Efficient Verified Red-Black Trees

ANDREW W. APPEL
Princeton University, Princeton NJ 08540, USA
(*e-mail:* appel@princeton.edu)
September 2011

Abstract

I present a new implementation of balanced binary search trees, compatible with the MSets interface of the Coq Standard Library. Like the current Library implementation, mine is formally verified (in Coq) to be correct with respect to the MSets specification, and to be balanced (which implies asymptotic efficiency guarantees). Benchmarks show that my implementation runs significantly faster than the library implementation, because (1) Red-Black trees avoid the significant overhead of arithmetic incurred by AVL trees for balancing computations; (2) a specialized delete-min operation makes priority-queue operations much faster; and (3) dynamically choosing between three algorithms for set union/intersection leads to better asymptotic efficiency.

1 Introduction

An important and growing body of formally verified software (with machine-checked proofs) is written in pure functional languages that are embedded in logics and theorem provers; this is because such languages have tractable proof theories that greatly eases the verification task. Examples of such languages are ML (embedded in Isabelle/HOL) and Gallina (embedded in Coq). These embedded pure functional languages extract to ML that can be compiled with optimizing compilers, so it's not crazy to think of building real software this way that's efficient enough to solve real problems.

Efficent programs need efficient algorithm-and-data-structure libraries, subject to this restriction that the programs are purely functional. Although some authors are experimenting with ways to evade the pure-functional restriction in Gallina (Nanevski *et al.*, 2008; Armand *et al.*, 2010), I believe we can get quite far without evasions.

Balanced binary search trees are an important data structure in computer science, and particularly so in pure functional programming. They are used to implement the abstract type of sets over totally ordered keys, with $O(\log N)$ insertion, lookup, and deletion. In a programming language with a sufficiently powerful module system (MacQueen, 1990) such as that of Standard ML or OCaml, one can specify the interface of the *set* abstract data type, parametrized over another abstract data type of totally ordered *keys*. Filliâtre and Letouzey (Filliâtre & Letouzey, 2004) show that in the Coq proof assistant, one can go even farther: in the *keys* module are not only the comparison operations on keys, but the specification expressed in logic (the Calculus of Inductive Constructions) that the key-comparison really is totally ordered; and in the *sets* module are the logical correctness specifications of all of the operations, also expressed in logic. For example,

```
Module Type Sets.
Declare Module K : OrderedType.
Parameter set : Type.
Parameter In : K.t → set → Prop.
Parameter insert : K.t → set → bool.
Parameter member : K.t → set → set.
Axiom insert_spec : ∀s x y, In y (insert x s) ↔E.eq y x ∨ In y s.
Axiom member_spec : ∀s x, member x s = true ↔In x s.
...
End Sets.
```

Here, K contains the operations *and specifications* of a total order, and Sets contains the operations and specifications of the operations on sets of keys.

Filliâtre and Letouzey then implemented this specification with balanced binary search trees: that is, they wrote *programs* for operations such as insert and member, and wrote machine-checked *proofs* for specifications such as insert_spec and member_spec. In fact, they compared the performance of AVL trees with Red-Black trees. The Red-Black trees performed faster, but for other reasons they chose the AVL trees for the Coq Library; since then, Filliâtre's Red-Black implementation is available in the Coq "User Contributions[1]" while the AVL trees are in the MSets module of the Coq Library.

My research group is building a verified implementation of the paramodulation algorithm for resolution theorem proving, we use MSets to keep sets of clauses, priority queues of clauses, and mappings from names to various types. I wanted our program to run faster, so I investigated an alternate implementation of MSets. My implementation is probably similar in many ways to Filliâtre's Red-Black implementation, but in this paper I want to focus on three specific design issues, which I discuss below.

In a binary search tree, each nonempty node has a *key* and two subtrees; every key within the left subtree is less than the node's key, and every key within the right subtree is greater. In a *balanced* binary search tree, each node has some extra information to keep track of balance conditions, that is, to make sure that the heights of the two subtrees are approximately the same. When a tree goes (or is about to go) out of balance, a *rotation* can adjust it. The height of an approximately balanced tree is $O(\log N)$, so the insert and lookup costs are logarithmic.

AVL trees are the granddaddy of efficient balanced binary search trees, invented in 1962. Each node keeps a *height* memoizing the height of that subtree, and by comparing heights one can know when to rotate. Instead of storing the raw height, one can store a 2-bit *balance factor*, the difference between the heights of the left and right subtrees. In a conventional programming language, a word-aligned record with key+left+right+extra takes 4 words whether the "extra" is a 2-bit balance factor or a short integer height, so it does not matter which representation is used.

