SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

TABLE OF CONTENTS INDEX ROADMAP

REDBLACK

IMPLEMENTATION AND PROOF OF RED-BLACK TREES

Required Reading

(1) General background on red-black trees,

- Section 3.3 of *Algorithms, Fourth Edition*, by Sedgewick and Wayne, Addison Wesley 2011; or
- Chapter 13 of *Introduction to Algorithms, 3rd Edition*, by Cormen, Leiserson, and Rivest, MIT Press 2009
- or Wikipedia.

(2) an explanation of the particular implementation we use here. Red-Black Trees in a Functional Setting, by Chris Okasaki. *Journal of Functional Programming*, 9(4):471-477, July 1999. http://www.westpoint.edu/eecs/SiteAssets/SitePages/Faculty20Documents/Okasaki/jfp99redblack.pdf

(3) Optional reading: Efficient Verified Red-Black Trees, by Andrew W. Appel, September 2011. http://www.cs.princeton.edu/~appel/papers/redblack.pdf

Red-black trees are a form of binary search tree (BST), but with *balance*. Recall that the *depth* of a node in a tree is the distance from the root to that node. The *height* of a tree is the depth of the deepest node. The `insert` or `lookup` function of the BST algorithm (Chapter SearchTree) takes time proportional to the depth of the node that is found (or inserted). To make these functions run fast, we want trees where the worst-case depth (or the average depth) is as small as possible.

In a perfectly balanced tree of N nodes, every node has depth less than or or equal to log N, using logarithms base 2. In an approximately balanced tree, every node has depth less than or equal to 2 log N. That's good enough to make `insert` and `lookup` run in time proportional to log N.

The trick is to mark the nodes Red and Black, and by these marks to know when to locally rebalance the tree. For more explanation and pictures, see the Required Reading above.

We will use the same framework as in Extract.v: keys are Ocaml integers. We don't repeat the `Extract` commands, because they are imported implicitly from Extract.v

```
Require Import Perm.
Require Import Extract.
Require Import Coq.Lists.List.
Export ListNotations.

Definition key := int.
```

```
Inductive color := Red | Black.

Section TREES.
Variable V : Type.
Variable default: V.

 Inductive tree : Type :=
 | E : tree
 | T: color → tree → key → V → tree → tree.

 Definition empty_tree := E.
```

lookup is exactly as in our (unbalanced) search-tree algorithm in Extract.v, except that the `T` constructor carries a `color` component, which we can ignore here.

```
Fixpoint lookup (x: key) (t : tree) : V :=
  match t with
  | E ⇒ default
  | T _ tl k v tr ⇒ if ltb x k then lookup x tl
                         else if ltb k x then lookup x tr
                         else v
  end.
```

The `balance` function is copied directly from Okasaki's paper. Now, the nice thing about machine-checked proof in Coq is that you can prove this correct without actually understanding it! So, do read Okasaki's paper, but don't worry too much about the details of this `balance` function.

In contrast, Sedgewick has proposed *left-leaning red-black trees*, which have a simpler balance function (but a more complicated invariant). He does this in order to make the proof of correctness easier: there are fewer cases in the `balance` function, and therefore fewer cases in the case-analysis of the proof of correctness of `balance`. But as you will see, our proofs about `balance` will have automated case analyses, so we don't care how many cases there are!

```
Definition balance rb t1 k vk t2 :=
 match rb with Red ⇒ T Red t1 k vk t2
 | _ ⇒
 match t1 with
 | T Red (T Red a x vx b) y vy c ⇒
      T Red (T Black a x vx b) y vy (T Black c k vk t2)
 | T Red a x vx (T Red b y vy c) ⇒
      T Red (T Black a x vx b) y vy (T Black c k vk t2)
 | a ⇒ match t2 with
            | T Red (T Red b y vy c) z vz d ⇒
               T Red (T Black t1 k vk b) y vy (T Black c z vz d)
            | T Red b y vy (T Red c z vz d) ⇒
               T Red (T Black t1 k vk b) y vy (T Black c z vz d)
            | _ ⇒ T Black t1 k vk t2
            end
  end
 end.

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

Fixpoint ins x vx s :=
 match s with
 | E ⇒ T Red E x vx E
 | T c a y vy b ⇒ if ltb x y then balance c (ins x vx a) y vy b
                        else if ltb y x then balance c a y vy (ins x vx b)
```

```
                                  else T c a x vx b
    end.

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

Now that the program has been defined, it's time to prove its properties. A red-black tree has two kinds of properties:

- `SearchTree`: the keys in each left subtree are all less than the node's key, and the keys in each right subtree are greater
- `Balanced`: there is the same number of black nodes on any path from the root to each leaf; and there are never two red nodes in a row.

First, we'll treat the `SearchTree` property.

