COS 226 Precept 8 Class Notes Fall 2023

Robert E. Tarjan

October 7 2023

1 October 6: Left-leaning red black trees

We discussed two problems concerning left-leaning red-black (LLRB) trees, both of which we left only partially solved. Here I'll present complete solutions.

1.1 The structure of LLRB trees

Suppose one is given an uncolored binary tree. How can one test whether there is a way to color the links to make it an LLRB tree? Let us review the requirements for a tree to be LLRB. I'll state the conditions in terms of node coloring rather than link coloring: Recall from lecture that if a link connects a parent node x with a child node y, we store the color of the link in the child node y.

Black rule: The root and all null nodes are black. All paths from the root to a null node contain the same number of black nodes.

The black rule is what makes the tree balanced. Red nodes provide the flexibility to insert (or delete) an item in the tree while keeping it an LLRB tree with only O($logn$) rotations and node color changes.

Red rule: Every red node is the left child of a black node.

The red rule implies that every right child is black. This together with the black rule implies that in any subtree the node with the largest key has smallest depth among nodes that have at least one null child. (Other nodes in the subtree may have the same depth.) (Note: In precept I got this backward, saying incorrectly that the largest node has *largest* depth among *all* nodes in the subtree.) This "smallest depth" condition is necessary for an uncolored tree to be colorable in a way that obeys the black and red rules, but it is far from sufficient.

Here is a condition that is both necessary and sufficient: Define the *right height* of a node in a binary tree to be the number of nodes on its right spine, defined to be the path from it to the node in its subtree containing the largest key. As a degenerate case, if the node is a null node, its right height is zero.

The nodes of an uncolored binary tree can be colored to make it an LLRB tree if and only if the tree has the following *right-heights* property: For each node x in the tree, either the right heights of $x.left$ and $x.right$ are equal, or the right heights of $x.left.left$, $x.left.right$, and $x.right$ are all equal.

Let's prove that this is true. Suppose a tree is an LLRB tree. Let x be a node in the tree. If x is a red node, or a black node with two black children, the black rule implies that the right heights of $x.left$ and $x.right$ are equal. If x is a black node with a red left child, the black rule implies that the right heights of $x.left.left$, $x.left.right$, and $x.right$ are all equal.

Conversely, suppose that an uncolored tree has the right-heights property. Color the root and all right children black. If $x$is a left child, color it black if its right height equals that of its sibling (the right child of its parent); otherwise, color it red. This coloring obeys the red rule. A proof by induction on increasing subtree size shows that the coloring also obeys the black rule.

Not only does this show that an uncolored tree that has the right-heights property can be colored to make it an LLRB tree, it shows that the coloring is unique. This is not in general true of other kinds of red-black trees.

The right-heights property is easy to test in time linear in the number of nodes in the tree, and also easy to apply.

1.2 Deletion in binary search trees

We started discussing deletion in arbitrary binary search trees and in LLRB trees. Here I'll develop a complete algorithm. I suggested you look in the textbook for an algorithm, but the one there is not so satisfactory (at least, I don't think so).

The first step in developing a way to do deletion in LLRB trees is to figure out how to do it in arbitrary binary search trees, ignoring the issue of balance. To delete an item, we search for the node containing it, say x. If x is a leaf (both its children are null), we merely delete it. If it is the root of the tree, the tree becomes empty. If x is unary (it has one non-null child and one null child), we replace x by its non-null child. That is, if x is the left (right) child of y, we set $y.left$ ($y.right$, respectively) equal to the non-null child of x. If x has no parent (it is the root of the tree), its non-null child becomes the root of the tree. Beware: Binary

search tree algorithms have many cases, including corner cases. If you ever have to implement algorithms on such trees, make sure to cover all the cases.

There is one more case, the most challenging one. Suppose x is a binary node. (It has two non-null children.) We cannot merely delete it, for then its parent would acquire two children in its place, not one, and the tree would no longer be binary. Instead we find the *successor* y of x, the node containing the next higher key, by going to the right child of x and proceeding through left children until reaching a node y with a null left child. Then we swap the items in x and y. A symmetric alternative is to find the *predecessor* w of x, the node containing the next lower key, and swapping the items in w and x.

After the swap, the item to be deleted is in a leaf or a unary node, and we can safely delete it as already described. One does not of course have to complete the swap: It is enough to move the replacing item into the node containing the item to be deleted, and then delete the newly vacant node.

1.3 Deletion in LLRB trees

When deleting an item from an LLRB tree, we must do more than delete the node containing it (after a swap, if a swap is needed to move it out of a binary node). Deletion of the node may have violated the black rule, and we need to restore it. Let's devise a way to do this.

