15-150 Lecture 18
Summer 2018

Modular Programming III

Red-Black Trees

(edited from slides by Dilsun Kaynar)
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 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**\( (\text{left}, (\text{key}_1, \text{value}_1), \text{right}) \)
   and every node **Black**\( (\text{left}, (\text{key}_1, \text{value}_1), \text{right}) \),
   every key in left is LESS than key\_1
   and every key in right is GREATER than key\_1.

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
functor RBTDict(K: ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a

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

  val empty = Empty
fun balance (Black (Red (Red (d1, x, d2), y, d3), z, d4)) = Red (Black (d1, x, d2), y, Black (d3, z, d4))
balance (Black (Red (d1, x, Red (d2, y, d3)), z, d4)) = Red (Black (d1, x, d2), y, Black (d3, z, d4))

balance (Black (d1, x, Red (d2, y, Red (d3, z, d4)))) = Red (Black (d1, x, d2), y, Black (d3, z, d4))
balance (Black (d1, x, Red (Red (d2, y, d3), z, d4))) = Red (Black (d1, x, d2), y, Black (d3, z, d4))
(* balance : ‘a dict -> ‘a dict
 * REQUIRES: root is black, one child is RBT, other is ARBT
     OR root is red     and both children are RBTs
 * ENSURES: root is black and balance d is an RBT
     OR root is red     and balance d is an ARBT

fun balance (Black (Red (Red (d1, x, d2), y, d3), z, d4)) =
    Red (Black (d1, x, d2), y, Black (d3, z, d4))
| balance (Black (Red (d1, x, Red (d2, y, d3)), z, d4)) =
    Red (Black (d1, x, d2), y, Black (d3, z, d4))
| balance (Black (d1, x, Red (d2, y, Red (d3, z, d4)))) =
    Red (Black (d1, x, d2), y, Black (d3, z, d4))
| balance (Black (d1, x, Red (Red (d2, y, d3), z, d4))) =
    Red (Black (d1, x, d2), y, Black (d3, z, d4))
| balance d => d
Implementing insert

Implement a helper function ins that may return an ARBT
(* REQUIRES: d is an RBT
  * ENSURES: preserves black-height root is black and ins d is an RBT
  OR root is red and ins d is an ARBT

fun insert d (k, v) =
  let fun ins Empty = Red (Empty, (k, v), Empty)
    ins (Red (l, (k', v'), r)) =
      (case Key.compare (k, k') of
          EQUAL => Red (l, (k, v), r)
        | LESS => Red (ins l, (k', v'), r)
        | GREATER => Red (l, (k', v'), ins r))
  | ins (Black (l, (k', v'), r)) =
      (case Key.compare (k', v') of
          EQUAL => Black (l, (k, v), r)
        | LESS => balance
            (Black (ins l, (k', v'), r))
        | GREATER => balance
            (Black (l, (k', v'), ins r)))
fun blackenRoot (Red n) = Black n
| blackenRoot d       = d

let fun insert d (k, v) =
  let fun ins d = ...
in
  blackenRoot (ins d)
end