```
Require Import Coq.Logic.FunctionalExtensionality.
Require Import ZArith.
Open Scope Z_scope.
```

Proof Automation for Case-Analysis Proofs.

```
Lemma T_neq_E:
  ∀ c l k v r, T c l k v r ≠ E.
Proof.
intros. intro Hx. inversion Hx.
Qed.
```

Several of the proofs for red-black trees require a big case analysis over all the clauses of the `balance` function. These proofs are very tedious to do "by hand," but are easy to automate.

```
Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

Here we go! Let's just "destruct" on the topmost case. Right, here it's `ltb x k`. We can use `destruct` instead of `bdestruct` because we don't need to remember whether `x<k` or `x≥k`.

```
destruct (ltb x k).
(* The topmost test is match c with..., so just destruct c *)
destruct c.
(* This one is easy. *)
apply T_neq_E.
(* The topmost test is match a1 with..., so just destruct a1 *)
destruct a1.
(* The topmost test is match s2 with..., so just destruct s2 *)
destruct s2.
(* This one is easy by inversion. *)
intro Hx; inversion Hx.
```

How long will this go on? A long time! But it will terminate. Just keep typing. Better yet, let's automate. The following tactic applies whenever the current goal looks like, `match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _`, and what it does in that case is, `destruct c`

```
match goal with
| |- match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
end.
```

The following tactic applies whenever the current goal looks like,

`match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _`,

and what it does in that case is, `destruct s`

```
match goal with
    | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
```

Let's apply that tactic again, and then try it on the subgoals, recursively. Recall that the `repeat` tactical keeps trying the same tactic on subgoals.

```
repeat match goal with
    | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
match goal with
  | |- T _ _ _ _ _ ≠ E ⇒ apply T_neq_E
end.
```

Let's start the proof all over again.

```
Abort.

Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

This is the beginning of the big case analysis. This time, let's combine several tactics together:

```
repeat match goal with
  | |- (if ?x then _ else _) ≠ _ ⇒ destruct x
  | |- match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
  | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
end.
```

What we have left is 117 cases, every one of which can be proved the same way:

```
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
(* Only 111 cases to go... *)
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
apply T_neq_E.
(* Only 107 cases to go... *)
Abort.

Lemma ins_not_E: ∀ x vx s, ins x vx s ≠ E.
Proof.
intros. destruct s; simpl.
apply T_neq_E.
remember (ins x vx s1) as a1.
unfold balance.
```

This is the beginning of the big case analysis. This time, we add one more clause to the `match goal` command:

```
repeat match goal with
  | |- (if ?x then _ else _) ≠ _ ⇒ destruct x
  | |- match ?c with Red ⇒ _ | Black ⇒ _ end ≠ _⇒ destruct c
  | |- match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end ≠ _⇒destruct s
  | |- T _ _ _ _ _ ≠ E ⇒ apply T_neq_E
 end.
Qed.
```

The SearchTree Property

The SearchTree property for red-black trees is exactly the same as for ordinary searchtrees (we just ignore the color `c` of each node).

```
Inductive SearchTree' : Z → tree → Z → Prop :=
| ST_E : ∀ lo hi, lo ≤ hi → SearchTree' lo E hi
| ST_T: ∀ lo c l k v r hi,
    SearchTree' lo l (int2Z k) →
    SearchTree' (int2Z k + 1) r hi →
    SearchTree' lo (T c l k v r) hi.

