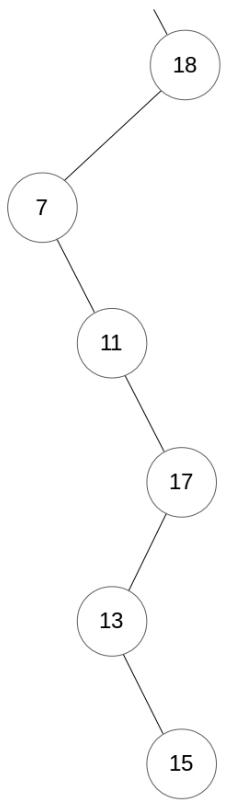
Red-Black Trees

As we have seen, the ideal Binary Search Tree has height approximately equal to log n, where n is the number of values stored in the tree. Such a BST guarantees that the maximum time for searching, inserting and deleting values is always ∈ O(log n). However, if we have no control over the order in which the values are added and/or deleted, the BST may end up looking like a linked list, with each vertex having just one child. In this case the maximum time for searching, inserting and deleting values is in O(n), which is no better (and in the case of searching, much worse) than the time required for these operations if we store the set in a sorted array.



If the values are inserted in the order 18, 7, 11, 17, 13, 15 the binary tree ends up looking like a linked list: each vertex has only one successor. The number of levels is equal to the number of values in the set.

In order to claim that BSTs are better than sorted arrays, we need to find a way to always attain that desirable O(log n) time for the three operations. One way would be to rebuild the tree from scratch after each insertion or deletion ... but that would be a lot of work.

A better option would be to establish a limit on the number of levels – for example, we might choose 2*log n. Then whenever the tree exceeds this number of levels, we could rebuild the tree from scratch and make it as compact as possible.

This is a very interesting idea. The "rebuild from scratch" operation is time-consuming (although not **too** bad – we can rebuild the tree in "perfect" form in O(n) time … I strongly recommend figuring out how!) but we probably don't do it very often. With luck, after rebuilding the tree we could do a lot of inserts before we had to rebuild again. Thus the average amount of extra work we do for each insert would be quite small. This would be a reasonable solution if we don't mind a significant delay, once in a while.

What we will see now is that it is possible to take another approach: keep the number of levels small by doing a little bit of extra work fairly often.

In the 1960's, people started to use the term "**balanced**" to describe trees where each vertex has the property that its left subtree and right subtree are "about the same height" ... of course "about the same height" can be interpreted in different ways.

Red-Black trees were invented in 1972 in an effort to create a binary search tree structure that maintains O(log n) height while requiring relatively few re-organizations of the tree. In a Red-Black tree, the idea of balance is "at each vertex, neither subtree is more than twice as tall as the other". For your own interest you may want to read about AVL trees, which have similar properties but a much stricter balance rule: at each vertex, the two subtrees must have heights that differ by no more than 1.

A Red-Black tree is a binary search tree in which each vertex is coloured either Red or Black. In practice all that is required is a single bit to indicate if the vertex is Red or Black, but for learning purposes we can imagine that the vertices are physically painted.

Red-Black trees are usually described as obeying the following rules :

1. All vertices are coloured Red or Black
2. The root is Black
3. All leaves are Black, and contain no data (ie data values are only stored in internal vertices)
4. Every Red vertex has 2 children, both of which are Black
5. At each vertex, all paths leading down contain the **same number** of Black vertices

This allows for significant differences in height between the left subtree and the right subtree at any given vertex. For example, the left subtree might consist entirely of Black vertices and have height x, while the right subtree might consist of alternating levels of Red and Black vertices and have height 2*x.

