15-150 Fall 2013
Lecture 18

Stephen Brookes *

Tuesday, 29 October

1 Topics

- Case study: implementing an abstract type of dictionaries.
- Information hiding, abstract types, and representation invariants.
- Localized reasoning, as an advantage of abstract types.
- A provably efficient, provably correct abstract type.

2 Outline

We begin by presenting a signature for an abstract type of dictionaries, with operations for inserting items (tagged with a key), and looking up the item associated with a key. When keys belong to an ordered type, we can implement dictionaries as binary search trees, whose structure exploits the ordering to support efficient lookup (usually faster than the naive list representation would allow). We can build binary search trees in the same way, regardless of the keys and the items being inserted. We take advantage of polymorphism and also we use functors to encapsulate the bst construction.

The binary search tree implementation still suffers from poor worst-case behavior, so this motivates us to explore a more sophisticated tree-based implementation (red-black trees). We then relate the two kinds of trees.

*Building on notes by Dan Licata.

3 Binary search trees

We define a parameterized type of trees with data at the nodes. If we use data of the form (k, v), where k is a "key" value belonging to some type equipped with a comparison function, we can use trees to represent "dictionaries" and design insertion and lookup functions that use key comparison for efficiency.

To do this in a modular manner, we begin with two signatures:

- `ORDERED`, an interface providing a type named `t` and a "comparison" function of type `t * t -> order`.
- `DICT`, an interface with a parameterized type `'a dict`, a structure `Key` with signature `ORDERED`, a value `empty` and functions `insert` and `lookup`. We also include a function `trav` that's useful for debugging.

```
signature ORDERED =
 sig
   type t
   val compare : t * t -> order
 end;

signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict -> Key.t * 'a -> 'a dict
    val lookup : 'a dict -> Key.t -> 'a option
    val trav : 'a dict -> (Key.t * 'a) list
  end;
```

Note: it's OK to have a signature containing a structure! But that will mean that when we implement this signature, we'll be building a structure containing another structure. Again, not a problem in the ML language.

Also note: we decided to curry the function types in the `DICT` interface. When we implement, we'll need to be careful to curry in the right places, too!

It's easy to build a structure that implements `ORDERED`. For example:

```
structure Integers : ORDERED =
 struct
    type t = int;
    fun compare(x, y) =
       if x<y then LESS else
       if y<x then GREATER else EQUAL
  end;
```

One way to implement `DICT` is to use integers as keys, and lists of integer-entry values as dictionaries:

```
structure Assoc : DICT =
 struct
   structure Key = Integers
   type 'a dict = (Key.t * 'a) list
   val empty = [ ]
   fun insert D (k, v) = ...
   fun lookup D k =
 end;
```

Incidentally, inside the body of this structure, `Key` is bound to the structure `Integers`, so `Key.t` is the type `Integers.t`, which is `int`. So we could just as well have written:

```
structure Assoc : DICT =
 struct
   structure Key = Integers
   type 'a dict = (int * 'a) list
   val empty = [ ]
   fun insert D (k, v) = ...
   fun lookup D k =
 end;
```

The body of `Assoc` can easily be completed to obtain a structure that implements `DICT`, in which dictionaries (values of type `entry list` for some type `entry`) are represented as lists of integer-entry pairs, and insertion and lookup use the integers as keys.