Inductive SearchTree: tree → Prop :=
| ST_intro: ∀ t lo hi, SearchTree' lo t hi → SearchTree t.
```

Now we prove that if `t` is a SearchTree, then the rebalanced version of `t` is also a SearchTree.

```
Lemma balance_SearchTree:
 ∀ c s1 k kv s2 lo hi,
   SearchTree' lo s1 (int2Z k) →
   SearchTree' (int2Z k + 1) s2 hi →
   SearchTree' lo (balance c s1 k kv s2) hi.
Proof.
intros.
unfold balance.
```

Use proof automation for this case analysis.

```
repeat match goal with
  | |- SearchTree' _ (match ?c with Red ⇒ _ | Black ⇒ _ end) _ ⇒ destruct
c
  | |- SearchTree' _ (match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) _ ⇒
destruct s
  end.
```

58 cases to consider!

```
* constructor; auto.
* constructor; auto.
* constructor; auto.
* constructor; auto.
  constructor; auto. constructor; auto.
  (* To prove this one, we have to do inversion on  the proof goals above the line. *)
  inv H. inv H0. inv H8. inv H9.
  auto.
  constructor; auto.
  inv H. inv H0. inv H8. inv H9. auto.
  inv H. inv H0. inv H8. inv H9. auto.
```

There's a pattern here. Whenever we have a hypothesis above the line that looks like,

- H: SearchTree' _ E _

- H: SearchTree' _ (T _) _

we should invert it. Let's build that idea into our proof automation.

```
Abort.

Lemma balance_SearchTree:
 ∀ c s1 k kv s2 lo hi,
   SearchTree' lo s1 (int2Z k) →
   SearchTree' (int2Z k + 1) s2 hi →
   SearchTree' lo (balance c s1 k kv s2) hi.
Proof.
intros.
unfold balance.
```

Use proof automation for this case analysis.

```
repeat match goal with
  | |- SearchTree' _ (match ?c with Red ⇒ _ | Black ⇒ _ end) _ ⇒
             destruct c
  | |- SearchTree' _ (match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) _ ⇒
             destruct s
  | H: SearchTree' _ E _ |- _ ⇒ inv H
  | H: SearchTree' _ (T _ _ _ _ _) _ |- _ ⇒ inv H
  end.
```

58 cases to consider!

```
* constructor; auto.
* constructor; auto. constructor; auto. constructor; auto.
* constructor; auto. constructor; auto. constructor; auto. constructor;
auto. constructor; auto.
* constructor; auto. constructor; auto. constructor; auto. constructor;
auto. constructor; auto.
* constructor; auto. constructor; auto. constructor; auto. constructor;
auto. constructor; auto.
```

Do we see a pattern here? We can add that to our automation!

```
Abort.

Lemma balance_SearchTree:
 ∀ c s1 k kv s2 lo hi,
   SearchTree' lo s1 (int2Z k) →
   SearchTree' (int2Z k + 1) s2 hi →
   SearchTree' lo (balance c s1 k kv s2) hi.
Proof.
intros.
unfold balance.
```

Use proof automation for this case analysis.

```
repeat match goal with
  | |- SearchTree' _ (match ?c with Red ⇒ _ | Black ⇒ _ end) _ ⇒
              destruct c
  | |- SearchTree' _ (match ?s with E ⇒ _ | T _ _ _ _ _ ⇒ _ end) _ ⇒
              destruct s
  | H: SearchTree' _ E _ |- _ ⇒ inv H
  | H: SearchTree' _ (T _ _ _ _ _) _ |- _ ⇒ inv H
  end;
 repeat (constructor; auto).
Qed.
```

Exercise: 2 stars (ins_SearchTree)

This one is pretty easy, even without proof automation. Copy-paste your proof of insert_SearchTree from Extract.v. You will need to apply `balance_SearchTree` in two places.