Red-black trees keep only 1 bit of balance information: the tree has *black* nodes and *red* nodes. The Red-Black invariant, which I will describe later, guarantees $O(\log N)$ efficiency

[1] http://coq.inria.fr/pylons/contribs/files/FSets/trunk/FSets.FSetRBT.html

for insert and lookup operations. Of course, 1 bit of balance information is as costly as a whole word, in a typical word-aligned implementation.[2]

In this article I present my Red-Black Tree implementation of MSets. When extracted to ML code, it's significantly faster than the existing Coq Library implementation, for three reasons:

1. Bookkeeping of heights in MSetAVL using the Z type of the Coq library, is expensive; bookkeeping of reds and blacks is much cheaper. (The AVL *balance factors* would probably be faster than Z but not as fast as Red-Black.) Filliâtre's Red-Black trees, if revived, would probably perform as fast as mine.
2. I combine min_elt (find the minimum element) and delete into a single operation delete_min that does not have to do any comparisons at all. This was omitted from the Coq Library MSets interface, with unfortunate consequences for clients that want to use MSets as priority queues.
3. Union, intersection, and similar operations have three implementations. Sets s and t can be unioned in time $|s|\log|t|$ (when $|s| \leq |t|$) by insert each element of s into t; or in $|t|\log|s|$ time (when $|t| \leq |s|$); or in $s+t$ time by flattening both trees, merging, and rebuilding a new one. The intersection and diff operations are analagous. Depending on the sizes of s and t, I choose between these three methods. Measuring the size of a Red-Black tree would take linear time, so I measure the approximate log of the size (the "black-node height") in $\log N$ time.

Reasons 2 and 3 are not specific to Red-Black trees, and would apply to most balanced binary search tree data structures.

2 Why are AVL trees slow?

Integer arithmetic in the Coq standard library is constructed from inductive structures as follows.

Inductive positive := xI : positive → positive | xO : positive → positive | xH : positive.

Inductive Z := Z0 : Z | Zpos : positive → Z | Zneg : positive → Z.

A positive number is either 1, represented by xH, or $2n$, represented by xO(n), or $1+2n$, represented by xI(n). Whenever we reason about integers (type Z) in Coq, we are in fact reasoning about data structures such as (Zneg(xI(xO(xH)))) and (Zpos(xO(xI(xI(xH))))). Such reasoning takes time (typically) linear in the size of the data structure, and logarithmic in the size of the numbers represented.

This explains why the implementation of AVL trees in the Coq library performs slowly for insert: balance numbers are represented as Z, and thus there is a $\log N$ penalty; effectively insert k t takes time $\log^2|t|$.

One might think, "when extracting to ML programs, why can't we represent Z as a single-word native integer, and do machine-native arithetic?" Indeed, Filliâtre and Letouzey

[2] One can do Red-Black trees with 0 bits of balance information, at the cost of extra comparisons Ex. 13.65, p. 560. But I want the data structure to be efficient even in regimes where comparisons are expensive, so this technique is not attractive.

write, “We could parameterize the whole formalization of AVL trees with respect to the arithmetic used for computing heights, using yet another functor. But we would lose the benefits of the `Omega` tactic (the decision procedure for Presburger arithmetic) which is of heavy used in this development.”

However, I believe they are underestimating a significant problem: machine arithmetic arithmetic can overflow. If one axiomatizes this overflow then one has many proof obligations of the form $x+y<2^{31}$. In practice, these proof obligations are overwhelmingly nasty. Furthermore, the specifications themselves would get much more complicated. One of the most important methods by which people have made progress in verified algorithms is by the clever trick of using infinite-precision integers, not because they will ever overflow, but so that the proofs are simpler.

Theorem: If we were to use machine integers to store the balance information for AVL trees, those integers would never overflow.

Proof. The height stored in an AVL tree never exceeds the log of the number of pointers in the tree, and thus on any machine where integers are at least as large as pointers, the height of the tree is representable. ∎

It is exceedingly difficult to convert this theorem to a machine-checkable result, and I will not even try. Thus, one can see why Filliâtre and Letouzey did not attempt using fixed-precision arithmetic for heights of AVL trees.

But there’s a simpler way. One should simply use a representation of balanced search trees that does not require integers: Red-Black trees.

3 Looking up keys in search trees

In Coq the Red-Black tree data structure is simply,

```
Local Notation "'key'" := K.t.
Inductive color := Red | Black.
Inductive tree : Type :=
 | E : tree
 | T: color → tree → key → tree → tree.
```

The implementation is a functor over any totally ordered type (module K: Orders.OrderedType).

The beautiful thing about Red-Black trees (or AVL trees) is that the lookup function can ignore all the balance information and just use the *searchtree* property:

```
Fixpoint member (x: key) (t : tree) : bool :=
  match t with
  | E ⇒ false
  | T _ tl k tr ⇒ match K.compare x k with
                        | Lt ⇒ member x tl
                        | Eq ⇒ true
                        | Gt ⇒ member x tr
                        end
  end.
```

But what is the searchtree property? It is that all the elements to the left are less than the node’s key, and so on. In practice we often need to say that t is a searchtree that can appear to the right of some key k_{low}, or to the left of some key k_{high}, or both. That is, we start with an “optional less than”,

```
Definition ltopt (x y : option key) :=
  match x, y with Some x', Some y' ⇒ K.lt x' y'    | _, _ ⇒ True end.
```

Thus, ltopt (Some x) (Some y) means x<y, but ltopt None (Some y) is vacuously true, as is ltopt (Some x) None. Then the searchtree property is defined as,

```
Inductive searchtree: tree → option key → option key → Prop :=
 | STE: ∀lo hi,    ltopt lo hi → searchtree E lo hi
 | STT: ∀c tl k tr lo hi,
       searchtree tl lo (Some k) → searchtree tr (Some k) hi → searchtree (T c tl k tr) lo hi.
```