We could also give a structure that implements dictionaries using binary search trees of integer-entry pairs. But again there would be a common design pattern: binary search trees over an ordered type of keys. Instead, let's introduce a functor!

```
functor BSTDict(Key : ORDERED) : DICT =
 struct
   structure Key : ORDERED = Key;
   datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree;
   type 'a dict = 'a tree;

   (* empty : 'a dict *)
   val empty = Leaf;

  (* lookup : 'a dict -> Key.t -> 'a option *)
  fun lookup Leaf k = NONE
    | lookup (Node (D1, (k', v'), D2)) k  =
         case Key.compare (k,k') of
             EQUAL    =>  SOME v'
           | LESS     =>  lookup D1 k
           | GREATER  =>  lookup D2 k;

  (* insert : 'a dict -> key * 'a -> 'a dict *)
  fun insert Leaf (k, v) = Node(Leaf, (k, v), Leaf)
   |  insert (Node(l, (k', v'), r)) (k, v) =
        case Key.compare(k, k') of
                EQUAL   => Node(l, (k,v), r)
         |      LESS    => Node(insert l (k,v), (k',v'), r)
         |      GREATER => Node(l, (k',v'), insert r (k,v));

    fun trav Leaf = [ ]
     |  trav (Node(l, (k,v), r)) = (trav l) @ (k,v) :: (trav r)
end;
```

Using this functor

```
structure S = BSTDict(Integers);

open S;

fun build [ ] = empty
 |  build (x::L) = insert (build L) (x, Int.toString x);

fun flatten Ls = foldr (op @) [ ] Ls;

fun splits [ ] = [([ ], [ ])]
 |  splits (x::r) =
       ([ ], x::r) :: (map (fn (l1,l2) => (x::l1,l2)) (splits r));

fun perms [ ] = [[ ]]
 |  perms (a::l) =
    let
      val t = flatten (map splits (perms l))
    in
      map (fn (l1, l2) => l1@[a]@l2) t
    end;

fun forall p [ ] = true
 |   forall p (x::L) = (p x) andalso forall p L;

 val true = forall bst (map build (perms [1,2,3,4,5,6,7,8]));
```

Here `bst` is the function we defined earlier for checking if a value of tree type is actually a binary search tree, adapted to use the appropriate comparison.

Here it is, adapted:

```
 fun sorted [ ] = true
  |  sorted ((k,_)::L) = forall (fn (k',_) => Key.compare(k,k') <> GREATER) L;

 fun bst T = sorted (trav T)
```

The above code fragment generates the list of all dictionary values expressible as `build L`, where `L` ranges over the list of permutations of `[1,2,3,4,5,6,7,8]`. Then it checks that each of these trees is indeed a binary search tree.

```
 - forall bst (map build (perms [1,2,3,4,5,6,7,8]));
val it = true : bool
```

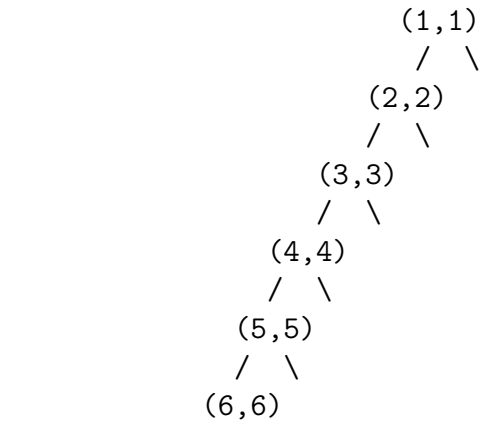
So this looks like our testing is producing satisfactory results: if we insert 8 different things into a dictionary, in any order, we always get a binary search tree.

However, this implementation gives no guarantees about balance!

For example:

```
- val T = build [1,2,3,4,5,6];
val T = Node (Node (Node #,(#,#),Leaf),(6,"6"),Leaf) : string tree
```

In pictorial form, this tree `T` looks like



Not very balanced! Lookup in an unbalanced tree is going to have worst-case runtime approximately the length of the longest path. Clearly we'd prefer to deal with balanced trees. But it's going to be expensive to revise the insertion function so that it preserves balance. (Try it!) Next we'll explore a way to implement `DICT` that yields better asymptotic runtime, by guaranteeing a form of balance.

For a given list of key-value pairs, there are generally many different binary search trees containing the data from this list. Some are poorly balanced, and some are better balanced. The problem with unbalanced trees is

that in the extreme case (when all the data is strung out on a single path) lookups and inserts take time linear in the number of items in the dictionary. With a balanced tree we'd expect logarithmic time lookups and inserts.

