Red-Black Trees Part 10

Robin Dawes

February 17, 2021

Continues the presentation of the insertion algorithm for Red-Black Trees. This is the most complex algorithm we have seen so far in this course. It rewards close attention with a deep understanding of how binary trees can be manipulated.

.

The Insertion Algorithm ... Live!

In the last note we discussed the cases that arise when we need to re-balance a Red-Black Tree after inserting a new value. Now we need to show that each of these cases can be resolved using a small number of pointer and colour adjustments to a few vertices of the tree.

I mentioned earlier that in the CLRS definition of a Vertex object, each Vertex contains a pointer to its parent Vertex. Using Parent pointers the balancing algorithm can be performed iteratively. The rotation operations require more updates since every time a child pointer is changed, the corresponding parent pointer must also change.

There is nothing wrong with this approach! If it appeals to you, please see the CLRS text for complete pseudo-code.

My personal preference is to do this recursively (surprise!) so that Parent pointers are not needed. Instead of looking "upwards" for a Red vertex with a Red parent, I prefer to look "downwards" from each vertex to see if it has a Red child that also has a Red child. In other words we make the grandparent work harder than anyone else (just like in real life).

The overall structure of this insertion method is identical to our previous insertion algorithm for plain binary search trees. All that has been added are the balancing operations. The fixes and rotations are done as we exit each level of recursion after the new vertex is added to the tree. Each vertex on the search path checks to see if there is a problem just below it, and fixes it if there is.

This is why the earlier insertion algorithm was written that way!

The logic of the recursive RB insert method looks like this:

```
recursive_RB_insert(Vertex v, int x):
    if (v is a leaf):
        return new Vertex(x)    # the new Vertex is automatically constructed with 2
                                # empty leaves below it
    else if (v.value > x):
        v.left_child = recursive_RB_insert(v.left_child, x)
                                # now v takes the role of GP and looks at its child and
                                # grandchild to see if there is a problem
        if (v.left_child.colour == B):
            return v            # no problem
        else:
                                # v.left_child is Red ... there may be a problem
                                # so check the children of v.left_child
            if (v.left_child.left_child.colour == R) OR (v.left_child.right_child.colour == R):
                                # Houston, we have a problem
                                #    figure out which case applies
                                #    make the fix
                                #    return the vertex that is now at the top
    else:                       # v.value < x
                                # do the mirror image of the the operations above, but use
                                # v.right_child instead of v.left_child
```

Note that there may be a problem at **v** itself - it may be Red and one of its children may also be Red. **v** doesn't care! It will let its parent solve that problem (this seems so familiar).

What follows is the annotated complete recursive RB insert algorithm, presented for your coding pleasure. Unfortunately it is too large for a single page.

```
# Each vertex in the tree is an object of the RB_Vertex class, which
# we assume is defined so that each vertex has the following attributes:
#     - is_a_leaf    : a Boolean values that is True if this vertex is a leaf
#     - colour       : Red or Black - this can be implemented as
#                      a single bit, or a string, or an integer
#     - value        : the value to be stored in this vertex, if any
#     - left_child   : a pointer to another RB_Vertex object
#     - right_child  : a pointer to another RB_Vertex object


def RB_insert(T,x):    # insert the value x into the Red-Black tree T
    T.root = rec_RB_insert(T.root,x)
    T.root.colour = Black      # we always colour the root Black


def rec_RB_insert(v,x):
    if (v.is_a_leaf):
        return new RB_Vertex(x)
                       # The constructor for RB_Vertex creates the vertex, colours
                       # it Red, and gives it two empty leaves coloured Black as children
    else if (v.value > x):
                       # we recurse down the left side
        v.left_child = rec_RB_insert(v.left_child,value)
                       # now check for problems:
                       #      we check to see if v needs to play the grandparent role
                       #      and fix a Red-Red problem between its child and grandchild

                       # Since we recursed down to the left from v, we only
                       # need to look at its left subtree.
        if (v.left_child.colour == Black):
                       # there is no problem here, officer
                       # (problem exists only if both child and grandchild are Red)
            return v
        else:
                       # v's left child is Red, so there may be a problem.
                       # Check the grandchildren - we know they exist because a Red vertex
                       # must have two children
            if (v.left_child.left_child.colour == Red) OR (v.left_child.right_child.colour == Red):
                       # PROBLEM!
                       # Now we identify the problem case
                if (v.left_child.left_child.colour == Red):
                    return Left_Left_fix(v)            # Note: we handle Case 1 & 2 problems
                                                       # inside these "fix" methods
                else:
```

```
                return Left_Right_fix(v)
        else:
                   # no problem after all
            return v

else:              # this is the else clause for "if (v.value > x)"
                   # We know v.value < x so we recurse down the right side.
                   # The logic is the same as for the left side,
                   # just with "left" and "right" exchanged
    v.right_child = rec_RB_insert(v.right_child,value)
                   # now check for problems
                   #      we check to see if v needs to play the grandparent role
                   #      and fix a Red-Red problem between its child and grandchild
                   # Since we recursed down to the right from v, we only
                   # need to look at its right subtree.
    if (v.right_child.colour == Black):
                   # there is no problem here, officer
                   # (problem exists only if both child and grandchild are Red)
        return v
    else:
                   # v's right child is Red, so there may be a problem.
                   # Check the grandchildren - we know they exist because a Red vertex
                   # must have two children
        if (v.right_child.right_child.colour == Red) OR (v.right_child.left_child.colour == Red):
                   # PROBLEM!
                   # Now we identify the problem case
            if (v.right_child.right_child.colour == Red):
                return Right_Right_fix(v)            # Note: we handle Case 1 & 2 problems
                                                     # inside these "fix" methods
            else:
                return Right_Left_fix(v)
        else:
                   # no problem after all
            return v
```