To specify what it means for the member function to be correct, we write an inductive definition for the *interp*retation of a tree as a predicate on keys; iff the key is present anywhere in the tree (regardless of searchtree properties), then the predicate will be True.

```
Inductive interp: tree → (key → Prop) :=
| member_here: ∀x y c tl tr, K.eq x y → interp (T c tl y tr) x
| member_left: ∀x y c tl tr, interp tl x → interp (T c tl y tr) x
| member_right: ∀x y c tl tr, interp tr x → interp (T c tl y tr) x.
```

If *t* is a bounded search tree, then any key in the interpretation of *t* is in bounds:

```
Lemma interp_range:
  ∀x t lo hi, searchtree lo hi t → interp t x →   ltopt lo (Some x) ∧ ltopt (Some x) hi.
```

And now the correctness of member: for any tree t that is a searchtree, member finds the key k if and only if interp t k.

```
Lemma interp_member:
    ∀x t, searchtree None None t →   (member x t = true ↔interp t x).
```

Proof. The Coq proof script is 18 lines (138 tokens) long. In the forward direction, we can ignore the searchtree property and do induction on t. In the backward direction, we do induction on the inductive predicate searchtree. ∎

The completion of this proof before we even define the balance property demonstrates that, not only can lookup on Red-Black trees ignore the colors—so can the proofs about lookup.

4 Insertion

Insertion into an *unbalanced* binary search tree is easy, and easy to prove correct:

```
Fixpoint unbal_ins x s :=
 match s with
 | E ⇒ T Red E x E
 | T _ a y b ⇒ match K.compare x y with
                        | Lt ⇒ T Red (unbal_ins x a) y b
                        | Eq ⇒ T Red a x b
                        | Gt ⇒ T Red a y (unbal_ins x b)
                        end
 end.
```

I arbitrarily put Red for the color, but trees built this way will not satisfy the Red-Black property; they will satisfy the searchtree property, and a lemma similar to the interp_insert property that I will define below.

So, if unbalanced insert is easy and correct, then why not do that? Because the trees might not be balanced, and therefore we cannot give $\log N$ guarantees for the operations.

We will make formal, machine-checked proofs of the functional correctness of our operations on search trees. But the proof theory of the Gallina language does not really permit the formal verification of execution-time properties. Instead, we will want formal (machine-checked) proofs that the search trees will have depth no more than $2\log N$. This, combined with our understanding of the recursion depth of the insert and lookup algorithms, will reassure us (in a rigorous but not machine-checked way) that the programs will run fast.

The *Red-Black invariant* is that every path from the root to a leaf has the same number of black nodes, and no such path has two red nodes in a row. Thus each leaf is at most twice as deep as any other leaf, and this means that the height of an N-node tree is at most $2\log N$. We formalize this invariant as follows.

```
Inductive is_redblack : tree → color → nat → Prop :=
| IsRB_leaf: ∀c, is_redblack E c 0
| IsRB_r: ∀tl k tr n, is_redblack tl Red n → is_redblack tr Red n → is_redblack (T Red tl k tr) Black n
| IsRB_b: ∀c tl k tr n,
      is_redblack tl Black n → is_redblack tr Black n → is_redblack (T Black tl k tr) c (S n).
```

The proposition is_redblack t c n means that t is a well-formed Red-Black tree, in color-context c, with black-height n. Color-context c means that the tree can be part of a well-formed Red-Black tree whose parent node has color c. Color-context Black accommodates any well-formed tree, but a Red context requires a Black root. Black-height n means that the number of Black nodes on any path from the root to a leaf is exactly n.

A well-formed Red-Black tree, in this definition, is not necessarily a search tree. We say that a valid tree is both a search tree and a Red-Black tree.

```
Definition valid (x) := searchtree None None x ∧ ∃n, is_redblack x Red n.
```

Most presentations of Red-Black trees are in an imperative setting: the *insert* function adds a new node to replace some leaf (by overwriting a NULL pointer with the pointer to a new node), then rearranges pointers in place until the Red-Black balance conditions are achieved. In a functional programming language where pointers are not to be updated in place, one wants something more like the unbal_ins function, except with balancing.

I follow Okasaki's presentation of Red-Black trees in a functional setting (Okasaki, 1999).

```
Definition balance color t1 k t2 :=
 match color with
 | Red ⇒ T Red t1 k t2
 | Black ⇒ match t1, t2 with
            | T Red (T Red a x b) y c, d ⇒ T Red (T Black a x b) y (T Black c k d)
            | T Red a x (T Red b y c), d ⇒ T Red (T Black a x b) y (T Black c k d)
            | a, T Red (T Red b y c) z d ⇒ T Red (T Black a k b) y (T Black c z d)
            | a, T Red b y (T Red c z d) ⇒ T Red (T Black a k b) y (T Black c z d)
            | _, _ ⇒ T Black t1 k t2
            end
 end.
```

```
Fixpoint ins x s :=
 match s with
 | E ⇒ T Red E x E
 | T c a y b ⇒ match K.compare x y with
                 | Lt ⇒ balance c (ins x a) y b
                 | Eq ⇒ T c a x b
                 | Gt ⇒ balance c a y (ins x b)
                 end
 end.

Definition makeBlack t :=
  match t with
  | E ⇒ E
  | T _ a x b ⇒ T Black a x b
  end.

Definition insert x s := makeBlack (ins x s).
```

These four functions are the direct translation of Okasaki's ML implementation into Gallina. Okasaki's proof is by appeal to diagrams, with the sentence, "It is routine to verify that the Red-Black balance invariants both hold for the resulting tree."