It would be hard to implement insertion in such a way that we always obtain a perfectly balanced tree (allowing for the fact that the number of items in the tree may be odd or even, and may not be close to a power of 2). In fact it would probably be too expensive to strive for optimal balance, and we won't even try. The problem becomes even more difficult if we extend our implementation to include a deletion operation as well.

Instead, we will examine a well known implementation of *red-black trees*, which guarantees a weaker but useful kind of balance: in a red-black tree the ratio of the length of the longest path to the length of the shortest path is at most 2-to-1. So grossly imbalanced trees like the one we saw last time will never arise! With this implementation, even though not perfectly balanced, lookups and inserts do indeed take logarithmic time.

The main ideas are as follows. We work with binary trees whose data are not just key-entry pairs, but are key-entry pairs tagged with a "color". There are two possible colors (red, and black!).

A binary tree of color-key-value data is a red-black tree iff each node is colored red or black, leaf nodes are black, and:

(1) *Sorted*: The keys are sorted wrt the comparison function;

(2) *Well-black*: Each path from root to a leaf has the same number of black nodes (this is called the black height of the tree);

(3) *Well-red*: No red node has a red child.

If all nodes are black, the well-black property would imply that the tree is perfectly balanced. The red-black properties imply that the longest path from root to leaf is no more than double the length of the shortest path from root to leaf; so a red-black tree with n nodes has height O(log n), so can be searched in logarithmic time. By making an abstract data type of dictionaries built this way, and only providing operations that preserve the red-black invariant, we guarantee that users always obtain balanced trees!

Moreover, the red-black invariant characterized by these three properties is easy to maintain with a slight modification to our insertion operation, as we will soon see.

The key ideas are that:

> *representation invariants can help specify and prove correctness of an implementation, and to analyze efficiency*

and

> *an abstract type supports local reasoning about correctness and efficiency, without worrying about client behavior*

In this example, the red-black invariant is a "representation invariant" that we build into the implementation. Every dictionary constructible by using the visible operations (empty and insert) is guaranteed to satisfy this property. And we can reason "locally" (referring solely to the data and functions inside the structure or the functor body). This is hugely important! And justified, because there's no way any user can break the representation invariant, because of the guarantee above and the scope rules governing the visibility of the types and operations defined in our module.

4 Signatures

We begin with the same two signatures as before:

- `ORDERED`, an interface providing a type named `t` and a "comparison" function of type `t * t -> order`.
- `DICT`, an interface with a parameterized type `'a dict`, a structure `Key` with signature `ORDERED`, a value `empty` and functions `insert` and `lookup`. We also include a function `trav` that's useful for debugging.

And for testing here is a structure that implements `ORDERED`:

```
structure Integers : ORDERED =
 struct
    type t = int;
    fun compare(x, y) =
       if x<y then LESS else
       if y<x then GREATER else EQUAL
  end;
```

5 Binary search trees, again

Before we introduce the red-black-tree implementation, here for contrast is the binary-search-tree implementation from before:

```
functor BSTDict(Key : ORDERED) : DICT =
 struct
   structure Key : ORDERED = Key;
   datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree;
   type 'a dict = 'a tree;

   (* empty : 'a dict *)
   val empty = Leaf;

  (* lookup : 'a dict -> Key.t -> 'a option *)
  fun lookup Leaf k = NONE
    | lookup (Node (D1, (k', v'), D2)) k  =
         case Key.compare (k,k') of
             EQUAL    =>  SOME v'
           | LESS     =>  lookup D1 k
           | GREATER  =>  lookup D2 k;

  (* insert : 'a dict -> key * 'a -> 'a dict *)
  fun insert Leaf (k, v) = Node(Leaf, (k, v), Leaf)
   |  insert (Node(l, (k', v'), r)) (k, v) =
        case Key.compare(k, k') of
                EQUAL   => Node(l, (k,v), r)
         |      LESS    => Node(insert l (k,v), (k',v'), r)
         |      GREATER => Node(l, (k',v'), insert r (k,v));

    fun trav Leaf = [ ]
     |  trav (Node(l, (k,v), r)) = (trav l) @ (k,v) :: (trav r)
end;
```

Recall also that "new" inserts (insertions of the form `insert D (k, v)` in which `k` is not `EQUAL` to any key in `D`) happen at leaf nodes, and that "updates" (insertions where there is an `EQUAL` key already in the dictionary) don't alter the tree structure.

