Red-Black Trees Part 1

Robin Dawes

February 17, 2021

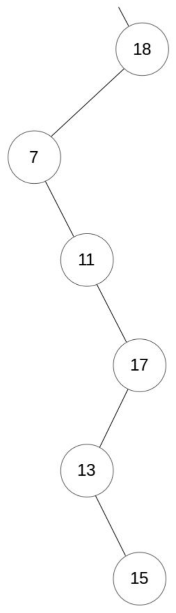
INTRODUCES and develops the most complex data structure covered in CISC-235: the Red-Black Tree. These structures have the admirable property that they guarantee $O(\log n)$ time for insertion, deletion and search operations.

.

Motivation and History

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 would guarantee that the maximum time for searching, inserting and deleting values is always in $O(\log n)$ because our algorithms for these operations have running time proportional to the height of the tree.

But 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: each vertex may have 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.



EXAMPLE: 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 $O(\log n)$ complexity for the three operations. One idea would be to rebuild the tree into its ideal shape after each insertion or deletion ... but that would be a lot of work. It would bring the complexity of each insert/delete back up to $O(n)$.

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 but we probably wouldn't have to do it very often. With luck, after rebuilding the tree we could do a lot of inserts and deletes before we had to rebuild again. Thus the *average* amount of extra work we do for each insert and delete would be quite small. This would be a reasonable solution if we don't mind a significant delay once in a while, in exchange for very fast performance almost all the time.

I strongly recommend figuring out how!

What we will see now is that it is possible to take another approach: we can keep the number of levels small - and thereby guarantee fast operations - 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. AVL trees are more compact than Red-Black trees but they require more frequent adjustments to stay in balance.

DEFINITION: A RED-BLACK TREE is a binary search tree in which each vertex is coloured either Red or Black.

Red-Black trees obey the following structural 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. All internal vertices have 2 children. The children of a Red vertex must both be Black.
5. At each vertex, all paths leading down to leaves contain the same number of Black vertices.

Regarding the colour information for vertices: 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 each vertex has an associated string containing "Red" or "Black".

The definition of Red-Black Trees 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 $2x$.

From the definition we can quickly show that in a Red-Black Tree each vertex has 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:

THEOREM: The longest path down from any vertex in a Red-Black Tree to a leaf is no more than twice the length of the shortest path down from that vertex to a leaf.

PROOF: Let v be any internal vertex in a Red-Black Tree, and let the longest path from v down to a leaf have length k. Let b be the number of Black vertices in this path. Every Red vertex in this path must have a Black child, so the number of Red vertices must be $\leq b$. Thus $k \leq 2b$

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

Putting these together gives $k \leq 2b \leq 2s$ .... 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. Yet by satisfying these local rules we obtain the desired "global" property that the whole tree has $O(\log n)$ height. This means that if we can guarantee that our insertions and deletions always maintain the local requirements, we never need to worry that the height of the tree is growing out of control.

We will see that 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 to implement.

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

Inserting A New Value into a Red-Black Tree

As WITH ANY binary search tree, there is exactly one legal place for a new value to be inserted. 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.

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

Once the insertion point has been found, the insertion process is 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. 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.

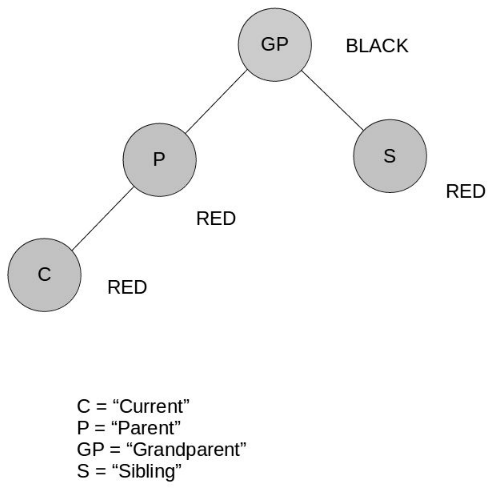
The CLRS text[1] uses Vertex objects that have Parent pointers, and gives very clear pseudo-code for the entire insert operation. The basic idea is that whenever we are currently at a RED vertex with a RED parent we need to fix something to satisfy the RB Tree rules. We do this by looking at the grandparent of the current vertex. We know three things about the grandparent:

[1] Cormen, Leiserson, Rivest, Stein

1. It must exist (because the Red parent cannot be the root)
2. It must be Black (because the tree did not contain any Red-Red conflicts before the insertion).
3. 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 "P" vertex as the left child of the "GP". There are mirror image cases where the P is the right child of the GP.

