulae. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. ICFP '10, ACM (2010) 321–332

11. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In Kaufmann, M., Paulson, L.C., eds.: Interactive Theorem Proving. Volume 6172 of Lecture Notes in Computer Science., Springer (2010) 339–354

12. Various authors: VACID-0 solutions http://vacid.codeplex.com.

13. Sozeau, M.: Program-ing finger trees in Coq. In Hinze, R., Ramsey, N., eds.: 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, ACM Press (2007) 13–24

14. Nordhoff, B., Körner, S., Lammich, P.: Finger Trees. Archive of Formal Proofs (October 2010) http://afp.sf.net/entries/Finger-Trees.shtml, Formal proof development.

A Size of the development

A.1 Lines of code

	Lines of implementation	Lines of specification/proof hints
Balancing	174	196
Selection	91	59
Associative Array	58	237
Maps	40	180
Sets	31	139
Random-access sequences	63	143
Priority queue	78	219
Association list properties	–	119
Sorted list theory	–	33
Preorder theory	–	22
Monoid theory	–	30
Total	535	1377

Overall, the proof hints corresponds to about 40 lemmas.

A.2 Verification setting

The verification was carried out using the development version of Why3, which features abstract program substitution during cloning. Though not released yet at the time this paper is written, this corresponds to the version 0.84. Each goal was discharged using one of the four SMT solvers Alt-Ergo, CVC3, CVC4 or Z3. The time limit was set to 5 seconds for the vast majority of them.

Prover	discharged goals	average time	maximum time
Alt-Ergo	471	0.29s	6.76s
CVC3	283	0.29s	3.01s
CVC4	66	0.68s	7.39s
Z3	11	1.37s	4.84s

A.3 Code for insertion, lookup and deletion

Note: the expensive list-manipulating code is ghost, and as such is not executed.

```
let rec insert (ghost r:ref (split (D.t 'a))) (s:selector)
  (d:D.t 'a) (t:t 'a) : t 'a
  requires { selection_possible s (m t).lis ∧ c t }
  ensures { c result ∧ (m result).lis = !r.left ++ Cons d !r.right }
  ensures { selected s !r ∧ rebuild !r = (m t).lis }
  writes { r }
  (* extra postcondition needed to prove the recursion. *)
  ensures { 1 ≥ (m result).hgt - (m t).hgt ≥ 0 }
```

```
  variant { (m t).hgt }
= match view t with
  | AEmpty → r := { left = Nil; middle = None; right = Nil };
    singleton d
  | ANode tl td tr _ _ → match selected_part (m tl).lis (m tr).lis
      s (total tl) td (total tr) with
    | Left sl → let nl = insert r sl d tl in
      { e with right = (!r).right ++ Cons td (m tr).lis }; balance nl td tr
    | Right sr → let nr = insert r sr d tr in
      r := { !nr with left = (m tl).lis ++ Cons td (!r).left }; balance tl td nr
    | Here → r := { left = (m tl).lis;
        middle = Some td;
        right = (m tr).lis };
      node tl d tr
    end
  end

let rec remove (ghost r:ref (split (D.t 'a))) (s:selector)
  (t:t 'a) : t 'a
  requires { selection_possible s (m t).lis ∧ c t }
  ensures { c result ∧ (m result).lis = !r.left ++ !r.right }
  ensures { selected s !r ∧ rebuild !r = (m t).lis }
  writes { r }
  (* needed to prove the recursion *)
  ensures { 1 ≥ (m t).hgt - (m result).hgt ≥ 0 }
  variant { (m t).hgt }
= match view t with
  | AEmpty → r := { left = Nil; middle = None; right = Nil}; t
  | ANode tl td tr _ _ → match selected_part (m tl).lis (m tr).lis
      s (total tl) td (total tr) with
    | Left sl → let nl = remove r sl tl in
      r := { !r with right = (!r).right ++ Cons td (m tr).lis; balance nl td tr
    | Right sr → let nr = remove r sr tr in
      r := { !r with left = (m tl).lis ++ Cons td (!r).left; balance tl td nr
    | Here → r := { left = (m tl).lis;
        middle = Some td;
        right = (m tr).lis };
      fuse tl tr
    end
  end

let rec get (ghost r:ref (split (D.t 'a))) (s:selector)
  (t:t 'a) : option (D.t 'a)
  requires { c t ∧ selection_possible s (m t).lis }
  ensures { selected s !r ∧ rebuild !r = t.m.lis }
  ensures { result = (!r).middle }
  writes { r }
  variant { (m t).hgt }
= match view t with
  | AEmpty → r := { left = Nil; middle = None; right = Nil }; None
```

```
| ANode tl td tr _ _ → match selected_part (m tl).lis (m tr).lis
    s (total tl) td (total tr) with
  | Left sl → let res = get r sl tl in
    r := { !r with right = (!r).right ++ Cons td (m tr).lis }; res
  | Right sr → let res = get r sr tr in
    r := { !r with left = (m tl).lis ++ Cons td (!r).left }; res
  | Here → r := { left = (m tl).lis;
      middle = Some td;
      right = (m tr).lis };
    Some td
  end
end
```