```
Lemma ins_SearchTree:
   ∀ x vx s lo hi,
                  lo ≤ int2Z x →
                  int2Z x < hi →
                  SearchTree' lo s hi →
                  SearchTree' lo (ins x vx s) hi.
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (valid)

```
Lemma empty_tree_SearchTree: SearchTree empty_tree.
(* FILL IN HERE *) Admitted.

Lemma SearchTree'_le:
  ∀ lo t hi, SearchTree' lo t hi → lo ≤ hi.
Proof.
induction 1; omega.
Qed.

Lemma expand_range_SearchTree':
  ∀ s lo hi,
   SearchTree' lo s hi →
   ∀ lo' hi',
   lo' ≤ lo → hi ≤ hi' →
   SearchTree' lo' s hi'.
Proof.
induction 1; intros.
constructor.
omega.
constructor.
apply IHSearchTree'1; omega.
apply IHSearchTree'2; omega.
Qed.

Lemma insert_SearchTree: ∀ x vx s,
    SearchTree s → SearchTree (insert x vx s).
(* FILL IN HERE *) Admitted.
```

☐

```
Import IntMaps.

Definition combine {A} (pivot: Z) (m1 m2: total_map A) : total_map A :=
  fun x ⇒ if Z.ltb x pivot then m1 x else m2 x.

Inductive Abs: tree → total_map V → Prop :=
| Abs_E: Abs E (t_empty default)
| Abs_T: ∀ a b c l k vk r,
      Abs l a →
      Abs r b →
      Abs (T c l k vk r) (t_update (combine (int2Z k) a b) (int2Z k) vk).

Theorem empty_tree_relate: Abs empty_tree (t_empty default).
Proof.
constructor.
Qed.
```

Exercise: 3 stars (lookup_relate)

```
Theorem lookup_relate:
  ∀ k t cts , Abs t cts → lookup k t = cts (int2Z k).
Proof. (* Copy your proof from Extract.v, and adapt it. *)
(* FILL IN HERE *) Admitted.
```

☐

```
Lemma Abs_helper:
  ∀ m' t m, Abs t m' → m' = m → Abs t m.
Proof.
   intros. subst. auto.
Qed.

Ltac contents_equivalent_prover :=
 extensionality x; unfold t_update, combine, t_empty;
 repeat match goal with
  | |- context [if ?A then _ else _] ⇒ bdestruct A
 end;
 auto;
 omega.
```

Exercise: 4 stars (balance_relate)

You will need proof automation for this one. Study the methods used in `ins_not_E` and `balance_SearchTree`, and try them here. Add one clause at a time to your `match goal`.

```
Theorem balance_relate:
  ∀ c l k vk r m,
    SearchTree (T c l k vk r) →
    Abs (T c l k vk r) m →
    Abs (balance c l k vk r) m.
Proof.
intros.
inv H.
unfold balance.
repeat match goal with
 | H: Abs E _ |- _ ⇒ inv H
end.
```

Add these clauses, one at a time, to your `repeat match goal` tactic, and try it out:

- 1. Whenever a clause `H: Abs E _` is above the line, invert it by `inv H`. Take note: with just this one clause, how many subgoals remain?
- 2. Whenever `Abs (T _ _ _ _ _) _` is above the line, invert it. Take note: with just these two clause, how many subgoals remain?
- 3. Whenever `SearchTree' _ E _` is above the line, invert it. Take note after this step and each step: how many subgoals remain?
- 4. Same for `SearchTree' _ (T _ _ _ _ _) _`.
- 5. When `Abs match c with Red ⇒ _ | Black ⇒ _ end _` is below the line, `destruct c`.
- 6. When `Abs match s with E ⇒ _ | T ... ⇒ _ end _` is below the line, `destruct s`.
- 7. Whenever `Abs (T _ _ _ _ _) _` is below the line, prove it by `apply Abs_T`. This won't always work; Sometimes the "cts" in the proof goal does not exactly match the form of the "cts" required by the `Abs_T` constructor. But it's all right if a clause fails; in that case, the `match goal` will just try the next clause. Take note, as usual: how many clauses remain?
- 8. Whenever `Abs E _` is below the line, solve it by `apply Abs_E`.
- 9. Whenever the current proof goal matches a hypothesis above the line, just use it. That is, just add this clause: | |- _ => assumption
- 10. At this point, if all has gone well, you should have exactly 21 subgoals. Each one should be of the form, `Abs (T ...) (t_update...)` What you want to do is replace (t_update...) with a different "contents" that matches the form required by the Abs_T constructor. In the first proof

