Red Black tree algorithm

Preserve BST properties add a color to each node

Color, parent, left, right, key

Tree is a BST with these properties

1. Every node must be either red or black
2. The root is black
3. Every leaf is black(Node at the bottom of the tree)
4. If the node is red, both its children must be black
5. For each node all paths from the node to the leaves must contain the same number of black nodes.

Rotations

Most tree balancing algorithms involve some rotations.

Left and right rotations are inverses of each other.

In the left rotation things that change are

x.parent

y.parent

x.right

y.left

left\_

rotate(T,x)

y = x.right

x.right = y.left

if y.left != NUL

y.left.p = x

y.p = x.p

if x.p – T.n.1

T.root = y //Makes the new root

Else if x == x.p.left //if x is its parents left child

x.p.left = y

else

x.p.right = y //make y the right child.

y.left = x

x.p = y

Sentinel node

T.nil NULL pointer is replaced by an empty node, T.nil is a black node.

Insert a node in a red black tree

1. Replace all instance of nil with t.nil
2. Set color of new node to red
3. Assign red to new node, may violate red-black properties, they are restored through the tree balancing algorithm
   1. Two of the red-black properties can be violated.
      1. The root must be black
      2. Children of a red node must be black

We can reduce all these problems down to six cases when adding a node.

Let x be the new node added to the tree T x.color = red, x.left = T.nil, x.right = T.nil

Let y be x’s uncle, x.parent.parent.right

Three cases

Case 1: x’s uncle y is red, if y is red, the x.parent is red

Case 2: x’s uncle y is black and x is a right child

Case 3: x’s uncle y is black and x is a left child

Example: Build a black red tree from <15, 11, 14, 2, 1>

rb.insert(T, node x) //Where x is the node being added

//Add 15 15(red) => 15(black)

//Add 11 11(red)

//Add 14 14(red) //Red node->red parent violated