15-150
Principles of Functional Programming

Some Slides for Lecture 18
Red Black Trees
March 27, 2018
Michael Erdmann
Dictionary Signature

```ocaml
signature DICT =
  sig
    type key = string         (* concrete *)
    type 'a entry = key * 'a  (* concrete *)

    type 'a dict              (* abstract *)

    val empty : 'a dict

    val lookup : 'a dict -> key -> 'a option
    val insert : 'a dict * 'a entry -> 'a dict
  end
```

Dictionary Implementation

signature DICT =
  sig
    type key = string         (* concrete *)
    type 'a entry = key * 'a  (* concrete *)
    type 'a dict              (* abstract *)
    val empty : 'a dict
    val lookup : 'a dict -> key -> 'a option
    val insert : 'a dict * 'a entry -> 'a dict
  end

Last week we implemented

structure BinarySearchTree : DICT = ...

using a tree to represent a dictionary, with the Representation Invariant that the tree is sorted on key.
Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

```haskell
datatype 'a dict =
    Empty
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict

(Empty considered black.)
```
Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

datatype 'a dict =
    Empty
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict

(Empty considered black.)

Red Black Tree (RBT) Invariants:

1. The tree is sorted on the key part of the entries.
2. The children of a Red node are Black.
3. Each node has a well-defined black height:
   The number of Black nodes on any path from the node down to an Empty is the same.
Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

```
datatype 'a dict =
    Empty |
      Red of 'a dict * 'a entry * 'a dict |
    Black of 'a dict * 'a entry * 'a dict
```

(Empty considered black.)

Red Black Tree (RBT) Invariants:

1. The tree is sorted on the key part of the entries.
2. The children of a Red node are Black.
3. Each node has a well-defined black height:
   The number of Black nodes on any path from the node down to an Empty is the same.

**Invariants imply the tree is roughly balanced:**

\[ \text{depth} \leq 2\log_2(|\text{nodes}| + 1) \]
A given Red Black Tree:

(For presentational simplicity, only showing keys, and using integer keys not strings.)
Now insert 20:
Now insert 20:

What should we color this node?
Let’s color it red, to preserve black height.
Now insert 19:
Now insert 19:

RED-RED VIOLATION!
Fix with a rotation and recoloring:
2 of the 4 possible kinds of rotations:
2 of the 4 possible kinds of rotations:
2 of the 4 possible kinds of rotations:
2 of the 4 possible kinds of rotations:

1. RestoreLeft (1st clause)
2. RestoreLeft (2nd clause)
The other 2 kinds of rotations:

1. restoreRight (1st clause)
2. restoreRight (2nd clause)
Here is another example:
Again, let’s insert 20:

Insert 20 and color red (as before)
Once again, let’s insert 19:
RED-RED VIOLATION!
Again, fix with rotation & recoloring:
OH NO! There is a new RED-RED VIOLATION!
That’s OK. We can rotate again ...
Use this kind of rotation:
Here’s the tree again before rotation:
...giving us this after the rotation:
(It’s not necessary, but we can also safely recolor the root black.)
Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

```
datatype 'a dict =
    Empty
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict
```

(Empty considered black.)

Red Black Tree (RBT) Invariants:

1. The tree is **sorted** on the **key** part of the entries.
2. The **children** of a **Red** node are **Black**.
3. Each node has a well-defined **black height**: The number of **Black** nodes on any path from the node down to an **Empty** is the same.
Red Black Tree Dictionaries

Binary search tree with Red and Black nodes:

```ocaml
datatype 'a dict =
    Empty
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict

(Empty considered black.)
```

Red Black Tree (RBT) Invariants:

1. The tree is **sorted** on the **key** part of the entries.
2. The **children** of a **Red** node are **Black**.
3. Each node has a well-defined **black height**: The number of **Black** nodes on any path from the node down to an **Empty** is the same.

Almost RBT (ARBТ) Invariants:

1. and (3) as above.
2'. Like (2), but: **Red root** may have one **Red** child.
Specs for restoreLeft

(*

restoreLeft : 'a dict -> 'a dict

REQUIRES: Either d is a RBT
    or d's root is Black,
    its left child is an ARBT,
    and its right child a RBT.

ENSURES: restoreLeft(d) is a RBT,
    containing exactly the same
    entries as d, and with the
    same black height as d.

*)
Picture-based Programming
fun

restoreLeft

(Black(Red(Red(d1, x, d2), y, d3), z, d4)) =

Red(Black(d1, x, d2), y, Black(d3, z, d4))
Code for restoreLeft

(*
  restoreLeft : 'a dict -> 'a dict

REQUIRES: Either d is a RBT
  or d's root is Black,
  its left child is an ARBT,
  and its right child a RBT.

ENSURES: restoreLeft(d) is a RBT,
  containing exactly the same
  entries as d, and with the
  same black height as d.
*)

fun
  restoreLeft(Black(Red(Red(d1,x,d2),y,d3),z,d4)) =
    Red(Black(d1,x,d2), y, Black(d3,z,d4))

|restoreLeft(Black(Red(d1,x,Red(d2,y,d3)),z,d4)) =
  Red(Black(d1,x,d2), y, Black(d3,z,d4))

|restoreLeft d = d
Specs for \texttt{insert} and \texttt{ins}:

\begin{verbatim}
(*
    insert : 'a dict * 'a entry -> 'a dict

REQUIRES: d is a RBT.
ENSURES: insert(d,e) is a RBT containing 
        exactly all the entries of d 
        plus e, with e replacing an entry 
        of d if the keys are EQUAL.