goal, do this: `eapply Abs_helper`. Notice that you have two subgoals. The first subgoal you can prove by: apply Abs_T. apply Abs_T. apply Abs_E. apply Abs_E. apply Abs_T. eassumption. eassumption. Step through that, one at a time, to see what it's doing. Now, undo those 7 commands, and do this instead: repeat econstructor; eassumption. That solves the subgoal in exactly the same way. Now, wrap this all up, by adding this clause to your `match goal`: | |- _ => eapply Abs_helper; `repeat econstructor; eassumption` |

- 11. You should still have exactly 21 subgoals, each one of the form, `t_update... = t_update...` . Notice above the line you have some assumptions of the form, `H: SearchTree' lo _ hi` . For this equality proof, we'll need to know that `lo ≤ hi`. So, add a clause at the end of your `match goal` to apply SearchTree'_le in any such assumption, when below the line the proof goal is an equality _ = _ .
- 12. Still exactly 21 subgoals. In the first subgoal, try: `contents_equivalent_prover`. That should solve the goal. Look above, at `Ltac contents_equivalent_prover`, to see how it works. Now, add a clause to `match goal` that does this for all the subgoals.
- Qed!

```
(* FILL IN HERE *) Admitted.
```

Extend this list, so that the nth entry shows how many subgoals were remaining after you followed the nth instruction in the list above. Your list should be exactly 13 elements long; there was one subgoal *before* step 1, after all.

```
Definition how_many_subgoals_remaining :=
    [1; 1; 1; 1; 1; 2

  ].
```

☐

Exercise: 3 stars (ins_relate)

```
Theorem ins_relate:
 ∀ k v t cts,
    SearchTree t →
    Abs t cts →
    Abs (ins k v t) (t_update cts (int2Z k) v).
Proof. (* Copy your proof from SearchTree.v, and adapt it.
     No need for fancy proof automation. *)
(* FILL IN HERE *) Admitted.
```

☐

```
Lemma makeBlack_relate:
 ∀ t cts,
    Abs t cts →
    Abs (makeBlack t) cts.
Proof.
intros.
destruct t; simpl; auto.
inv H; constructor; auto.
Qed.

Theorem insert_relate:
 ∀ k v t cts,
    SearchTree t →
    Abs t cts →
    Abs (insert k v t) (t_update cts (int2Z k) v).
Proof.
intros.
unfold insert.
apply makeBlack_relate.
```

```
  apply ins_relate; auto.
Qed.
```

OK, we're almost done! We have proved all these main theorems:

```
Check empty_tree_SearchTree.
Check empty_tree_relate.
Check lookup_relate.
Check insert_SearchTree.
Check insert_relate.
```

Together these imply that this implementation of red-black trees (1) preserves the representation invariant, and (2) respects the abstraction relation.

Exercise: 4 stars, optional (elements)

Prove the correctness of the `elements` function. Because `elements` does not pay attention to colors, and does not rebalance the tree, then its proof should be a simple copy-paste from SearchTree.v, with only minor edits.

```
Fixpoint elements' (s: tree) (base: list (key*V)) : list (key * V) :=
 match s with
 | E ⇒ base
 | T _ a k v b ⇒ elements' a ((k,v) :: elements' b base)
 end.

Definition elements (s: tree) : list (key * V) := elements' s nil.

Definition elements_property (t: tree) (cts: total_map V) : Prop :=
   ∀ k v,
    (In (k,v) (elements t) → cts (int2Z k) = v) ∧
    (cts (int2Z k) ≠ default → In (k, cts (int2Z k)) (elements t)).

Theorem elements_relate:
  ∀ t cts,
  SearchTree t →
  Abs t cts →
  elements_property t cts.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Proving Efficiency

Red-black trees are supposed to be more efficient than ordinary search trees, because they stay balanced. In a perfectly balanced tree, any two leaves have exactly the same depth, or the difference in depth is at most 1. In an approximately balanced tree, no leaf is more than twice as deep as another leaf. Red-black trees are approximately balanced. Consequently, no node is more then 2logN deep, and the run time for insert or lookup is bounded by a constant times 2logN.

We can't prove anything *directly* about the run time, because we don't have a cost model for Coq functions. But we can prove that the trees stay approximately balanced; this tells us important information about their efficiency.

Exercise: 4 stars (is_redblack_properties)

The relation `is_redblack` ensures that there are exactly `n` black nodes in every path from the root to a leaf, and that there are never two red nodes in a row.

