Modular Programming

Red-Black Trees
Last week

- Used binary search trees to implement dictionaries
- Observation: Binary search trees can degenerate into lists defeating the purpose of fast access and parallelism support.
Today

- Implement dictionaries using **Red-Black** trees — some sort of balanced tree.
Red/Black Tree Representation (RBT) Invariants:

1. The tree is **sorted**: for every node **Red** (left, (key1, value1), right) and every node **Black** (left, (key1, value1), right), every key in left is LESS than key1 and every key in right is GREATER than key1.

2. The tree is “**well-red**” : the children of a red node are black.

3. For any node, the number of black nodes on any two paths from that node to an Empty (leaf) is the same. This number is called the called the **black height** of the node.
Balance

Invariants imply the tree is roughly balanced:

\[ \text{height} \leq 2 \log_2 (|\text{nodes}| + 1) \]
Key idea

Abstract types enable local reasoning about representation invariants

Using the module system we will ensure that users of the case will ever see well-formed RBTs
Preserving the invariant

• In writing code:
  • **May assume** the invariant holds
  • **Must guarantee** to preserve the invariant
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:
The other 2 kinds of rotations:
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.)
Almost Red/Black Tree (ARBT) Invariants:

1. The tree is **sorted:**
   for every node **Red**(left, (key₁,value₁), right)
   and every node **Black**(left, (key₁,value₁), right),
   every key in left is LESS than key₁
   and every key in right is GREATER than key₁.

2. A red root may have one Red child.

3. For any node, the number of black nodes on any two paths from that node to an Empty (leaf) is the same. This number is called the **black height** of the node.
Preserving the invariant

• By inserting a node we may introduce a red-red violation

• Use a “relaxed” invariant: Almost Red-Black Tree that allows a single red-red violation

• Remove that violation and propagate that kind of removal upwards in the tree if necessary by recursive calls
signature DICT =

sig

type key = string  (* concrete type *)
type 'a entry = key * 'a (* concrete type *)

type 'a dict (* abstract type *)

val empty : 'a dict
val lookup : 'a dict -> key -> 'a option
val insert : 'a dict * 'a entry -> 'a dict
end
structure RedBlackTree :> DICT =
struct
  type key = string
  type 'a entry = string * 'a

datatype 'a dict =
  Empty               (* considered black *)
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict
Implementing rotations
(* 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. *)

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

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)) =
| restoreLeft dict =
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 dict =
(* 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. *)

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 dict = dict
(* restoreRight :'a dict -> 'a dict
  REQUIRES: Either d is a RBT
  or d's root is black, its left child is a RBT,
  and its right child an ARBT.
  ENSURES: restoreRight(d) is a RBT,
  containing exactly the same entries as d. *)

fun restoreRight(Black(d1, x, Red(d2, y, Red(d3, z, d4)))) =
  | restoreRight(Black(d1, x, Red(Red(d2, y, d3), z, d4))) =
  | restoreRight dict =
(* restoreRight : 'a dict -> 'a dict
   REQUIRES: Either d is a RBT
            or d's root is black, its left child is a RBT,
            and its right child an ARBT.
   ENSURES: restoreRight(d) is a RBT,
            containing exactly the same entries as d.
   *)

fun restoreRight(Black(d1, x, Red(d2, y, Red(d3, z, d4)))) = 
   Red(Black(d1, x, d2), y, Black(d3, z, d4))
| restoreRight(Black(d1, x, Red(Red(d2, y, d3), z, d4))) = 
   Red(Black(d1, x, d2), y, Black(d3, z, d4))
| restoreRight dict = dict
(* insert : 'a dict * 'a entry -> 'a dict
 REQUIRE: 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. *)