According to Webster's dictionary, *routine* can mean "monotonous or tedious" or "a sequence of instructions for performing a task that forms a program." Okasaki was right in both senses. It is tedious to prove the correctness of balance *by hand* by applying standard tactics in Coq; instead, I write *a program* in the Ltac language to prove it. I illustrate with just the proof of theorem searchtree_balance, that if T c s k t is a search tree, then balance c s k t is a search tree.

```
Ltac inv H := inversion H; clear H; subst.
Ltac do_searchtree :=
  assumption ||
  constructor ||
  match goal with
  | ⊢ searchtree _ _ (match ?C with Red ⇒ _ | Black ⇒ _ end) ⇒ destruct C
  | ⊢ searchtree _ _ (match ?C with E ⇒ _ | T _ _ _ _ ⇒ _ end) ⇒ destruct C
  | H: searchtree _ _ E ⊢ _ ⇒ inv H
  | H: searchtree _ _ (T _ _ _ _) ⊢ _ ⇒ inv H
  | ⊢ ltopt _ _ ⇒ unfold ltopt in ∗; auto
  | ⊢ match ?A with Some _ ⇒ _ | None ⇒ _ end ⇒ destruct A
  | H: K.lt ?A ?B ⊢ K.lt ?A ?C ⇒ try solve [apply lt_trans with B; assumption]; clear H
  end.

Lemma searchtree_balance:
 ∀c s1 t s2 lo hi,
   ltopt lo (Some t) → ltopt (Some t) hi →
   searchtree lo (Some t) s1→  searchtree (Some t) hi s2 →
    searchtree lo hi (balance c s1 t s2).
Proof. intros. unfold balance. repeat do_searchtree. Qed.
```

The "proof" of the theorem is this: Each subgoal may be solved by either

1. it's trivially true (quod erat demonstrandum is a current hypothesis);
2. apply a constructor of the inductive searchtree predicate;

3. if there is a hypothesis of a certain form, do case analysis the color C (Red or Black);
4. if there is a hypothesis of a certain form, do case analysis on whether a variable C is a leaf E or a nonleaf (T _ _ _ _);
5. invert a hypothesis searchtree _ _ E into its one component assumption;
6. invert a hypothesis searchtree _ _ (T _ _ _ _) into its three component assumptions;
7. unfold the definition of ltopt;
8. if the proof goal is case analysis on an option(key), do case-splitting;
9. try transitivity of less-than.

This program called do_searchtree and, as shown, constructs a proof: exactly 1125 repetitions of do_searchtree builds the proof term. The proof term, not shown, is huge, of course. So, Okasaki is right: the tactics used in do_searchtree are quite routine, and that's all it takes to prove the theorem.

A tree is "nearly Red-Black" if it is nonempty and would be Red-Black if only the root node were colored Black.

```
Inductive nearly_redblack : tree → nat → Prop :=
| nrRB_r: ∀tl k tr n,
        is_redblack tl Black n → is_redblack tr Black n → nearly_redblack (T Red tl k tr) n
| nrRB_b: ∀tl k tr n,
        is_redblack tl Black n → is_redblack tr Black n → nearly_redblack (T Black tl k tr) (S n).

Lemma ins_is_redblack:
  ∀x s n,
    (is_redblack s Black n → nearly_redblack (ins x s) n) ∧
    (is_redblack s Red n → is_redblack (ins x s) Black n).
Proof. ... Qed.

Lemma is_redblack_Black_to_Red:
            ∀s n, is_redblack s Black n → ∃n, is_redblack (makeBlack s) Red n.
Proof. intros; inv H; repeat econstructor; eauto. Qed.

Lemma insert_is_redblack: ∀x s n, is_redblack s Red n → ∃n', is_redblack (insert x s) Red n'.
Proof. intros. unfold insert. destruct (ins_is_redblack x s n).
      apply is_redblack_Black_to_Red with n; auto.
Qed.
```

The theorem that insert preserves the Red-Black balance properties is also "routine;" the ellipsis in the proof of ins_is_redblack conceals some Ltac hacking that's quite similar to do_searchtree.

Finally, we prove that insert is actually correct. That is,

```
Lemma interp_balance: ∀c tl k tr y, interp (balance c tl k tr) y ↔interp (T c tl k tr) y.
Proof. destruct c, tl, tr; unfold balance; intuition; repeat do_interp_balance. Qed.

Lemma interp_insert:
      ∀x y s, searchtree None None s → ((K.eq x y ∨ interp s x) ↔interp (insert y s) x).
```

The proof is "routine:" an easy automated case-analysis, implemented by an Ltac much like the do_searchtree shown above, does most of the work.

Left-leaning Red-Black trees. Sedgewick (Sedgewick, 2008) proposed *left-leaning Red-Black trees*, a data structure identical to ordinary Red-Black trees but with the extra constraint that no node has a red left child. This reduces the number of cases to be handled, either in the (imperative, pointer-swizzling) implementation of the algorithm or the proofs of correctness and balance.