Locally defined helper function \texttt{ins}:

ins : 'a dict -> 'a dict

REQUIRES: d is a RBT.
ENSURES: ins(d) is a tree containing 
        exactly all the entries of d 
        plus e, with e replacing an entry 
        of d if the keys are EQUAL.

ins(d) has the same black height as d.

Moreover, ins(Black(t)) is a RBT 
and ins(Red(t)) is an ARBT.
*)
\end{verbatim}
Code for **insert**

(* insert : 'a dict * 'a entry -> 'a dict REQUIRES and ENSURES RBT. *)

fun insert (d, e as (k, v)) = 
  let
    fun ins ... (will write shortly)
  in
    case ins(d) of
    Red(t as (Red_,_,_,_)) => Black(t)
  d' => d'
  end
Code for **insert**

(* insert : 'a dict * 'a entry -> 'a dict 
  REQUIRES and ENSURES RBT. *)

fun insert (d , e as (k, v)) = let
  fun ins ... (will write shortly)
in
  case ins(d) of
    Red(t as (Red_,_,_)) => Black(t)
    | Red(t as (_,_,Red_)) => Black(t)
    | d' => d'
end

recall the keyword **as** means
*layered pattern matching*
Code for insert

(* insert : 'a dict * 'a entry -> 'a dict
  REQUIRES and ENSURES RBT. *)

fun insert (d, e as (k, v)) =
  let
    fun ins ... (will write shortly)
    in
      case ins(d) of
        Red(t as (Red_,_,_,_)) => Black(t)
        | Red(t as (_,_,_,Red_)) => Black(t)
        | d' => d'
    end

Here is an acceptable alternate for the case:

    case ins(d) of
        Red(t) => Black(t)
        | d' => d'
Code for `ins`

(* ins : 'a dict -> 'a dict  
  REQUIRES: d is RBT.  
  ENSURES: ins(Black(t)) is RBT, 
            ins(Red(t)) is ARBT.  
  Recall:  e as (k,v) is in scope.*)

fun ins (Empty) = Red(Empty, e, Empty)  
|  ins (Black(l, e’ as (k’,_), r)) =  
   (case String.compare(k,k’) of  
      EQUAL => Black(l,e,r) (* replace *)  
    | LESS => restoreLeft(Black(ins(l),e’,r))  
    | _ => restoreRight(Black(l,e’,ins(r))))  
|  ins (Red(l, e’ as (k’,_), r)) =  
   (case String.compare(k,k’) of  
      EQUAL => Red(l,e,r) (* replace *)  
    | LESS => Red(ins(l),e’,r)  
    | GREATER => Red(l,e’,ins(r)))
**Code for ins**

(* ins : 'a dict -> 'a dict

REQUIRES: d is RBT.
ENSURES: ins(Black(t)) is RBT,
         ins(Red(t)) is ARBT.
Recall:  e as (k,v) is in scope.*)

fun ins (Empty) = Red(Empty, e, Empty)
| ins (Black(l, e' as (k',_), r)) =
  (case String.compare(k,k') of
     EQUAL => Black(l,e,r) (* replace *)
  | LESS => restoreLeft(Black(ins(l),e',r))
  |_ => restoreRight(Black(l,e',ins(r))))
| ins (Red(l, e' as (k',_), r)) =
  (case String.compare(k,k') of
     EQUAL => Red(l,e,r) (* replace *)
     LESS => Red(ins(l),e',r)
     GREATER => Red(l,e',ins(r)))

Why do we not call restoreLeft or restoreRight here?
Code for `lookup`

(* lookup : 'a dict -> key -> 'a option *)

fun lookup d k =
  let
    fun lk (Empty) = NONE
    | lk (Red t) = lk' t
    | lk (Black t) = lk' t

    and lk' (l, (k',v), r) =
      (case String.compare(k,k') of
        EQUAL => SOME(v)
        | LESS => lk(l)
        | GREATER => lk(r))

    in
      lk d
    end
### Code for `lookup`

(* `lookup` : 'a dict -> key -> 'a option *)

```ml
fun lookup d k =
  let
    fun lk (Empty) = NONE
    | lk (Red t) = lk' t
    | lk (Black t) = lk' t
    and lk' (l, (k', v), r) =
      (case String.compare(k, k') of
        EQUAL => SOME(v)
      | LESS => lk(l)
      | GREATER => lk(r))
  in
    lk d
  end
```

mutual recursion
Sample Usage

Suppose we have implemented the previous code as:

```ml
structure RBT : DICT = struct ... end
```

Now consider:

```ml
val r1 = RBT.insert(RBT.empty, ("a", 1))
```

Then ML will print:

```ml
val r1 = - : int RBT.dict
```

because of opaque ascription

because we put in an integer value

Now create the following:

```ml
val r2 = RBT.insert(r1, ("b", 2))
val look2 = RBT.lookup r2
```

Then

```
look2 : RBT.key -> int option
```

```ml
look2 "a" => SOME 1
```

```
look2 "c" => NONE
```