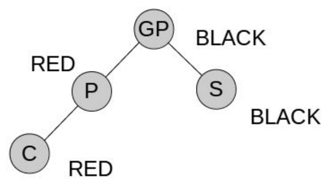
Case 1: The Sibling is RED



Case 1

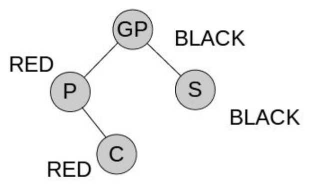
Case 2: The Sibling is BLACK

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

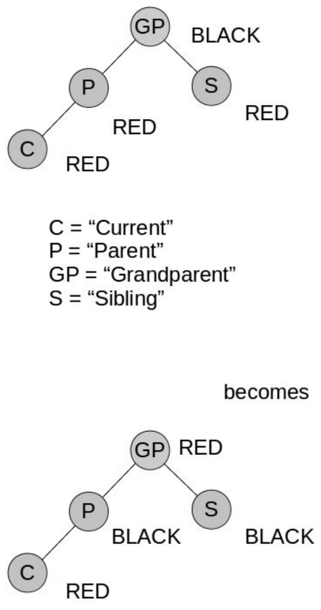
Case 2.2: The Current and Parent are on different sides (ie. one is a left child and the other is a right child).



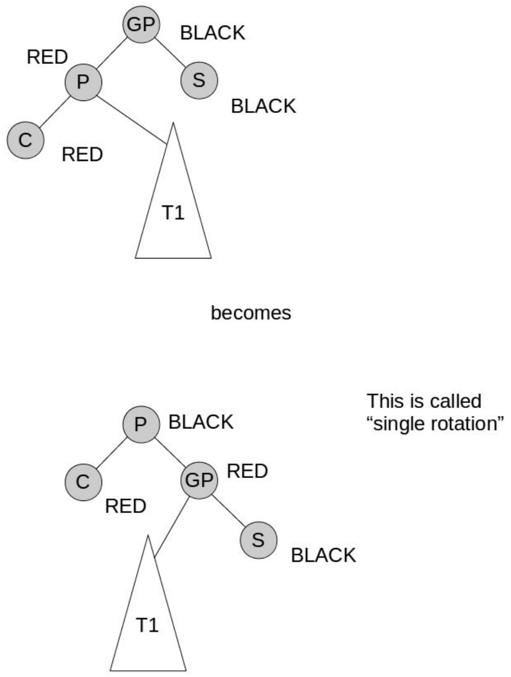
Case 2.2

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 (S).

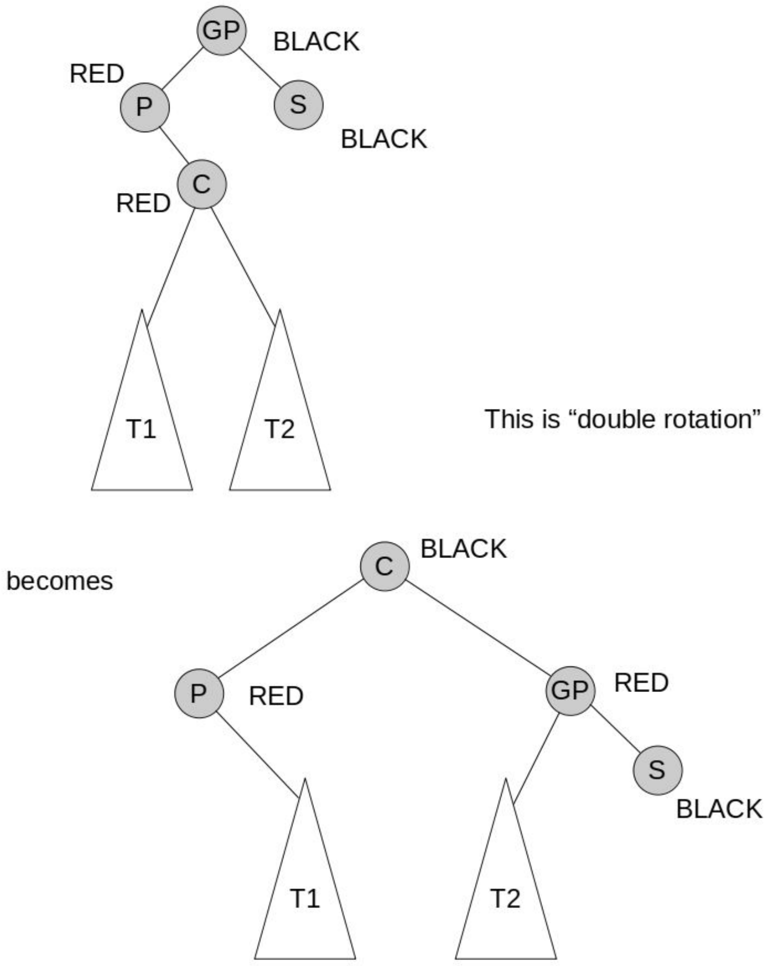


Case 2.1 Do a SINGLE ROTATION and recolour the vertices.



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 P 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.

The two crucial things to understand about each of these fixes are that each of them

- 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* the insertion! 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 (and then fixing them), eventually we will get to the root of the tree where there cannot be a Red-Red conflict because the root is always Black.

This explains why we have *that* rule!

The decision regarding which case applies is based completely on the local structure - there is no randomness or calculation involved, and we never have to look at the values in the vertices. This is the decision sequence:

```
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 previously, 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 fix requiring constant time. Thus the complexity of rebalancing the tree is the same as the complexity of finding the insertion point.

In the next note we will look at the implementation of the insert algorithm.