In addition, Sedgewick shows how to factor the implementation of rebalancing Red-Black trees into three operations, rotateLeft, rotateRight, and colorFlip; the proofs can be refactored correspondingly.

My student Max Rosmarin (Rosmarin, 2011) studied the question of whether using the left-leaning invariant would mix well with the Okasaki-style functional program, so as to factor the implementations and proofs. Rosmarin demonstrated that Okasaki's balance function can be factored into Sedgewick's three operations. Although it is not conceptually more complex, the factored function has more lines of code. Recall that Okasaki's function, as I presented it here, has only 10 lines, which is hard to improve on.

The proofs can be factored as well. Recall that my proofs about Okasaki's balance function took 1125 steps. Undoubtedly, proofs factored in left-leaning style would take fewer steps. But my 1125 steps were computed automatically from the 8 one-line proof tactics outlined in **Ltac** do_balance. In that sense, my proof is "routine." Rosmarin found that left-leaning factored proofs were not as "routine," and therefore required more human effort to build.

5 Deletion

It is well known that deletion from Red-Black trees is messier and more difficult both to implement and to prove correct than insertion. Most authors leave it out of their papers. Kahrs (Kahrs, 2001) extends Okasaki's functional Red-Black trees with deletion, and shows an all-too-clever correctness proof miraculously embedded into the type-checking of the Haskell program, as a GADT (Generalized Abstract Data Type). I say all-too-clever because I cannot understand it. I prefer to specify and prove correctness properties in a general-purpose logic meant for that purpose, such as the Calculus of Inductive Constructions (i.e., Coq).

However, Kahrs does explain in English the invariants for deletion. So I was able to take the Kahrs functional-redblack-deletion algorithm and use his invariants to prove it correct in Coq. I use the same kind of Ltac proof automation.

But the delete algorithm is bigger than for insert, and so are the proofs. Here I will just show the Kahrs's invariant for his del function, translated into Coq:

```
Inductive infrared : tree → nat → Prop :=
| infrared_e: infrared E 0
| infrared_r: ∀tl k tr n,
    is_redblack tl Black n → is_redblack tr Black n → infrared (T Red tl k tr) n
| infrared_b: ∀tl k tr n,
    is_redblack tl Black n → is_redblack tr Black n → infrared (T Black tl k tr) (S n).

Definition is_red_or_empty t := match t with T Black _ _ _ ⇒ False | _ ⇒ True end.
Definition is_black t := match t with T Black _ _ _ ⇒ True | _ ⇒ False end.
```

Lemma del_shape:
  ∀x t, (∀ n, is_redblack t Red (S n) → is_black t → infrared (del x t) n) ∧
      (∀ n, is_redblack t Black n → is_red_or_empty t → is_redblack (del x t) Black n).

Rosmarin (Rosmarin, 2011) also studied delete, comparing my implementation and proofs (following Kahrs) with the left-leaning case, and his preliminary results showed that left-leaning delete may in fact be *harder* to reason about than Kahrs-style.

Delete-min. The MSets interface in the Coq Library has an operation min_elt(s) that returns the minimum element of a set s. This allows the use of MSets, such as Red-Black trees, as priority queues in which each operation (insert and delete-min) can be done in $O(\log N)$ time. But there is no delete_min(s) in the interface, which means that delete_min must be constructed from min_elt and delete. Although this is still $O(\log N)$, the constant factor is quite high for two reasons: the tree must be traversed twice, and the delete traversal does comparisons.

By contrast, a straightforward delete_min operation keeps moving leftward in the tree without doing any comparisons, and is therefore much faster. However, on the way back up, it must rebalance the tree much as delete does, and in fact we can re-use much of delete's rebalancing implementation.

I do not prove directly that delete_min preserves the search-tree property, preserves the Red-Black property, and returns the correct result. Instead I prove that delete−min produces the identical key-value and tree to a combination of min_elt and delete—from which, these properties follow as a corollary.

6 Union, intersection, difference

Binary search trees are often used to implement general "set" abstract-data types, where the operations are not limited to insert, lookup, and delete: often the clients want set-union, intersection, and set-difference as well. Search trees are not ideally suited to these operations, but that does not stop the clients from wanting them. So we do the best we can.

Let s and t be binary search trees with cardinalities $|s|$ and $|t|$. To compute $s \cup t$ we can either:

- Insert each element of s into t, in time $O(|s| \log |t|)$ if $|s| \leq |t|$.
- Insert each element of t into s, in time $O(|t| \log |s|)$ if $|t| \leq |s|$.
- Flatten s and t into sorted lists, merge the lists, then reconstruct the sorted list into a tree, all in time $O(|s| + |t|)$.

If $|s|$ is similar to $|t|$, then the linear-time method is faster; otherwise the $|s| \log |t|$ or $|t| \log |s|$ algorithm is best.

The log-linear method just calls upon the insert function, and is easy to prove correct. Flattening a tree into a sorted list is a simple recursive tree-walk and is easy to implement and prove correct.

Building trees from sorted lists To implement linear-time set-union (or intersection, or difference), we need linear-time construction of a Red-Black tree from a sorted list.