In fact, we can quickly show that in a Red-Black tree each vertex must have the property that if we look at the longest path down from this vertex to a leaf, this path cannot be more than twice the length of the shortest path down from this vertex to a leaf:

Let v be any internal vertex, and let the **longest** path from v down to a leaf have length k, and let b be the number of Black vertices in this path. In this path, every Red vertex must have a Black child, so the number of Red vertices must be $\leq \frac{k}{2}$. Thus $b \geq \frac{k}{2}$

Now consider the **shortest** path from v down to a leaf. Let the length of this path be k'. Since all paths from v down to a leaf must contain the same number of Black vertices (Rule 5), we know $k' \geq b$

Putting these together gives $k' \geq b \geq \frac{k}{2}$, which gives us $k \leq 2 * k'$ .... that is, the length of the longest path is $\leq$ twice the length of the shortest path.

At this point I am just going to claim that a tree that satisfies these rules must have O(log n) height, where n is the number of values in the tree. We will prove this claim later.

The significance of these rules is that they are all "local" in the sense that we are specifying properties of individual vertices, and yet by satisfying these local rules, we obtain the desired "global" property that the whole tree has O(log n) height. This means that when we do insertions and deletions, as long as our operations on the tree are such that the local requirements are always satisfied, we never need to worry that the height of the tree is growing out of control . As we will see, inserting new values into the tree can be done in such a way that the requirements are satisfied using only local changes to the tree. What's more, the balancing operations are simple.

We will not discuss the details of deleting a value from a Red-Black tree. The principles are the same, but it is time-consuming to cover all the details.

From now on in these notes I am going to be lazy and use RB instead of Red-Black.

Inserting A New Value into an RB Tree

As with any binary search tree, there is exactly one legal place for a new value to be inserted as a leaf. The RB insertion algorithm starts by finding this place. Due to the structural requirements of the tree the location will be occupied by a leaf which contains no data value.

Once the insertion point has been found, the insertion process proceeds as follows:

1. Replace the leaf by a new internal vertex containing the new value. Give this vertex two (empty) leaves as children, both coloured Black. Colour the new vertex Red. In practice, this can all be done in the constructor method for the new vertex.

Note that at this point, requirement 5 is still satisfied because inserting a Red vertex does not change the number of Black vertices on any path. The only requirements that may be violated are 1 (if the tree was empty, the new vertex is the root, and it should be Black) or 4 (the parent of the new vertex might be Red). We will deal with violations of Requirement 1 by ending every insertion with a "Colour the root Black" operation.

Violations of Requirement 4 are the ones that will occupy us.

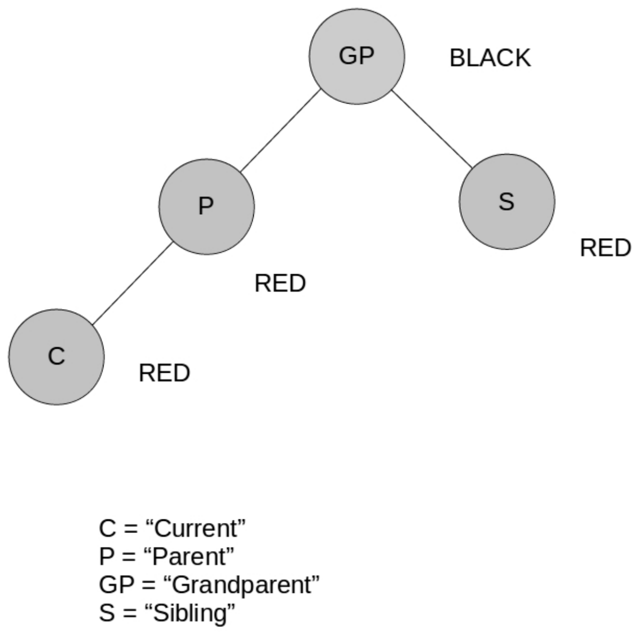
2. We work back up the path from the new vertex to the root, fixing the tree so that the requirements are satisfied at each point. Remarkably (and this is a wonderful feature of the RB tree structure), **we never have to check to make sure Rule 5 is satisfied!** This is a good thing because checking this would take a long time (at least O(n) for each insertion). The operations we do to fix the tree guarantee that Rule 5 will always be satisfied.