```
Inductive is_redblack : tree → color → nat → Prop :=
| IsRB_leaf: ∀ c, is_redblack E c 0
```

```
  | IsRB_r: ∀ tl k kv tr n,
          is_redblack tl Red n →
          is_redblack tr Red n →
          is_redblack (T Red tl k kv tr) Black n
  | IsRB_b: ∀ c tl k kv tr n,
          is_redblack tl Black n →
          is_redblack tr Black n →
          is_redblack (T Black tl k kv tr) c (S n).

Lemma is_redblack_toblack:
  ∀ s n, is_redblack s Red n → is_redblack s Black n.
Proof.
(* FILL IN HERE *) Admitted.

Lemma makeblack_fiddle:
  ∀ s n, is_redblack s Black n →
            ∃ n, is_redblack (makeBlack s) Red n.
Proof.
(* FILL IN HERE *) Admitted.
```

nearly_redblack expresses, "the tree is a red-black tree, except that it's nonempty and it is permitted to have two red nodes in a row at the very root (only)."

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

Lemma ins_is_redblack:
  ∀ x vx s n,
    (is_redblack s Black n → nearly_redblack (ins x vx s) n) ∧
    (is_redblack s Red n → is_redblack (ins x vx s) Black n).
Proof.
induction s; intro n; simpl; split; intros; inv H; repeat constructor; auto.
*
destruct (IHs1 n); clear IHs1.
destruct (IHs2 n); clear IHs2.
specialize (H0 H6).
specialize (H2 H7).
clear H H1.
unfold balance.
```

You will need proof automation, in a similar style to the proofs of ins_not_E and balance_relate.

```
(* FILL IN HERE *) Admitted.

Lemma insert_is_redblack:
  ∀ x xv s n, is_redblack s Red n →
                    ∃ n', is_redblack (insert x xv s) Red n'.
Proof.
  (* Just apply a couple of lemmas:
     ins_is_redblack and makeblack_fiddle *)
(* FILL IN HERE *) Admitted.
```

□

```
End TREES.
```

Extracting and Measuring Red-Black Trees

```
Extraction "redblack.ml" empty_tree insert lookup elements.
```

You can run this inside the ocaml top level by:

```
#use "redblack.ml";;
#use "test_searchtree.ml";;
run_tests();;
```

On my machine, in the byte-code interpreter this prints,

```
Insert and lookup 1000000 random integers in 0.889 seconds.
Insert and lookup 20000 random integers in 0.016 seconds.
Insert and lookup 20000 consecutive integers in 0.015 seconds.
```

You can compile and run this with the ocaml native-code compiler by:

```
ocamlopt redblack.mli redblack.ml -open Redblack test_searchtree.ml -o test_redblack
./test_redblack
```

On my machine this prints,

```
Insert and lookup 1000000 random integers in 0.436 seconds.
Insert and lookup 20000 random integers in 0. seconds.
Insert and lookup 20000 consecutive integers in 0. seconds.
```

Success!

The benchmark measurements above (and in Extract.v) demonstrate that:

- On random insertions, red-black trees are slightly faster than ordinary BSTs (red-black 0.436 seconds, vs ordinary 0.468 seconds)
- On consecutive insertions, red-black trees are *much* faster than ordinary BSTs (red-black 0. seconds, vs ordinary 0.374 seconds)

In particular, red-black trees are almost exactly as fast on the consecutive insertions (0.015 seconds) as on the random (0.016 seconds).