We don't want to simply insert each element, as that would take $N \log N$ time. Instead we construct the tree directly, using a pair of mutually recursive functions. But to do this, we need to know in advance the size of the tree.

Algorithms to build balanced trees from sorted lists are certainly not new (Hinze, 1999), but my algorithm takes particular advantage of Coq's inductive construction of the positive integers (the positive datatype) to guide its tree-construction. Recall:

```
Inductive positive :=   xI : positive → positive |  xO : positive → positive |  xH : positive.
```

where xH$= 1$, xO$(n) = 2n$, xI$(n) = 2n+1$. In the Coq library there is a function Psucc that computes successor on positive by the usual ripple-carry method. Therefore, the function

```
Fixpoint poslength {A} (l: list A) := match l with nil ⇒ xH | _::tl ⇒ Psucc (poslength tl) end.
```

in linear time can compute the length of a list, plus one. It's not completely obvious that this takes linear time, since Psucc can take $\log N$ time in the worst case, but in fact the *average* case for ripple carry is constant time.

To turn a list l of length $N-1$ into a Red-Black tree, we execute treeify_g (poslength l) l, which calls upon the following pair of recursive functions:

```
Definition bogus : tree ∗ list key := (E, nil).

Fixpoint treeify_f (n: positive) (l: list key) : tree ∗ list key:=
 match n with
 | xH ⇒ match l with x::l1 ⇒ (T Red E x E, l1) | _ ⇒ bogus end
 | xO n' ⇒ match treeify_f n' l with
                  | (t1, x::l2) ⇒ let (t2,l3) := treeify_g n' l2 in (T Black t1 x t2, l3)
                  | _ ⇒ bogus
                  end
 | xI n' ⇒ match treeify_f n' l with
                  | (t1, x::l2) ⇒ let (t2,l3) := treeify_f n' l2 in (T Black t1 x t2, l3)
                  | _ ⇒ bogus
                  end
 end
 with treeify_g (n: positive) (l: list key) : tree ∗ list key :=
 match n with
 | xH ⇒ (E,l)
 | xO n' ⇒ match treeify_g n' l with
                  | (t1, x::l2) ⇒ let (t2,l3) := treeify_g n' l2 in (T Black t1 x t2, l3)
                  | _ ⇒ bogus
                  end
 | xI n' ⇒ match treeify_f n' l with
                  | (t1, x::l2) ⇒ let (t2,l3) := treeify_g n' l2 in (T Black t1 x t2, l3)
                  | _ ⇒ bogus
                  end
 end.

Definition treeify (l: list key) : tree := fst (treeify_g (poslength l) l).
```

The basic idea is this: To treeify a sorted list of length $2n+1$, first treeify the first part, of length n, yielding subtree t1; then grab the next element k of the list; then treeify the last part of length n, yielding subtree t2; finally construct the node T ? t1 k t2. But what color should go in place of the question-mark, and what if the length is not exactly $2n+1$?

We will place all Red nodes at the bottom. That is, there will be an exactly balanced binary tree of Black nodes; the leaves of this black tree will have children that are either Red or E.

The function treeify_f n l takes a sorted list l of at least n nodes. It consumes n nodes from the list, and builds them into a Red-Black tree t of black-height $n-1$. It returns the pair (t, l') where l' is the rest of the list beyond the nth element.

The function treeify_g n l takes a sorted list l of at least $n-1$ nodes. It consumes $n-1$ nodes from the list, and builds them into a Red-Black tree t of black-height $\lfloor \log_2(n-1) \rfloor$.

For each function, the case $n = 1$ is easy. treeify_f xH l grabs the first element x of l and constructs the tree T Red E x E, whose black-height is 0. treeify_g xH l simply returns the tree E, whose black-height is also 0.

For the case $n = 2n'$, treeify_f (xO n') l builds two subtrees by calling treeify_f and treeify_g; that is, it consumes $n' + 1 + (n' - 1) = n$ nodes from the list.

For the case $n = 2n'$, treeify_g (xO n') l builds two subtrees by calling treeify_g and treeify_g; that is, it consumes $(n' - 1) + 1 + (n' - 1) = n - 1$ nodes from the list.

For the case $n = 2n' + 1$, treeify_f (xI n') l builds two subtrees by calling treeify_f and treeify_f; that is, it consumes $n' + 1 + n' = n$ nodes from the list.

For the case $n = 2n' + 1$, treeify_g (xI n') l builds two subtrees by calling treeify_f and treeify_g; that is, it consumes $(n' - 1) + 1 + n' = n - 1$ nodes from the list.

The proofs are straightforward, except for one thing: Coq will generate an induction scheme for these two mutually recursive functions with 11 cases in the induction. There are the 6 "good" cases (described verbally above), and 5 "bogus" cases, in which the bogus value is returned. Of course the bogus cases will never occur, provided that length(l)>n (for treeify_g n l), or length(l)≥n (for treeify_f n l).

So, before proving the main theorems about treeify, we prove two preliminary lemmas about lengths of lists (using the horrible 11-case induction scheme), and then we use these to prove a specialized 6-case induction lemma (Figure 1).