And let's remind ourselves of a simple example that leads to a poorly balanced tree: a sequence of inserts in which the keys are in increasing order.

Draw the trees produced by inserting entries with keys 1,2,3 in that order, starting from an empty dictionary. (We assume here that `Key.t` is `int` and `Key.compare` is the usual comparison for integers.)

6 Red-black trees

To build a better dictionary implementation we'll define a functor

```
RBTDict
```

that can be applied to a structure `Key : ORDERED` to build a structure with signature `DICT`. So we'll need to include in the functor body a definition for the type `'a dict`, definitions for `empty:'a dict`, `insert`, and so on.

Inside the functor body we'll define a datatype `color` for colors:

```
datatype color = Red | Black;
```

and we'll implement `'a dict` as binary trees with color-key-data entries of type `color * (Key.t * 'a)` at the nodes:

```
datatype a tree =
      Leaf | Node of a tree * (color * (Key.t * a)) * a tree;
type a dict = a tree;
```

(Since `Leaf` and `Node` are not in the signature `DICT`, users won't be able to use them.)

We can implement the empty dictionary as before:

```
val empty = Leaf;
```

And the lookup function doesn't need to pay attention to colors, so we can adapt the old function definition in the obvious way:

```
(* lookup : a dict -> Key.t -> a option *)
 fun lookup Leaf k = NONE
   | lookup (Node (D1, (_ , (k, v)), D2)) k  =
        case Key.compare (k,k) of
            EQUAL    =>  SOME v
          | LESS     =>  lookup D1 k
          | GREATER  =>  lookup D2 k;
```

We used an underscore pattern to emphasize the fact that colors at nodes are ignored by the lookup function.

As you should expect, in designing the insertion operation we need to pay close attention to color. We need to design

```
insert : a dict -> Key.t * a -> a dict
```

to ensure that, whenever `D` is a dictionary value satisfying the red-black-tree properties, then for all key-value pairs `(k,v)`, `insert D (k, v)` returns a dictionary value that also satisfies the red-black-tree properties. How should we achieve this?

Let's start by mimicking the original insertion function, but let's rename the function `ins` in the expectation that we're going to have to tweak it later to obtain the final `insert` function:

```
fun ins Leaf (k, v) = Node(Leaf, (k, v), Leaf)
  |  ins (Node(l, (k', v'), r)) (k, v) =
       case Key.compare(k, k') of
              EQUAL   => Node(l, (k,v), r)
        |     LESS    => Node(ins l (k,v), (k',v'), r)
        |     GREATER => Node(l, (k',v'), ins r (k,v));
```

If we adjust to take account of colors, the first issue is: what color to use on the right-hand-side of the first clause? What color should we use for the single-node tree built by inserting `(k,v)` into the empty tree? Remembering that "new" inserts happen at leaf nodes, and that's what we're defining here, it would be a bad idea to choose `Black`: this would mess up the black height on the side of the tree where the insert is going. So, let's use `Red`, since that would obviously give us a tree with the required properties (1), (2) and (3). So our new first clause is going to be:

```
ins Leaf (k, v) = Node(Leaf, (Red, (k, v)), Leaf)
```

The second clause of the function also needs to be adjusted, because the tree argument on the left-hand-side has a color field and we need to specify a color on the right-hand-side. Here's the obvious first attempt, which is also obviously not going to give us good balance but at least has the virtue of being well typed and maintaining sortedness:

```
ins (Node(l, (c, (k', v')), r)) (k, v) =
        case Key.compare(k, k') of
                EQUAL   => Node(l, (c, (k,v)), r)
         |      LESS    => Node(ins l (k,v), (c, (k',v')), r)
         |      GREATER => Node(l, (c, (k',v')), ins r (k,v));
```

