Red-Black Trees — Haskell

Bejenariu Răzvan-Andrei

May 2020

1 Binary Search Trees

Binary Search Trees (BST) is a node-based binary tree data structure, it allows for fast lookup, addition and removal of items having in general for this operations , a time complexity of **O(h)** where h is height of BST and in the worst case complexity of **O(n)**. Every BST has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

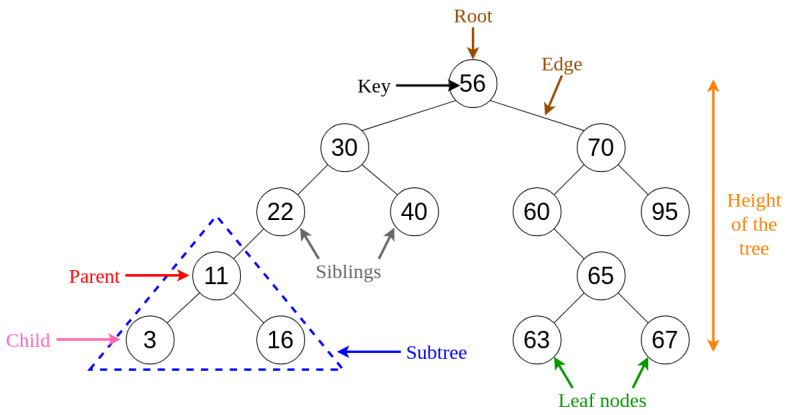


Figure 1: Visualization of Basic Terminology of Binary Search Trees. Credit to: https://towardsdatascience.com/self-balancing-binary-search-trees-101-fc4f51199e1d

2 Red-Black Trees

Red-Black Tree is a binary search tree that automatically tries to keep its height as minimal as possible at all times (even after performing operations such as insertions or deletions) and where every node complies with the following rules.

- Every node has a color either red or black.
- Root of tree is always black
- There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes

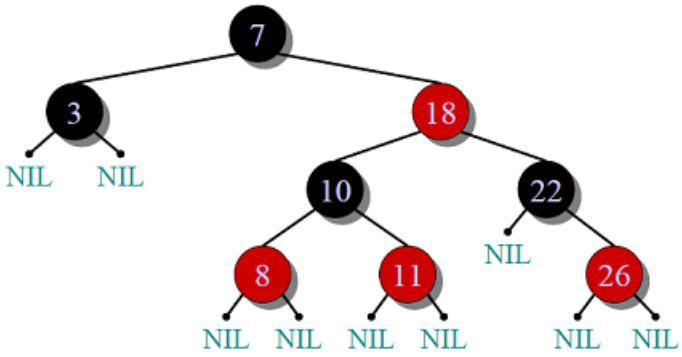


Figure 2: Example of an Red-Black Tree. Credit to: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

2.1 Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time. The cost of these operations may become O(n). If we make sure that height of the tree remains O($\log n$) after every insertion and deletion, then we can guarantee an upper bound of O($\log n$) for all these operations. The height of a Red-Black tree is always O($\log n$) where n is the number of nodes in the tree.

3 Haskell Implementation

3.1 Defining the tree

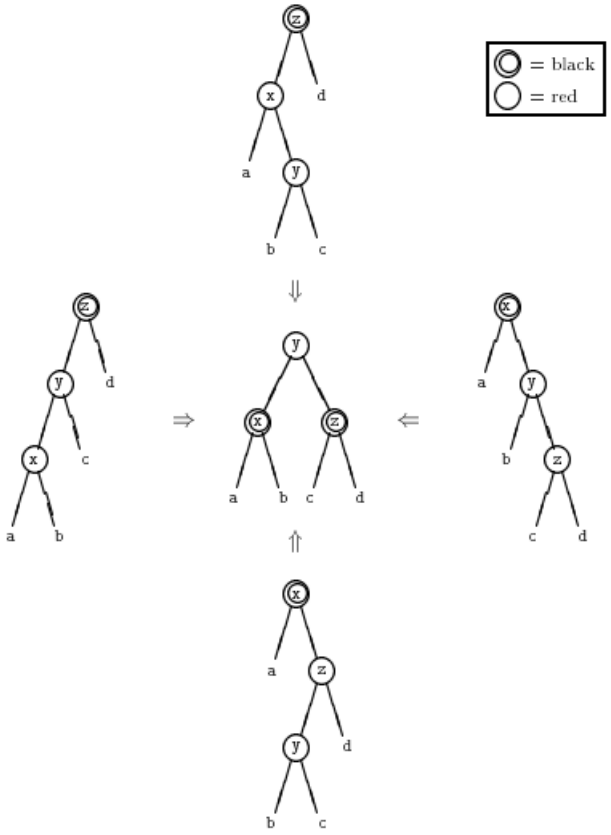
A tree consists of either an Empty node (NIL) which also represents an empty tree, or a branch containing one int value and one color (Red or Black which i defined) with exactly two subtrees.

```haskell
data Color = Red | Black deriving (Show, Eq)

data RBTree = NIL | Node Color Int RBTree RBTree deriving (Show, Eq)
```