Although treefy_g_induc looks scary, it's straightforward to use in practice. Remember that every premise in each of the 6 clauses makes it *easier* to use this induction scheme, not harder. In proving a lemma such as,

```
Lemma treeify'_g_is_redblack:
  ∀n l, length l >= nat_of_P n → is_redblack (fst (treeify_g n l)) Red (plog2 n).
```

each of the 6 cases takes just a few lines of proof-script, and the entire proof is 48 lines of Coq.

Using the treeify function (and its proofs), it is simple to implement linear_union, a linear-time set-union algorithm for Red-Black trees. Set intersection and set difference are similar, and use the same treeify function.

Dynamically choosing between the implementations. To measure whether $|s| \gg |t|$ or $|t| \gg |s|$ or neither, one does not want to compute $|s|$, which takes linear time. But we can cheaply compute the approximate log of $|s|$, that is, the black-node height of the tree (since the black-node depth of every leaf is the same). Even more cheaply, we can test whether the black-height of s is at least twice the black-height of t, or vice-versa.

Figure 1. Induction scheme for treeify

```
Lemma treeify_f_length:
  ∀n l, length l > nat_of_P n → length(snd(treeify_f n l))+nat_of_P n = length l.

Lemma treeify_g_length:
  ∀n l, length l ≥ nat_of_P n → length(snd(treeify_g n l))+nat_of_P n = S (length l).

Lemma treeify_g_induc:
        ∀fP gP : positive → list key → tree * list key → Prop,
(*1*) (∀ l n' t1 x l2 t2 l3,
        length l >= nat_of_P n' → length l2 >= nat_of_P n' → fP n' l (t1, x::l2) →
        treeify_f n' l = (t1, x::l2) → fP n' l2 (t2, l3) → treeify_f n' l2 = (t2, l3) →
        fP (xI n') l (T Black t1 x t2, l3)) →
(*2*) (∀ l n' t1 x l2 t2 l3,
        length l >= nat_of_P n' → S(length l2) >= nat_of_P n' → fP n' l (t1, x :: l2) →
        treeify_f n' l = (t1, x :: l2) → gP n' l2 (t2, l3) → treeify_g n' l2 = (t2, l3) →
        fP (xO n') l (T Black t1 x t2, l3)) →
(*3*) (∀ x l1, fP xH (x :: l1) (T Red E x E, l1)) →
(*4*) (∀ l n' t1 x l2 t2 l3,
        length l >= nat_of_P n' → S(length l2) >= nat_of_P n' → fP n' l (t1, x :: l2) →
        treeify_f n' l = (t1, x :: l2) → gP n' l2 (t2, l3) → treeify_g n' l2 = (t2, l3) →
        gP (xI n') l (T Black t1 x t2, l3)) →
(*5*) (∀ l n' t1 x l2 t2 l3,
        S(length l) >= nat_of_P n' → S(length l2) >= nat_of_P n' → gP n' l (t1, x :: l2) →
        treeify_g n' l = (t1, x :: l2) → gP n' l2 (t2, l3) → treeify_g n' l2 = (t2, l3) →
        gP (xO n') l (T Black t1 x t2, l3)) →
(*6*) (∀ l, gP xH l (E, l)) →
(* conclusion *) ∀n l, length l >= nat_of_P n → gP n l (treeify_g n l).
```

```
Definition skip_red t := match t with T Red t' _ _ ⇒ t' | _ ⇒ t end.
Definition skip_black t := match skip_red t with T Black t' _ _ ⇒ t' | t' ⇒ t' end.

Fixpoint compare_height (sx s t tx: tree) : comparison :=
 match skip_red sx, skip_red s, skip_red t, skip_red tx with
 | T _ sx' _ _, T _ s' _ _, T _ t' _ _, T _ tx' _ _ ⇒ compare_height (skip_black tx') s' t' (skip_black tx')
 | _, E, _, T _ _ _ _ ⇒ Lt
 | T _ _ _ _, _, E, _ ⇒ Gt
 | T _ sx' _ _, T _ s' _ _, T _ t' _ _, E ⇒ compare_height (skip_black sx') s' t' E
 | E, T _ s' _ _, T _ t' _ _, T _ tx' _ _ ⇒ compare_height E s' t' (skip_black tx')
 | _, _, _, _ ⇒ Eq
 end.
```

The calculation compare_height s s t t starts the pointers sx,tx racing down the two trees at double-speed, and the pointers s,t walking down at normal speed. Depending on which of these four pointers bottoms out first, we can say informally that

$$c_1 \log|s| < \frac{1}{2}\log|t|, \qquad c_2 \log|s| < \log|t| \wedge \log|s| > c_2 \log|t|, \quad \textit{or} \quad \frac{1}{2}\log|s| > c_3 \log|t|$$

for various constants c_i close to 1.

We do not have to prove this formally! The compare_height function will be used only to select between three different proved-correct implementations of set-union. If we get it wrong, the algorithm will still be formally verified as functionally correct, but it may be inefficient. For efficiency we are relying on a combination of formal proofs about balance properties, plus informal proofs about efficiency. The informal proof is simple.

Theorem: compare_height is correct.

Proof: Obviously it's correct. ▮

Then we combine the three versions of set-union, as follows:

```
Definition union (s t: tree) : tree :=
  match compare_height s s t t with
     | Lt ⇒ fold insert s t
     | Gt ⇒ fold insert t s
     | Eq ⇒ linear_union s t
   end.
```

where fold is a function such that (for example),

fold insert s t = insert s_1 (insert s_2 (insert s_3 $\ldots$ (insert s_n t) $\ldots$))

where s_i are all the keys in tree s.