The text uses Vertex objects that have Parent pointers, and gives very clear pseudo-code for the entire insert operation. The basic idea is that whenever we are currently at a **RED** vertex with a **RED** parent, we need to fix something to satisfy the RB-tree rules. We do this by looking at the grandparent of the current vertex. We know three things about the grandparent:

- It must exist (because the Red parent cannot be the root)
- It must be Black (because the tree did not contain any Red-Red conflicts before the insertion).
- It must have two children (because all internal vertices in a RB tree have two children)

We examine the colour of the grandparent's other child, which in the figures below is labeled "S" for "sibling". There are several cases – each of the ones shown has the "Parent" as the left child of the "Grandparent". There are "mirror image" cases where the Parent is right child of the Grandparent.

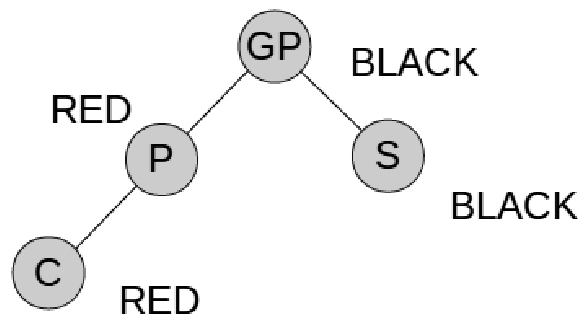
Case 1: The Sibling is RED



Case 1

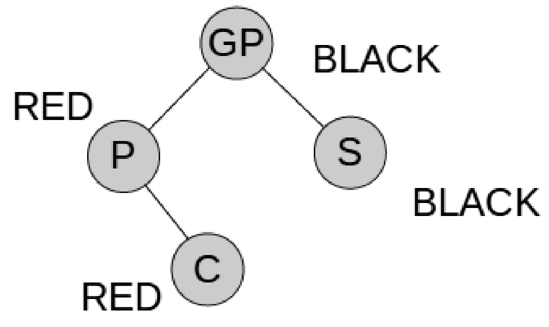
Case 2: The Sibling is BLACK

Case 2.1: The Current and Parent are on the same side (ie they are both left children or both right children)



Case 2.1

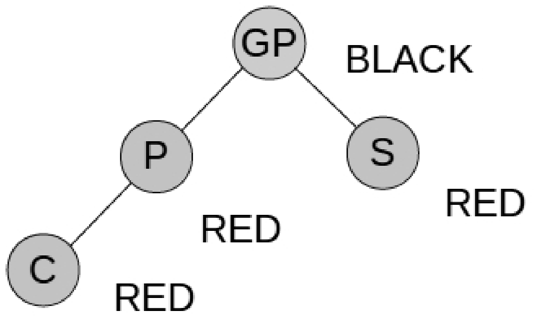
Case 2.2: The Current and Parent are on different sides (ie one is a left child and the other is a right child)



Case 2.2

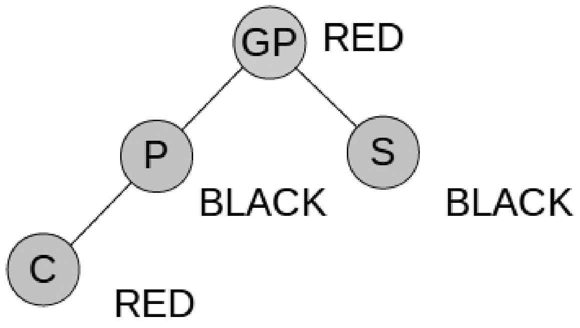
Based on the case, we either

Case 1: Don't change the structure. Simply recolour the grandparent, the parent and the grandparent's other child