Deleting a leaf or unary node that is red does not violate the balance rules, so this case requires no rebalancing. But if the deleted node is black we need to rebalance, since its deletion violates the black rule. To handle this, we "paint it black."

More precisely, if the deleted node is unary, we change the color of the node replacing it (its old child) to black if it is red or to double black if it is black. A double black node counts two rather than one when computing the number of black nodes on a path. If the deleted node is a leaf, we color the new null child of its parent double black.

This coloring restores the black rule, but it leaves us with an illegal coloring: There is a double black node. Just as in the case of insertion, we need to do local changes to the colors and the tree structure (the latter by doing left and right rotations) to eliminate the violation or push it up toward the root. If the root becomes double black we make it black, finishing the rebalancing. This case is the only one in which the black height of the tree changes, decreasing by 1. It is the opposite of what happens during an insertion, in which a color flip can make the root red, followed by a corner case color flip that makes it black again, increasing the black height of the tree by 1.

Now it is a matter of finding a set of local transformations that will do the job. The set of transformation I'll present work, but they have one annoying feature: Some of them eliminate the double black node but create a violation of the red rule, which must then be fixed by using the insertion transformations.

There are six cases we need to handle, depending on the colors of the parent and sibling of the double black node and on the colors of the children of its sibling. Let x be the double black node, y its sibling, and z its parent. Node y must be non-null by the black rule (even if x is null).

> Case 1: The parent z of x is red. Color the sibling y of x red, and color x and z black. This amounts to an inverse color flip. This preserves the black rule and eliminates the double black, but it may violate the red rule, because y could be the right child of z, and it might have a red left child. We fix this as in an insertion, but there is a extra case that does not arise in an insertion, that of a red right child that has a red left child. If this is the case, we do a right rotation on the left red link to make it a right link, then a left rotation on the top right red link. This results in a node with two red children, a case that does occur in insertion. We do a color flip and proceed as in an insertion.
>
> Case 2: The sibling y of x is red. y is a left child of z, and z is black. Both children of y are non-null and black. We rotate the red link connecting y and z. This y black and z red. Now x has a red parent (z), and we do an inverse color flip, making x and z black and the new (left) sibling of x red. (This sibling was previously a child of y and is black.) This eliminates the double black but may violate the red rule: The new red node may have a red left child. We fix the violation of the red rule as in an insertion.

In the remaining four cases the parent and sibling of x are black. The appropriate case depends on whether x is a left or right child and on whether its sibling has a red child.

Case 3: Node x is a right child and both children of its sibling y are black. We color x black, y red, and z double black. This preserves the balance rules and moves the violation to the parent of x, pushing it up the tree.

Case 4: Node x is a left child and both children of its sibling y are black. This case is symmetric to Case 4; because of the asymmetry of the red rule it requires a rotation. We color x black, y red, and z double black, and rotate the red link connecting y an z. This swaps the colors of y and z, making y double black and z red. It preserves the balance rules and pushes the violation up the tree.

In the two remaining cases y, the sibling of x, has a red left child. By doing appropriate rotations and recolorings, we can eliminate the double black without violating the red rule.

Case 5: Node x is a right child and its sibling y has a red left child. Ignoring node colors, rotate the left link connecting y and z. Color x, y, and z black. This preserves the balance rules.

Case 6: Node x is a left child and its sibling y has a red left child. Ignoring node colors, rotate the right link connecting y and z. Color x, y, and z black. This preserves the balance rules.

I strongly encourage you to draw pictures of these cases and verify that they are complete and correct. The time for rebalancing after a deletion is at most a constant times the tree height and hence O($logn$), just like an insertion.

Note that Case 1 is actually two cases, because the transformation depends on whether x is a left or right child. This there are seven cases, plus the three needed in insertion (color flip, left rotation, right rotation), for a total of ten. LLRB-tree deletion is indeed complicated!

1.4 Variants of red-black trees

The design space of balanced trees is enormous; LLRB trees are only one point in this space. They were chosen to make insertion simple to code, but they have some disadvantages. For example, we have seen that deletion is complicated. Also, an insertion or deletion can require $\Theta(\log n)$ structural changes (rotations), even in the amortized case.

Among the possible variants of red-black trees, one I like much better than LLRB trees is the *weak AVL tree*. We can define the structure of weak AVL trees using node colors. Instead of using red and black, we use black and double black. A weak AVL tree is a tree whose nodes are colored black or double black and which obeys the following rule:

Black rule: All null nodes are black. All paths from the root to a null node have the same total count, where a black node counts 1 and a double black node counts 2.

In weak AVL trees, an insertion or deletion requires at most two rotations in the worst case. There are six cases in insertion rebalancing and eight in deletion rebalancing; the cases occur in symmetric pairs. Weak AVL trees have many other nice properties that we may explore in COS 423.