This just copies the same color over to the root on the right-hand-side! It's not hard to see that when the key being used to insert is the same as the key at the root (the first case in the `case` expression) the result will be a red-black-tree, since the original tree was a red-black-tree and we return another tree with the same shape and colors. So "updates" work fine, and preserve the representation invariant. But this simple-minded color-propagation cannot work for "new" insertions, because we will be putting a new (and `Red`-colored) node at the leaf of the tree; if the original tree was a proper red-black-tree, there's no reason why this should also produce a red-black-tree. (Because we only insert a red node, the black-height of the tree stays the same. We also preserve sortedness. But the well-red part of the invariant will not hold any more.)

If we insert `(1, "1")` into the red-black-tree

```
    (Black, (4,"4"))
     /             \
(Red, (3, "3"))  (Red, (5, "5"))
   / \                 / \
  .   .               .   .
```

We'll get

```
    (Black, (4, "4"))
    /               \
(Red, (3, "3"))  (Red, (5, "5"))
  /           \        / \
(Red, (1, "1"))   .     .   .
  / \
 .   .
```

which has a red node with a red parent, so isn't a red-black-tree. But *that's okay, as long as we fix it up eventually.* This is the important idea of a *critical section* of code: inside the implementation of a module, you can break the representation invariant in a piece of code, provided you repair the invariant

before returning any values to clients. From the client's viewpoint, `insert` takes a RBT and produces a RBT. But internally, in the process of doing an insert, it's OK to work temporarily with trees that do not satisfy the invariants.[1]

And in this example, there's an obvious easy way to fix it up: just rotate the tree at the root and adjust the colors, to get:

```
      (Red, (3, "3"))
      /              \
(Black, (1, "1")) (Black, (4, "4"))
 /   \               /  \
.     .             .   (Red, (5, "5"))
                          / \
                         .   .
```

This turns out to be a special case of the more general fix that we'll explain shortly.

Similarly, if we naïvely `(1, "1")` into a RBT with a single node, a red root, such as

```
(Red, (3, "3"))
 /      \
.        .
```

we get an almost-well-red tree as a result,

```
  (Red, (3, "3"))
   /            \
(Red, (1, "1"))    .
 /  \
 .  .
```

and again there is an easy fix: blacken the root color, to get

```
(Black, (3, "3"))
 /              \
(Red, (1, "1"))   .
 /  \
 .  .
```

[1] Analogy: the representation invariant is "your room is clean". The clients are your parents. During the semester, your room can be dirty, as long as you clean it before parents' weekend and the end of the year.

(This is obviously a red-black-tree!)

More generally, we need to allow invariant violations of the following kind: an *almost red-black tree (ARBT)* is like a RBT (well-black, and sorted), except that instead of being well-red, it is

> (almost-well-red) no red node has a red child, except possibly the root node

Our `ins` function, as we have seen, may try to build a tree whose left- or right-child is an almost red-black tree; and there are four possible situations that cover all ways for this to happen: left- or right child with the defect; and in each case, either the left or right child of the defective subtree could be the "red child of a red node". We can *rebalance* in such cases, to produce a tree that is well-red and without messing up the black-height or sortedness of the tree. [2]

We can make this precise by defining an ML function

```
balance : 'a tree * (color * (Key.t * 'a)) * 'a tree
            -> 'a tree * (color * (Key.t * 'a) * 'a tree
```

that we can apply before using `Node`.

The 4 possible ways for the shape of a triple

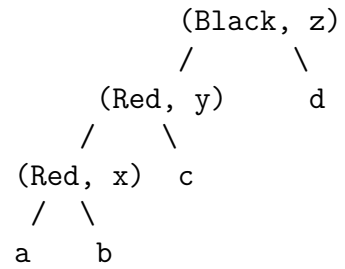
```
(T1, (c, (k,v)), T2)
```

to cause trouble. In each case we can easily define a *pattern* that matches the bad combinations, and in each case there is a simple "rotation" that can be done to solve the problem.