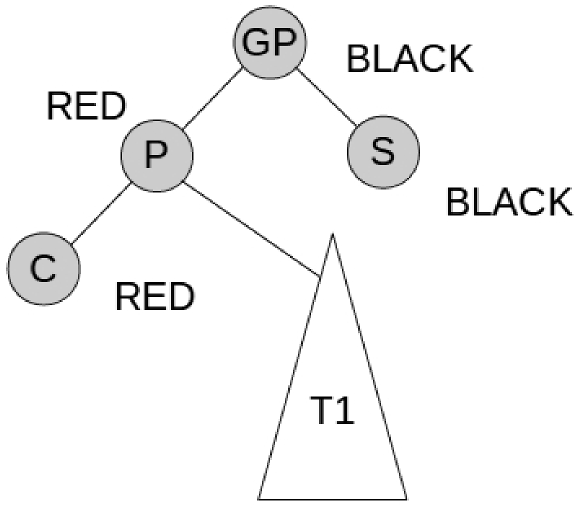


C = “Current”
P = “Parent”
GP = “Grandparent”
S = “Sibling”

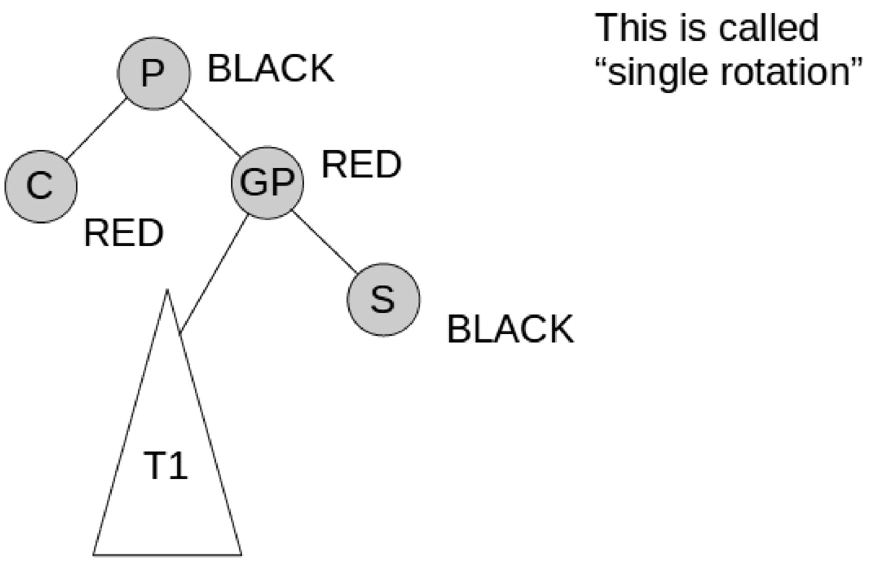
becomes



Case 2.1 Do a “single rotation” and recolour the vertices

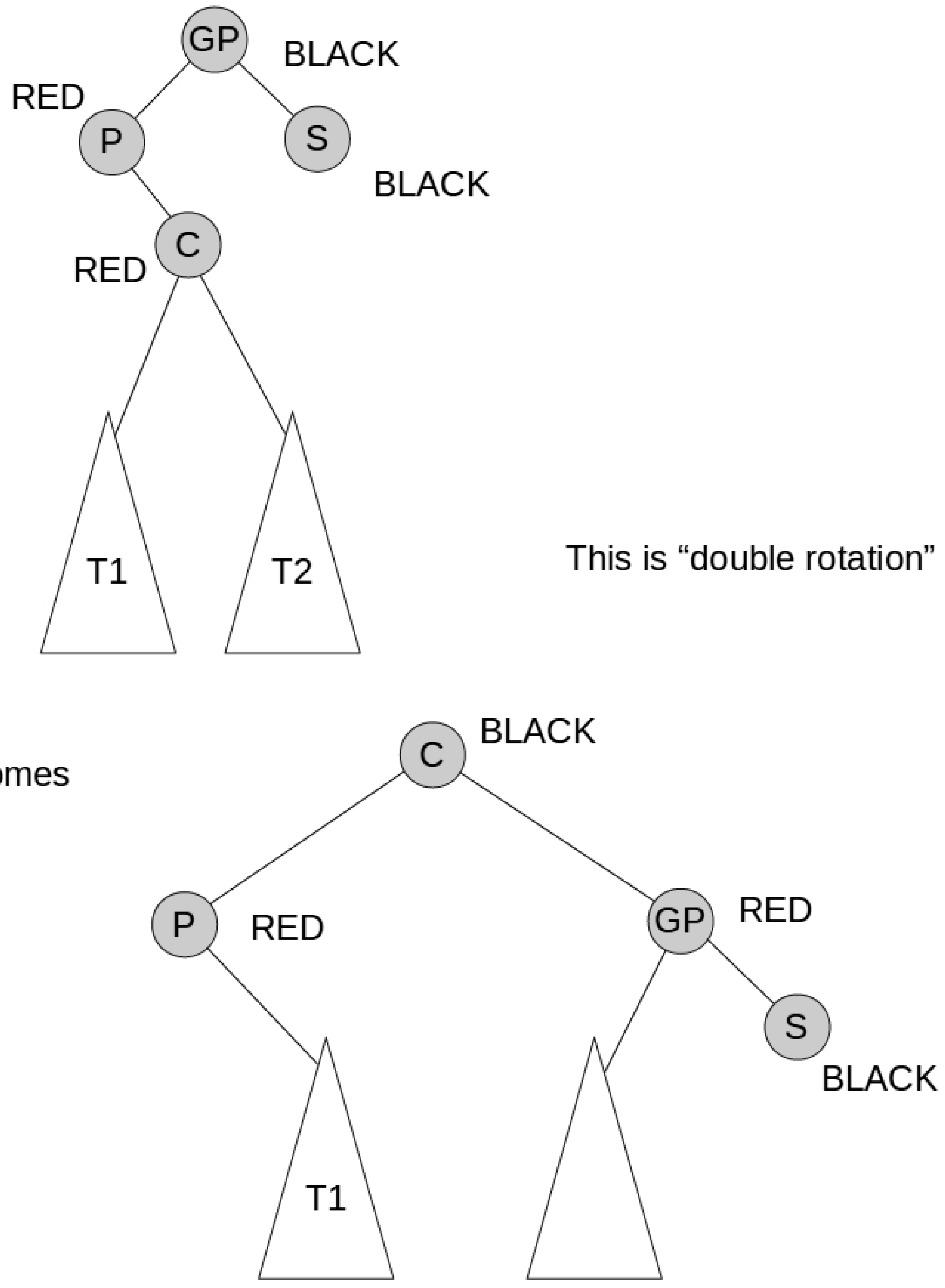


becomes



T1 has been included in the figure to show that it moves from being the right child of P to being the left child of GP.

Case 2.2 Do a “double rotation” and recolour the vertices



It is important to understand that these diagrams only show the parts of the tree that move and/or change colour – everything else is unaffected by the operation. For example: in the double rotation we know that P has a left child (because it is RED), but that left child is still the left child of P after the rotation so I didn't show it in the figure. Similarly in the single rotation we know that C has two children, but they are still the children of C after the rotation so there is no need to show them in the diagram.

It's important to establish that each of these three fixes

- eliminates the Red-Red conflict.
- maintains the balance property. Each vertex ends up with an equal number of Black vertices on every path to a leaf below it.

Furthermore, the number of Black vertices on each path from the root of the subtree to the leaves is *exactly the same after the fix as it was before!* In other words, the subtree looks exactly the same to the vertices above GP in terms of the number of Black vertices encountered on paths to the leaves.

This seems like magic. We added a vertex, did some twisting and juggling, and the paths didn't get any longer? And this happens every time? How is that possible?

Well of course the paths do get longer – it's just that we don't count the Red vertices.