3.2 Insertion

I use the algorithm described by Chris Okasaki in his book “Purely Functional Data Structures”. First, I create a new node that is initialized as being red, secondly I force the root of the tree resulted to be black so I can assure that the root is always black then I call the function balance which ”balances” the tree by checking if the insertion made 2 red nodes to be consecutive. There are 4 possible ways this can happen and using pattern matching we can rearrange each case separately.



Credit to: Chris Okasaki Functional Programming

```haskell
balance :: RBTree -> RBTree
balance (Node Black b (Node Red r1 (Node Red r2 tr2l tr2r) tr1r) btr) = (Node Red r1 (Node Black r2 tr2l t
balance (Node Black b (Node Red r1 tr1l (Node Red r2 tr2l tr2r)) btr) = (Node Red r2 (Node Black r1 tr1l t
balance (Node Black b btl (Node Red r1 (Node Red r2 tr2l tr2r) tr1r)) = (Node Red r2 (Node Black b btl tr2
balance (Node Black b btl (Node Red r1 tr1l (Node Red r2 tr2l tr2r))) = (Node Red r1 (Node Black b btl tr1
balance (Node color x tl tr) = (Node color x tl tr)

-----------------------------------------------------------

insertV :: Int -> RBTree -> RBTree
insertV v NIL = (Node Red v NIL NIL)
insertV v (Node c x tl tr)
    | v==x = (Node c x tl tr)
    | v>x = (balance (Node c x tl (insertV v tr)))
    | v<x = (balance (Node c x (insertV v tl) tr))

insertValue :: Int -> RBTree -> RBTree
insertValue v NIL = Node Black v NIL NIL
insertValue v (Node c x tl tr) = colorNode Black (insertV v (Node c x tl tr))
```

3.3 Deletion

I implemented deletion using multiple functions:

- deleteValue: Forces the root to be black after the operation is completed.

```
deleteValue :: Int -> RBTree -> RBTree
deleteValue v tree = colorNode Black (del v tree)
```

- del: searches for the required value recursively throughout the tree and calls the delL/delR function depending on the subtree the value is in or if it's found calls function uni.

```
del :: Int -> RBTree -> RBTree
del v NIL = NIL
del v (Node c x tl tr)
  | v < x = delL v (Node c x tl tr)
  | v > x = delR v (Node c x tl tr)
  | v==x = balance ( uni tl tr )
```

- delL / delR: As in the insertion algorithm we take each possible case and solve it independently in the balL/ balR and uni function.

```
delL :: Int -> RBTree -> RBTree
delL v (Node c x tl tr) = balL (Node c x (del v tl) tr)


delR :: Int -> RBTree -> RBTree
delR v (Node c x tl tr) = balR (Node c x tl (del v tr))
```

- uni: Function that unites the subtrees of the delete node.

```
uni :: RBTree -> RBTree -> RBTree
uni NIL NIL = NIL
uni tree NIL = tree
uni NIL tree = tree
uni (Node Black x tl tr) (Node Red x2 tl2 tr2 ) = (Node Red x2 (uni (Node Black x tl tr) tl2) tr2)
uni (Node Red x tl tr) (Node Black x2 tl2 tr2) = (Node Red x tl (uni tr (Node Black x2 tl2 tr2)))
uni (Node Red x tl tr) (Node Red x2 tl2 tr2)  =
    let c1 = uni tr tl2
    in case c1 of
        (Node Red x3 tl3 tr3) -> (Node Red x3 (Node Red x tl tl3) (Node Red x2 tr3 tr2))
        (Node Black x3 tl3 tr3)   -> (Node Red x tl (Node Red x2 c1 tr2))
        NIL -> (Node Black x tl (Node Red x2 tl2 tr2))
uni (Node Black x tl tr) (Node Black x2 tl2 tr2)  =
    let c2 = uni tr tl2
    in case c2 of
        (Node Red x3 tl3 tr3) -> (Node Red x3 (Node Black x tl tl3) (Node Black x2 tr3 tr2))
        (Node Black x3 tl3 tr3)   -> balL (Node Black x tl (Node Red x2 c2 tr2))
        NIL -> (Node Black x tl (Node Red x2 tl2 tr2))
```