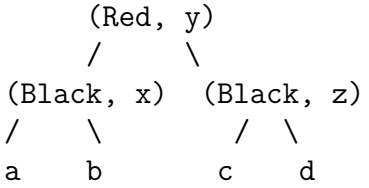
If we are combining one ARBT and one RBT under a black root, there are four possible situations where the RBT invariant is violated: the left tree is an ARBT and the right is an RBT, and the left child of the left is root is a red child of a red node; the left tree is an ARBT and the right is an RBT, and the right child of the left is root is a red child of a red node; and symmetrically, with the right tree an ARBT and the left a RBT. In each of these four cases, we can *rotate* at the root making the root red with two black children, while preserving the sortedness and well-blackness of the tree, *and* ensuring that there are no well-red violations *except possibly at the root.*

[2] This balancing scheme is due to Chris Okasaki (*Red-Black Trees in a Functional Setting*, Journal of Functional Programming, 1999).

Consider the case of a tree with a black root, where the left child is an ARBT, the right is an RBT, the left-left child is a red child of a red node, and the tree is sorted. Let's draw this pictorially as



To balance this picture we rotate at the root and re-color, to get



Why does this work? We need to show that the new tree has the three desired properties:

- Sortedness: because the original tree was assumed to be sorted, the keys in `c` are greater than or equal to the key in `y`, but less than or equal to the key in `z`, so the new tree with `c` as can the left child of `z` will also be sorted.

- Well-red: by assumption, `a, b, c, d` are red-black trees; a black node can have any red-black trees as children, so the trees rooted at `x` and `z` are well-red. Because `x` and `z` are colored black, the overall tree is a red-black tree. (Note that this works even if the root of `c` is also red, which is possible – in an almost red-black tree, both children of a red root might be red.)

- Black-height: By assumption, the original tree has a black-height, which must be positive, say $h + 1$, because the root is black; hence each of `a,b,c,d` has a black-height of h. Thus, the result tree also has a black-height of $h + 1$ because each child of the red root is black.

You might like to consider if there are any other plausible choices of coloring that could have been made here. [3]

The correctness proofs for the other cases of rebalancing are analogous, *by symmetry*.

Notice that the almost red-back trees whose shape resembles the picture shown above will match the following ML pattern:

```
Node (Node(Node(a,(Red,x),b), (Red,y), c), (Black, z), d)
```

where we use variable patterns `a, b, c, d` to match against subtrees, and variable patterns `x,y,z` to match against color-key-value data. And — with these variables bound to pieces of a tree — we can obtain the rotated tree simply by evaluating

```
Node (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z),d))
```

Here is the ML code for balancing, based on this idea. We choose to do balancing "inside" the `Node`. There are four non-trivial cases, as above; in all other cases we leave the components alone.

```
(* balance : a data -> a data *)
  fun balance (Node(Node(a,(Red,x),b), (Red,y), c), (Black, z), d)
            = (Node(a,(Black,x),b),  (Red,y), Node(c,(Black,z),d))
  |   balance (Node(a, (Red,x), Node(b,(Red,y),c)),  (Black,z), d)
            = (Node(a,(Black,x),b), (Red, y), Node(c,(Black,z),d))
  |   balance (a,(Black,x),  Node(Node(b,(Red,y),c), (Red, z), d))
            = (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z), d))
  |   balance (a, (Black,x), Node(b, (Red,y), Node(c,(Red,z), d)))
            = (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z), d))
  |   balance p = p;
```

So now we're trying to work with the following insertion function:

[3] Alternative colorings: We cannot make z red, because the root of d might be red. We could make x red and the root y black, but this would not satisfy the black-height invariant: if the black-height of the input is $h+1$, then the number of black nodes on the path to a leaf in b would be $h+1$, whereas the number of black nodes on a path to a leaf in d would be $h+2$. We could make each of x and y and z black; in this case, the result would have a black-height that is one more than the black-height of the input. However, the call site of rebalancing requires that it preserves the black-height, rather than incrementing it.

```
fun ins Leaf (k, v) = Node(Leaf, (Red, (k, v)), Leaf)
   |  ins (Node(l, (c, (k, v)), r)) (k, v) =
        case Key.compare(k, k) of
                EQUAL   => Node(l, (c, (k,v)), r)
         |      LESS    => Node(balance(ins l (k,v), (c, (k,v)), r))
         |      GREATER => Node(balance(l, (c, (k,v)), ins r (k,v)));
```

But there's one last problem. This doesn't quite guarantee to produce a red-black tree. But it *almost* works. All we need to do to finish the job is to change the color at the root to `Black`. So let's introduce a helper function

```
(* blackenroot : a dict -> a dict *)
fun blackenroot Leaf = Leaf
  | blackenroot (Node(D1, (_, (k,v)), D2)) = Node(D1, (Black, (k,v)), D2);
```

Whenever `D` is an almost red-black tree, `blacken root D` is a red-black-tree, because: making the root black obviously maintains sortedness, removes the only possible well-redness violation, and maintains the fact that the tree has a black-height, even though it potentially increases the black-height by one.

So our finished product, the `insert` function, will use `ins` followed by `blackenroot`:

```
fun ins Leaf (k, v) = Node(Leaf, (Red, (k, v)), Leaf)
   |  ins (Node(l, (c, (k', v')), r)) (k, v) =
        case Key.compare(k, k') of
                EQUAL   => Node(l, (c, (k,v)), r)
         |      LESS    => Node(balance(ins l (k,v), (c, (k',v')), r))
         |      GREATER => Node(balance(l, (c, (k',v')), ins r (k,v)));

(* insert : a dict -> Key.t * a -> a dict *)
 fun insert D (k, v) = blackenroot (ins D (k, v))
```

Thus, whenever `D` is a red-black tree, so is the result of `insert D (k, v)`.

Comment

Although the representation invariants in this example are a little involved, the code is nice and clean— and the rebalancing function makes good use of pattern-matching.

To complete the functor definition we modify the traversal function in the obvious way, omitting colors. (Why is this the right thing to do?) Here is the function definition:

```
fun trav Leaf = [ ]
|  trav (Node(l, (_, (k,v)), r)) = (trav l) @ (k,v) :: (trav r)
```

Here, for completeness, and for comparison with the original unbalanced implementation, is the red-black tree functor.

```
functor RBTDict(Key : ORDERED) : DICT =
 struct
   structure Key : ORDERED = Key;

   datatype color = Red | Black;

   datatype a tree =
      Leaf | Node of a tree * (color * (Key.t * a)) * a tree;
   type a dict = a tree;

   (* empty : a dict *)
   val empty = Leaf;

  (* lookup : a dict -> Key.t -> a option *)
  fun lookup Leaf k = NONE
    | lookup (Node (D1, (_ , (k, v)), D2)) k  =
         case Key.compare (k,k) of
             EQUAL    =>  SOME v
           | LESS     =>  lookup D1 k
           | GREATER  =>  lookup D2 k;

  (* blackenroot : a dict -> a dict *)
  fun blackenroot Leaf = Leaf
    | blackenroot (Node(D1, (_, (k,v)), D2)) = Node(D1, (Black, (k,v)), D2);

  type a data = a dict  * (color * (Key.t * a)) * a dict;
  (* Node : a data -> a dict *)
```

```
  (* balance : a data -> a data *)
  fun balance (Node(Node(a,(Red,x),b), (Red,y), c), (Black, z), d)
            = (Node(a,(Black,x),b),  (Red,y), Node(c,(Black,z),d))
  |   balance (Node(a, (Red,x), Node(b,(Red,y),c)),  (Black,z), d)
            = (Node(a,(Black,x),b), (Red, y), Node(c,(Black,z),d))
  |   balance (a,(Black,x),  Node(Node(b,(Red,y),c), (Red, z), d))
            = (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z), d))
  |   balance (a, (Black,x), Node(b, (Red,y), Node(c,(Red,z), d)))
            = (Node(a,(Black,x),b), (Red,y), Node(c,(Black,z), d))
  |   balance p = p;

 (* ins : a dict -> key * a -> a dict *)
  fun ins Leaf (k, v) = Node(Leaf, (Red, (k, v)), Leaf)
   |  ins (Node(l, (c, (k, v)), r)) (k, v) =
        case Key.compare(k, k) of
                EQUAL   => Node(l, (c, (k,v)), r)
         |      LESS    => Node(balance(ins l (k,v), (c, (k,v)), r))
         |      GREATER => Node(balance(l, (c, (k,v)), ins r (k,v)));

 (* insert : a dict -> Key.t * a -> a dict *)
 fun insert D (k, v) = blackenroot (ins D (k, v));

 fun trav Leaf = [ ]
 |  trav (Node(l, (_, (k,v)), r)) = (trav l) @ (k,v) :: (trav r)
end;
```