7 Performance measurements

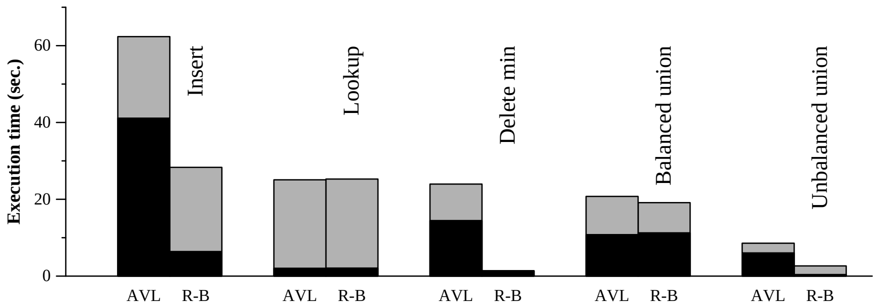

Fig. 1. Performance of AVL vs. Red-black trees. Black bars are actual running time with fast comparisons. Black+grey bars are actual running time with slow comparisons.

Red-black trees run much faster than Letouzey's AVL trees for insert, delete−min, and *unbalanced* union, and run at the same speed for lookup, and *balanced* union. Figure 1 shows measurements of five performance benchmarks:

Insert: Insert 10^6 keys, randomly selected between 1 and 10^6 (with duplication), into an initially empty tree, resulting in a tree t_1 of 631,895 nodes.

Lookup: Look up 10^6 keys, randomly selected between 1 and 10^6, in the tree t_1.

Delete min: Repeatedly delete the minimum element of t_1 until it is empty.

Balanced union: Repeat 10 times, union t_1 with itself, using the linear-time algorithm.

Unbalanced union: Repeat 10^5 times, union a random 10-key tree with t_1.

Benchmarks were compiled by the OCaml compiler and run on an Intel Core 2 Duo E8500 at 3.16GHz with 4GB of RAM.

I show each implementation measured on *fast comparisons*, implemented by two native integer comparisons (the second of which executes with probability $\frac{1}{2}$), and with *slow* *comparisons*, in which a tight loop iterates 100 times before doing the fast comparison. In this way we can measure how much of the balanced-binary tree algorithm is *comparisons* and how much is *overhead*. The comparisons show as a grey bar in the graph, and the overhead as a black bar.

The improvement in *Insert* is explained by the cost of positive arithmetic in the AVL algorithm. *Lookup* shows no improvement, as both of these balanced-binary-tree algorithm ignore the balance conditions during lookup. The improvement in *Delete min* is explained in Section 5. *Balanced union* shows no significant improvement. *Unbalanced union* is faster in my implementation because the AVL implementation uses the linear-time algorithm for this case.

8 Conclusion

Balanced binary search trees are an important data structure, especially for pure functional programming and therefore for verified software. However, several design decisions influence the efficiency of search-tree algorithms. In particular, because in Coq the use of arithmetic usually imposes a $\log N$ penalty, it is advantageous to use search-tree algorithms that avoid arithmetic as they rebalance trees. In addition, for use as priority queues a specialize delete-min operation is much more efficient than separate min-elt and delete; and one can speed up set union or intersection by dynamic choice of algorithm depending on the relative depths of the trees.

Acknowledgments. This research was supported in part by the Air Force Office of Scientific Research (grant FA9550-09-1-0138) and the National Science Foundation (grant CNS-0910448).

References

Armand, Michaël, Grégoire, Benjamin, Spiwack, Arnaud, & Théry, Laurent. (2010). Extending Coq with imperative features and its application to SAT verification. *Pages 83–98 of: Itp'10: International conference on interactive theorem proving*, vol. LNCS 6172. Springer.

Filliâtre, J.-C., & Letouzey, P. (2004). Functors for Proofs and Programs. *Pages 370–384 of: Esop'04: European symposium on programming*, vol. LNCS 2986. Springer.

Hinze, Ralf. 1999 (Sept.). Constructing red-black trees. *Pages 89–99 of:* Okasaki, Chris (ed), *Waaapl'99: Workshop on algorithmic aspects of advanced programming languages*.

Kahrs, Stefan. (2001). Red-black trees with types. *Journal of functional programming*, **11**(04), 425–432.

MacQueen, David B. (1990). A higher-order type system for functional programming. *Pages 353–68 of: Research topics in functional programming*. Reading, MA: Addison-Wesley.

Nanevski, Aleksandar, Morrisett, Greg, Shinnar, Avraham, Govereau, Paul, & Birkedal, Lars. 2008 (Sept.). Ynot: Dependent types for imperative programs. *Icfp '08: Proceedings of the 13th acm sigplan international conference on functional programming*.

Okasaki, Chris. (1999). Red-black trees in a functional setting. *J. functional programming*, **9**(4), 471–477.

Rosmarin, Max. 2011 (Aug.). *Red-black trees in a functional context: Left-leaning and otherwise*. Princeton University Department of Computer Science.

Sedgewick, Robert. (2008). *Left-leaning red-black trees*.