15-150 Lecture 21:
Red-Black Trees

Lecture by Dan Licata

April 3, 2012

Last time, we talked about the signature for dictionaries:

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

signature DICT =
sig
  structure Key : ORDERED
  type 'v dict

  val empty  : 'v dict
  val insert : 'v dict -> (Key.t * 'v) -> 'v dict
  val lookup : 'v dict -> Key.t -> 'v option
end
```

and we implemented dictionaries as binary search trees. The problem with this implementation is that certain sequences of inserts will lead to unbalanced trees, in which case operations like lookup and insert take linear, rather than logarithmic, work.

A *balanced binary search tree* consists of a collection of operations that maintain balance, so that the operations take log time. In this lecture, we will implement *red-black trees*. The code is in Figure 1.

1 Invariants

The datatype for red black trees, `'v tree`, is like the binary search tree from the previous lecture, except we annotate each node with a color, which is either red or black. By convention, `Empty` is considered black. A tree is a *red-black tree (RBT)* if it satisfies the following invariants:

(sorted) It is sorted according to `Key.compare`.

(well-red) No red node has a red child.

```
functor RBTDict(Key : ORDERED) : DICT =
struct
  structure Key : ORDERED = Key
  datatype color = Red | Black
  datatype 'v tree =
      Empty
    | Node of 'v tree * (color * (Key.t * 'v)) * 'v tree
  type 'v dict = 'v tree (* representation invariant: is a RBT *)

  val empty = Empty
  fun lookup d k =
      case d of
          Empty => NONE
        | Node (L, (_ , (k', v')), R) =>
              case Key.compare (k,k') of
                  EQUAL => SOME v'
                | LESS => lookup L k
                | GREATER => lookup R k

  fun insert d (k, v) =
   let
    (* Root is Red,   both RBT --> ARBT;
       Root is Black, at most one ARBT, and the other(s) RBT --> RBT;
       if both args have the same black-height, then so does the result.
     *)
    fun balance p =
     case p of
         (Node(Node (a , (Red, x) , b) , (Red , y) , c) , (Black , z) , d) =>
             Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
       | (Node(a , (Red , x) , Node (b , (Red , y) , c)) , (Black , z) , d) =>
             Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
       | (a , (Black , x) , Node(Node (b , (Red, y) , c) , (Red , z) , d)) =>
             Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
       | (a , (Black , x) , Node(b , (Red , y) , Node (c , (Red , z) , d))) =>
             Node (Node (a , (Black , x) , b) , (Red , y), Node (c , (Black , z) , d))
       | _ => Node p
    (* if d is an RBT[Red]   then ins d is an ARBT;
       if d is an RBT[Black] then ins d is an RBT;
       preserves the black-height. *)
    fun ins d =
        case d of
            Empty => Node (empty, (Red, (k, v)), empty)
          | Node (l, (c , (k', v')), r) =>
                case Key.compare (k,k') of
                    EQUAL => Node (l, (c, (k, v)), r)
                  | LESS => balance (ins l, (c , (k', v')), r)
                  | GREATER => balance (l, (c , (k', v')), ins r)
    (* if t is an ARBT then blackenRoot t is a RBT *)
    fun blackenRoot t = case t of Empty => Empty
       | Node (l , (_ , x) , r) => Node (l , (Black , x) , r)
   in blackenRoot (ins d)
   end
end
```


Figure 1: Red-Black Trees

(black-height) All paths from the root to a leaf have the same number of black nodes. This number is called the *black-height*.

If all nodes were black, then the black-height invariant would ensure that the tree is perfectly balanced. In a well-red tree, the most a path can deviate from this is by alternating red and black nodes, which means that the longest path is no more than twice the length of the shortest path. This is balanced enough to get good logarithmic time bounds on insert and lookup. As we will see, the above invariants are a good *fullcrum*: they are easier to manage in code than "the tree has logarithmic depth," in part because well-red is a local structural invariant. But they are a sufficient condition for this property.

The key idea of today's lecture is that

abstract types enable local reasoning about representation invariants

Using the module system, we can ensure that clients of the `RBTDict` functor only ever see well-formed red-black trees. We do this by defining a representation invariant of the abstract type `'v dict` which states that a `'v dict` must be a RBT.

Then, considering all the operations in the signature, we prove that the operations that produce dictionaries produce trees that satisfy the above invariants, assuming their inputs do. In this case:

- `empty` must be a RBT.
- `insert d (k,v)` must be a RBT if `d` is.

If we can show this, then we know that *in any program using `RBTDict`, every value of type `dict` is a RBT.* The reason is that, because `dict` is abstract, the only way a client can make one is by using the operations in the signature, and we have just proved that these operations preserve the RBT invariants. Thus, we can prove that the representation invariant holds in any big program, just by reasoning about one module. This is a really important notion of *modular program verification.*

Thus, we need to consider these two operations. `empty` is easy: Fortunately, a `Empty` is one: It is trivially sorted; it is trivially well-red (there are no red nodes), and the one path from the root to itself as length one.

`lookup` is implemented in the same way as above, except that it ignores the color of the node. It may assume that the tree it is given is a RBT, which ensures that lookups take logarithmic time.

The interesting bit is in `insert`, which may assume it is given a RBT, but must ensure that the result is as well.

2 Insert

Suppose we do a simple-minded `insert`:

```
fun insert d (k, v) =
 let
  fun ins d =
      case d of
          Empty => Node (empty, (Red, (k, v)), empty)
        | Node (l, (c , (k', v')), r) =>
              case Key.compare (k,k') of
```

```
                    EQUAL => Node (l, (c, (k, v)), r)
                  | LESS => Node (ins l, (c , (k', v')), r)
                  | GREATER => Node (l, (c , (k', v')), ins r)
in (ins d)
end
```

If the key is not found, we create a new red node; after recursively inserting, we simply reconstruct the tree. This satisfies the black-height invariant, because it only inserts a red node. However, it runs afoul of the well-red invariant:

If we insert 1 into into

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

We'll get

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

which has a red node with a red parent.

This is not a RBT. But *that's okay, as long as we fix it up eventually.* This is the important idea of a *critical section*: inside the implementation of a module, you can break the external invariants, as long as you fix them up by the time clients seem the results. From the outside, `insert` takes a RBT and produces a RBT. But internally, in the process of doing an insert, it may work with trees that do not satisfy these invariants.[1]

In particular, we will allow invariant violations of the following sort: an *almost red-black tree (ARBT)* is like a RBT, except that instead of being well-red, it must be

> (almost-well-red) a tree is almost well-red if no red node has a red child, except perhaps the root

2.1 Rebalancing

Above, when we naïvely inserted into a RBT with a red root,

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

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

then we got an almost-well-red tree as a result.

When we try to make an almost-well-red tree the child of a black node, we can *rebalance* the tree to produce a tree that is actually well-red; this balancing scheme is due to Chris Okasaki (*Red-Black Trees in a Functional Setting*, Journal of Functional Programming, 1999).

Balancing is illustrated in Figure 2.1. 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 any case, we can *rotate* `z` under `y`, making the root red and `x` and `z` black.

This produces a well-formed RBT. Consider the case where the left is an ARBT, the right is an RBT, the left-left child is a red child of a red node, and the input tree is sorted.

- Sortedness: because the input is sorted, everything in c is greater than or equal to y, but less than or equal to z, so it can go as z's left subchild.

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

- Black-height: By assumption, the input has a black-height, which must be $h + 1$ (because the root is black), where each of a, b, c, d has a black-height of h. Thus, the result also has a black-height of $h + 1$ because each of the two children of the red root are black.

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.

The correctness proof for the other cases of rebalancing are analogous.

2.2 Code

Returning to Figure 1, the function `balance` implements the rebalancing described above. Each of the first four clauses is a simple ML transcription of the four invariant-violating states; the only difference is that the two-dimensional tree is turned into a linear sequence of constructors; the result in each case corresponds directly to the rearrangement described above. If the input is not in one of these four states, we simply apply the `Node` constructor.