Make sure you understand what's visible outside the functor body, and what's not.

Draw pictures of the red-black trees that represent the results of building:

- `val T0 = empty`
- `val T1 = insert T0 (1, "1")`
- `val T2 = insert T1 (2, "2")`
- `val T3 = insert T2 (3, "3")`

7 Comparing two implementations

Suppose we build a red-black tree implementation of dictionaries, and a binary-search tree implementation of dictionaries. Each is a structure with the same signature, `Dict`. So each one allows users to build an empty dictionary and perform a sequence of insertions, to search the dictionary for an entry with a key "equal" to a given key, and also to extract the sorted list of key-value entries at any stage.

How, if at all, could a user distinguish between the two implementations? Well, we've shown that the red-back tree version is asymptotically faster. But in practice, unless we're able to run a large-scale comparison on a huge sized dictionary, with instrumentation tools for assessing runtime. it's probably not going to be easy to notice much difference in practice that way. And if we just ask for "visible" results, i.e. we check the traversal lists produced using both versions after exactly the same sequences of insertions, we'll *never* be able to observe a difference!

The reason is that we can prove a *theorem* that makes precise the sense win which the two implementations are *observably equivalent*. Since the only visible operations are `empty`, `insert`, `lookup`, and `trav`, what we need are some lemmas that link the observable behavior of these functions in the two implementations. SInce the actual types used inside these implementations are different (one with colors, one without) we need to be careful to avoid confusion. Luckily we can use qualified names to keep things straight!

Suppose `Bst:DICT` and `Rbt:DICT` are a binary-search tree dictionary implementation and a red-black tree dictionary implementation, defined by

```
structure Bst : DICT = BSTDict(Integers);
structure Rbt : DICT = RBTDict(Integers);
```

The only ways to produce "observable" results from a dictionary value are to apply `lookup`, or `trav`. So we'll say that a pair of values `(L, R) : string Bst.dict * string Rbt.dict` is *observationally equivalent* iff

```
(i)  For all values k:int, Bst.lookup L k = Rbt.lookup R k.
(ii) Bst.trav L = Rbt.trav R.
```

So observational equivalence means giving the same lookup results and the same traversal list.

The following results can be proven:

- `Bst.empty` and `Rbt.empty` are obs. equivalent.
- Whenever `L` and `R` are obs. equivalent, so are `Bst.insert L (k, v)` and `Rbt.insert R (k, v)`.

And as a corollary, because of the fact that *only* the empty value and the insert function are offered to clients of this abstract type, there's no way for clients to tell the two implementations apart. (If you think that users could guess which is which based on the names we gave to the structures, that's naïve. We could have been maliciously used the wrong names, or invented completely misleading names.)