Note that the cases that involve rotation produce a modified subtree with a Black vertex at its top. This means that there will be no more Red-Red conflicts for this insertion, so the fix is complete. However, when we apply the fix for Case 1 we end up with a Red vertex at the top of the subtree. If this vertex has a Red parent we have a new problem to fix. But note that the new problem is higher up the tree (closer to the root). That means that even if we keep on introducing new Red-Red conflicts, eventually we will get to the root of the tree, where we can resolve the Red-Red conflict simply by colouring the root Black (which we are required to do anyway).

The decision regarding which case applies is based completely on the local structure - there is no randomness or calculation involved. The decision sequence is this:

```
if Current and Parent are both red:
     if Grandparent’s other child (Sibling) is red:
          Colour Grandparent Red
          Colour Parent and Sibling Black
     elsif the Current and the Parent are both on the “same side”:
          Do a single rotation
     else:
          Do a double rotation
```

In every case the number of operations is fixed and takes constant time. As mentioned above, fixing a Case 1 problem may introduce a new Red-Red situation, so we move up the tree and fix the new problem. Since we always move upwards we will do at most one fix on each level of the tree, each requiring constant time. Thus the complexity of rebalancing the tree is the same as the complexity of finding the insertion point.

Using Parent pointers as the text does, the balancing can be done iteratively. The rotation operations require more updates since every time a child pointer is changed, the corresponding parent pointer must also change.

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. (This is why the earlier insertion algorithm was written that way.)

The logic of the RB recursive 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 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
                  // check the children of v.left_child
                  if  (v.left_child.left_child.colour == R) ||
                        (v.left_child.right_child.colour == R):
                        // Houston, we have a problem
                        // now 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 udr
            // 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
    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
               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) ||
                    (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)
                    else:
                         return Left_Right_fix(v)
                    // Note: we handle Case 1 inside these "fix"
                    // methods.
                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
           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):
               // It's all good.  Nothing to see here folks.
               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.left_child.colour == Red) ||
                    (v.right_child.right_child.colour == Red):
                    // OH NO!
                    // Now we identify the problem case
                    if (v.right_child.left_child.colour == Red):
                         return Right_Left_fix(v)
                    else:
                         return Right_Right_fix(v)
                    // Note: we handle Case 1 inside these "fix"
                    // methods.
               else:
                    // false alarm
                    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):
     // GP's left child is Red, and that child's left 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 single rotation
       // fix the pointers
       GP.left_child = P.right_child
       P.right_child = GP
       // fix the colours
       P.colour = Black
       GP.colour = Red
       // return the new root of this subtree
       return P

def Left_Right_fix(GP):
     // GP's left child is Red, and that child's right child is
     // 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
       // fix the pointers
       C = P.right_child
       P.right_child = C.left_child
       GP.left_child = C.right_child
       C.left_child = P
       C.right_child = GP
       // fix the colours
       C.colour = Black
       GP.colour = Red
       // 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(current) - you
     // can write this
```

An interesting observation about the RB balancing operations is that when we are in the process of an insertion, 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) 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 disadvantage affect the big O classification of the algorithm, but they can affect the real time performance.

Oh, if only there were some way to retain the advantages (at least, the advantages that I perceive!) 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!

For exercise, try implementing the RB insertion algorithm using a stack. If you do, you can be justifiably proud of yourself.

Deletions from R-B 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 need to cover them in CISC-235.

An important restriction on R-B 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 is essential? (Hint: think about what might happen after a rotation on a tree that contains duplicate values.)

The pseudo-code for “insert” without all that pesky 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) ||
                    (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.left_child.colour == Red) ||
                    (v.right_child.right_child.colour == Red):
                    if (v.right_child.left_child.colour == Red):
                         return Right_Left_fix(v)
                    else:
                         return Right_Right_fix(v)
               else:
                    return v
```

And now the “fixers”:

```
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


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(current) - you
     // can write this
```