And now the methods that actually do the fixes:

```
# First the fixes that apply when we recursed to the left


def Left_Left_fix(GP):     # We know GP's left child is Red, and that child's left child is also Red
    P = GP.left_child      # P for Parent, following the nomenclature of the figures used above
    S = GP.right_child     # S for Sibling
    if S.colour == Red:
                           # Case 1 applies: no rotation needed
                           # Just recolour and return
        P.colour = Black
        S.colour = Black
        GP.colour = Red
        return GP
    else:
                           # S.colour == Black, so we need to do a single rotation
                           # We just fix the pointers appropriately
        GP.left_child = P.right_child
        P.right_child = GP
                           # and fix the colours
        P.colour = Black
        GP.colour = Red
                           # and return the new root of this subtree
        return P

def Left_Right_fix(GP):    # We know GP's left child is Red, and that child's right child is also Red
    P = GP.left_child
    S = GP.right_child
    if S.colour == Red:
                           # Case 1 applies: no rotation needed
                           # Just recolour and return
        P.colour = Black
        S.colour = Black
        GP.colour = Red
        return GP
    else:
                           # S.colour == Black, so we need to do a double rotation
                           # We just fix the pointers appropriately
        C = P.right_child
        P.right_child = C.left_child
        GP.left_child = C.right_child
        C.left_child = P
        C.right_child = GP
```

```
                        # and fix the colours
        C.colour = Black
        GP.colour = Red
                        # and return the new root of this subtree
        return C


# and now the fixes that apply when we recursed to the right

def Right_Right_fix(GP):
# just the mirror image of Left_Left_fix(GP) - you can write this

def Right_Left_fix(GP):
# just the mirror image of Left_Right_fix(GP) - you can write this
```

We noted above that once we reach a point where there is no problem (either because a vertex that was just coloured Red has a Black parent, or because we did a rotation, or because the tree was already in good balance without any fixes) there is no more fixing to be done: we don't need to look for more problems at any vertices closer to the root. This means we could terminate the insertion process immediately - but the recursive version will require us to continue to exit one level at a time.

The advantages of the recursive version are that it is concise (if you remove all the comment lines from the pseudo-code given above you will see how few lines of code there actually are - see below) and that it does not require Parent pointers - having Parent pointers would require more update operations during each rotation. The downside of the recursive version is that we cannot terminate the insertion process as soon as it is safe to do so.

Neither the advantages nor the disadvantages affect the *O* classification of the algorithm, but they can affect the real time performance.

Oh, if only there were some way to retain the advantages of the recursive method and eliminate the negatives ... but wait ... there is! We have seen a data structure that lets us simulate recursion without actually using recursion: we can use a stack! All we need to put on the stack are the vertices we visit during the search for the insertion point. Then we can pop them off the stack to work back up the tree, and as soon as we know the tree is properly balanced and coloured, we can just stop. Best of both worlds!

At least, the advantages that I perceive!

As an exercise, try implementing the RB insertion algorithm using a stack to simulate recursion. If you do, you can be justifiably proud of yourself.

Deletions from RB Trees are handled in the same general way: we do the deletion exactly as we learned for simple Binary Search Trees, then we work back up the tree making adjustments to restore the balance. The details are messy and we don't cover them in these notes.

An important restriction on RB Trees is that the values stored must all be distinct. For example, we cannot store the values $3, 8, 9, 8, 5$ in a R-B tree because there are two 8s in the set. Can you see why this restriction is essential?

Think about what might happen after a rotation on a tree that contains duplicate values.

Just the Pseudo-Code, Ma'am

HERE'S THE PSEUDO-CODE for **RB-insert** without all that annoying documentation:

```
def RB_insert(T,x):
    T.root = rec_RB_insert(T.root,x)
    T.root.colour = Black


def rec_RB_insert(v,x):
    if (v.is_a_leaf):
        return new RB_Vertex(x)
    else if (v.value > x):
        v.left_child = rec_RB_insert(v.left_child,value)
        if (v.left_child.colour == Black):
            return v
        else:
            if (v.left_child.left_child.colour == Red) OR (v.left_child.right_child.colour == Red):
                if (v.left_child.left_child.colour == Red):
                    return Left_Left_fix(v)
                else:
                    return Left_Right_fix(v)
            else:
                return v
    else:
        v.right_child = rec_RB_insert(v.right_child,value)
        if (v.right_child.colour == Black):
            return v
        else:
            if (v.right_child.right_child.colour == Red) OR (v.right_child.left_child.colour == Red):
                if (v.right_child.right_child.colour == Red):
                    return Right_Right_fix(v)
                else:
                    return Right_Left_fix(v)
            else:
                    return v
```

As promised, it is pretty concise when you consider all the different situations it has to handle.

And now the methods that actually do the fixes:

```
def Left_Left_fix(GP):
    P = GP.left_child
    S = GP.right_child
    if S.colour == Red:
        P.colour = Black
        S.colour = Black
        GP.colour = Red
        return GP
    else:
        GP.left_child = P.right_child
        P.right_child = GP
        P.colour = Black
        GP.colour = Red
        return P

def Left_Right_fix(GP):
    P = GP.left_child
    S = GP.right_child
    if S.colour == Red:
        P.colour = Black
        S.colour = Black
        GP.colour = Red
        return GP
    else:
        C = P.right_child
        P.right_child = C.left_child
        GP.left_child = C.right_child
        C.left_child = P
        C.right_child = GP
        C.colour = Black
        GP.colour = Red
        return C


# and now the fixes that apply when we recursed to the right

def Right_Right_fix(GP):
# just the mirror image of Left_Left_fix(GP)

def Right_Left_fix(GP):
# just the mirror image of Left_Right_fix(GP)
```