Red-Black Tree

A *red-black tree* is a binary search tree with every node coloured red or black:

```
data Ord a => RBTree a = Nil
                       | Node RB a (RBTree a) (RBTree a)
data RB = R | B
```

with these restrictions:

- a red node has no red child
- all paths have the same number of black nodes

Red-Black Tree is Balanced

Suppose every path has m black nodes. Then:

- the shortest path has at least m nodes (say, all black)

- the longest path has at most $2m + 1$ nodes (alternating red and black)

It can be proved, using these two facts, that if a red-black tree has n nodes altogether, then the longest path has at most $2 \lfloor \lg (n + 1) \rfloor$ nodes. So every search takes logarithmic time.

```
member :: Ord a => a -> RBTree a -> Bool
member k Nil = False
member k (Node _ x t1 t2) | k<x = member k t1
                          | k>x = member k t2
                          | otherwise = True
```

Red-Black Tree Insertion

How to insert a key in logarithmic time and still preserve the red-black tree properties?

- First, insert the key as usual (treating the tree as a binary search tree).

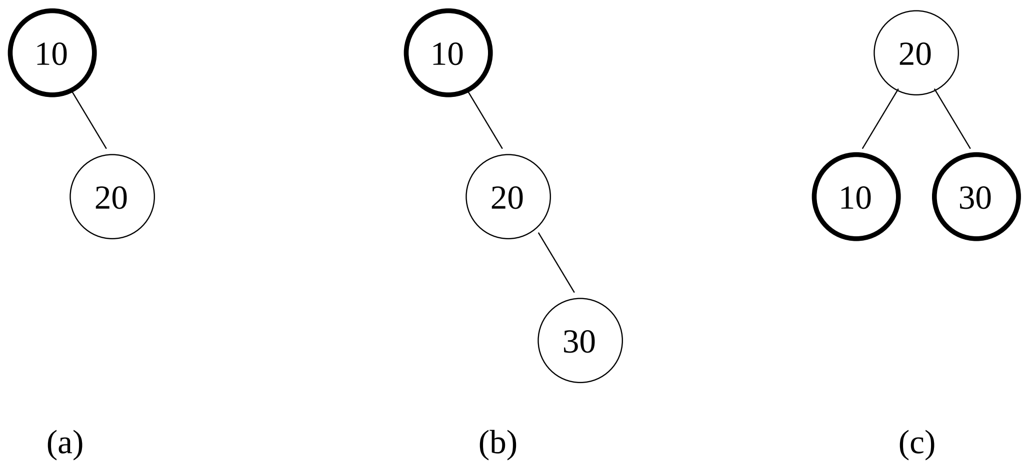
- What should be the colour of the new node?

  - If it is black, it will cause a difference in the number of black nodes on paths, and this is expensive to repair.
  - So we make it red.

- But then, the parent of the new node may be red as well. We will fix this by balancing as we return back to the root. This only takes logarithmic time.

Insertion and Balancing Example

In this example, we insert 30 into the tree in (a). Bold nodes are black.

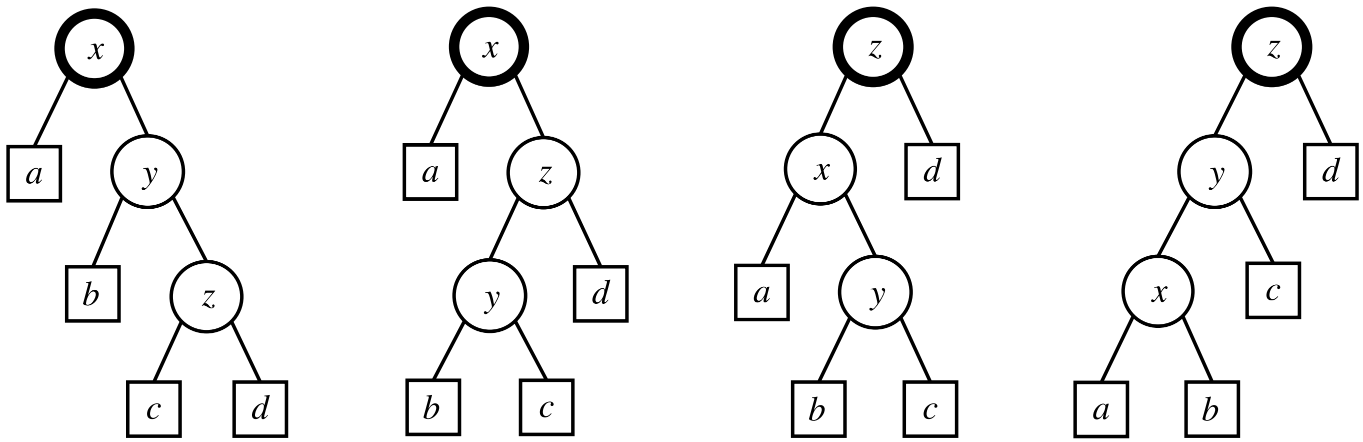


(a) start with this tree
(b) insert 30, get red-red violation
(c) balance

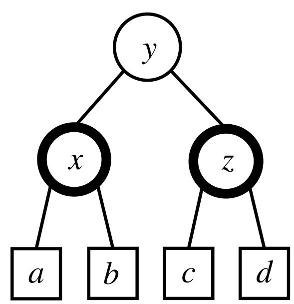
If this is a subtree and has a red parent, we go up and balance again.

Balancing Details

There are four forms of red-red violation:



Each is rotated to obtain the following:



Balance Code

The rotations are coded as follows:

```
balance B x a (Node R y b (Node R z c d)) =
  Node R y (Node B x a b) (Node B z c d)
balance B x a (Node R z (Node R y b c) d) =
  Node R y (Node B x a b) (Node B z c d)
balance B z (Node R x a (Node R y b c)) d =
  Node R y (Node B x a b) (Node B z c d)
balance B z (Node R y (Node R x a b) c) d =
  Node R y (Node B x a b) (Node B z c d)
balance c x t1 t2 = Node c x t1 t2
```

Note: This function works on the contents of the black node, instead of the black node itself, because it most probably gets thrown away anyway. (See how it is called in `insert` below.)

Insertion Code

Code to perform BST insertion and balancing:

```
ins d Nil = Node R d Nil Nil
ins d n@(Node c k t1 t2)
  | d<k = balance c k (ins d t1) t2
  | d>k = balance c k t1 (ins d t2)
  | otherwise = n
```

We need a finishing touch: when the red-red violation propagates to the root, we have no further black parent to help us balance it.

Solution: simple, just mark the root as black. Always.

```
insert d t = Node B x t1 t2
  where Node _ x t1 t2 = ins d t
```

(blank slide)