The spec for balance is that (1) if the root is black, and at most one tree is an ARBT and the other(s) is (are) an RBT, then the result is an RBT; (2) if the root is red, and both subtrees are RBTs, then the result is an ARBT; (3) if both inputs have the same black-height, then so does the result. Proof: For (1), if either tree is in fact an ARBT with a red child of a red root, then it matches one of the first four clauses, and will be rebalanced as above, which produces an RBT, as argued above. If there is no violation, then both the left and right are RBTs, and putting two

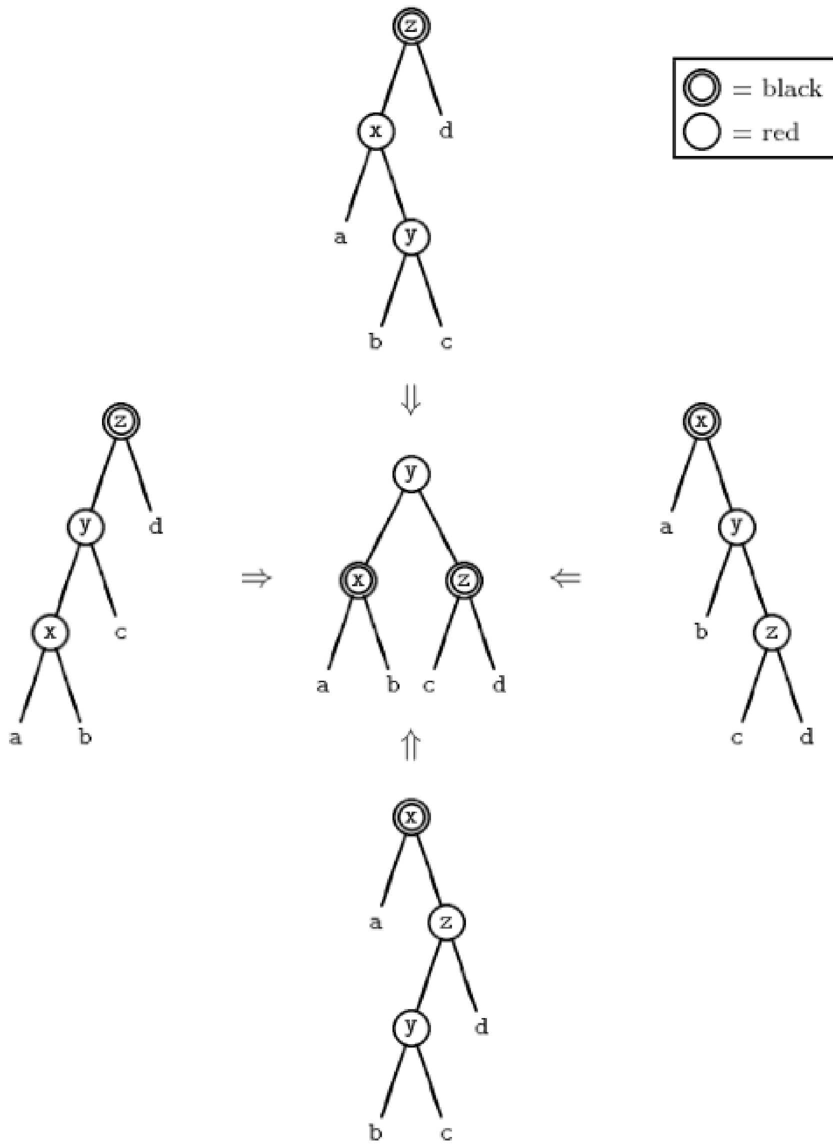

Fig. 1. Eliminating red nodes with red parents.

Figure 2: RBT Balancing from Okasaki, JFP'99

RBTs under a black node creates an RBT. For (2), because both the left and right are RBTs, we will apply the `Node` constructor, and putting two RBTs under a red node constructs an ARBT (because one or both of the roots might be red). For (3), we proved this above for the rebalancing cases; applying `Node` clearly maintains the black-height as well.

There are several circumstances under which this balance function does not create an RBT: First, a red root with an ARBT as a child—it would create a tree with a red root, child, and grandchild. Second, a black root with two ARBTs as children—the rotation only fixes one of them. Fortunately, neither of these come up at the call sites of balance.

The reason is that the spec for `ins` says that it (1) takes a red-rooted tree to an ARBT and (2) takes a black-rooted tree to an RBT. In the leaf case, the one-element tree is an RBT. In the `Node` case, if the key is found, then we leave the structure of the tree unchanged, so it is still an RBT. Otherwise, `ins` proceeds by recursively inserting into one side or the other; consider the case for `LESS`. If `c` is black, then `ins l` will be an RBT if the root of `l` is black, or an ARBT if the root of `l` is red. In either case, `balance` creates an RBT, because `r` is an RBT by assumption. By assumption, `l` and `r` have the same black-height; because `ins` preserves the black-height, `ins l` and `r` have the same black-height; and because `balance` preserves the black-height, the result has the same black-height as the input. On the other hand, if `c` is red, then the root of `l` *must be black* (because the input is well-red) and so `ins l` is an RBT, and thus `balance` creates an ARBT (the impossible case where the root of `l` is red is one the cases that `balance` does not handle). The black-height is preserved similarly. The case for `GREATER` is similar. Because we only recur into one side or the other, at most one side will be an ARBT, avoiding the other case that `balance` doesn't handle.

Because `ins` sometimes returns an ARBT, we need to restore the RBT invariant before returning from `insert`. Fortunately, an ARBT can always be transformed into an RBT by making the root black. This trivially maintains sorting. It makes a well-red tree out of an almost-well-red one, because the red-red root violation is removed. And it maintains the fact that the tree has a black-height, though it potentially increases the black-height by one.

Thus, the result of insert is an RBT.

Though the invariants are a little involved, the code is nice and clean—rebalancing makes good use of pattern-matching.