The locally defined helper function ins satisfies:

ins : 'a dict -> 'a dict
 REQUIRE: 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 preserves black height.
 Moreover, if d's root is black, then ins(d) is a RBT;
 if d's root is red, then ins(d) is an ARBT.
*)
Briefly

ins (Black(…)) satisfies the RBT invariants
ins (Red(…)) satisfies the ARBT invariants
fun insert (dict, entry as (key, datum)) = 
  let
    fun ins ....
  in
    case ins dict of
      Red (t as (Red _, _, _)) => Black t (* re-color *)
    | Red (t as (_, _, Red _)) =>  Black t (* re-color *)
    | dict => dict
  end
end (* structure RedBlackTree *)
fun insert (dict, entry as (key, datum)) = 
    let
        fun ins (Empty) = Red(Empty, entry, Empty)
            | ins (Red ...) = ins (Red ...
            | ins (Black ...) = ins (Black ...

    in
        case ins dict of
            Red (t as (Red _, _, _)) => Black t (* re-color *)
            Red (t as (_, _, Red _)) => Black t (* re-color *)
            dict => dict

    end

end (* structure RedBlackTree *)
fun insert (dict, entry as (key, datum)) = 
    let
        fun ins (Empty) = Red(Empty, entry, Empty)
        | ins (Red(left, entry1 as (key1, _), right)) =
          (case String.compare (key, key1) of
            EQUAL => Red(left, entry, right)
            | LESS => Red(ins left, entry1, right)
            | GREATER => Red(left, entry1, ins right))
        | ins (Black ...) 
    in
        case ins dict of
            Red (t as (Red _, _, _)) => Black t (* re-color *)
            | Red (t as (_, _, Red _)) => Black t (* re-color *)
            | dict => dict
    end
end (* structure RedBlackTree *)
fun insert (dict, entry as (key, datum)) = 
  let 
    fun ins (Empty) = Red(Empty, entry, Empty) 
        | ins (Red ....) 
        | ins (Black(left, entry1 as (key1, _), right)) = 
          (case String.compare (key, key1) of 
            EQUAL => Black(left, entry, right) 
            | LESS => Black(ins left, entry1, right) 
            | GREATER => Black(left, entry1, ins right)) 
  in 
    case ins dict of 
      Red (t as (Red _, _, _)) => Black t (* re-color *) 
      | Red (t as (_, _, Red _)) => Black t (* re-color *) 
      | dict => dict 
  end 
end (* structure RedBlackTree *)
fun insert (dict, entry as (key, datum)) =
  let
    fun ins (Empty) = Red(Empty, entry, Empty)
    | ins (Red(left, entry1 as (key1, _), right)) =
      (case String.compare (key, key1) of
        EQUAL => Red(left, entry, right)
      | LESS => Red(ins left, entry1, right)
      | GREATER => Red(left, entry1, ins right))
    | ins (Black(left, entry1 as (key1, _), right)) =
      (case String.compare (key, key1) of
        EQUAL => Black(left, entry, right)
      | LESS => restoreLeft(Black(ins left, entry1, right))
      | GREATER => restoreRight(Black(left, entry1, ins right)))
  in
    case ins dict of
      Red (t as (Red _, _, _)) => Black t (* re-color *)
    | Red (t as (_, _, Red _)) => Black t (* re-color *)
    | dict => dict
  end
end (* structure RedBlackTree *)
fun lookup dict key =
  let
    fun lk (Empty) = NONE
    | lk (Red tree) = lk' tree
    | lk (Black tree) = lk' tree
    and lk' (left, (key1,value1), right) =
      (case String.compare(key,key1)
        of EQUAL => SOME(value1)
        | LESS => lk left
        | GREATER => lk right)
    in
      lk dict
    end
Sample Usage

Assume `structure RBT :> DICT = struct ... end`

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

SML prints `val r1 = _ : int RBT.dict`

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

```ml
val look2 = RBT.lookup r2
```

SML prints `val look2 = fn : RBT.key -> int option`

```ml
look2 "a" evaluates to SOME 1
look2 "c" evaluates to NONE
```